EVALUATING STANDARD AND CUSTOM APPLICATIONS IN IPV6 WITHIN A SIMULATION FRAMEWORK

Brittany Michelle Clore

Thesis submitted to the faculty of the Virginia Polytechnic Institute and State University in partial fulfillment of the requirements for the degree of

Master of Science
In
Computer Engineering

July 30, 2012
Blacksburg, VA

Keywords: IPv6, Simulation, Custom Application, Framework, OPNET

EVALUATING STANDARD AND CUSTOM APPLICATIONS IN IPV6 WITHIN A SIMULATION FRAMEWORK

Brittany Michelle Clore

ABSTRACT

Internet Protocol version 6 (IPv6) is being adopted in networks around the world as the Internet Protocol version 4 (IPv4) addressing space reaches its maximum capacity. Although there are IPv6 applications being developed, there are not many production IPv6 networks in place in which these applications can be deployed. Simulation presents a cost effective alternative to setting up a live test bed of devices to validate specific IPv6 environments before actual physical deployment. OPNET Modeler provides the capability to simulate the IPv6 protocol and System-in-the-Loop, an add-on module, allows for real communication traffic from physical devices to be converted and sent over the simulated network. This research has developed a campus framework, modeled after the Virginia Tech Blacksburg campus, to verify and validate standard and custom IPv6 applications. Specifically, the framework was used to test MT6D, a custom IPv6 security application developed in the Virginia Tech IT Security Lab (ITSL) as well as test Voice over IP (VoIP) as a somewhat bandwidth demanding benchmarking standard application. The work presented shows that simulation helped to identify potential issues within the applications and verified the results after fixes were applied. It also reveals challenges and shortcomings of OPNET Modeler's IPv6 implementation and presents potential solutions to these problems.

ACKNOWLEDGEMENTS

There are several people that have been an integral part of my time at Virginia Tech. They have been instrumental in guiding me through my education, helping through the difficult times and providing needed support.

I would like to thank Dr. Tront for being my committee advisor and guiding my education throughout my time at Virginia Tech. My first college experience sitting in your Intro to Computer Engineering class made me realize that I made the right choice for school. I know that my graduate experience has been a positive one because of your help and guidance.

I would like to thank Randy for giving me a spot in the lab and providing a wealth of information that could come from no other person. By being around you and the lab, I know that I've learned more through osmosis than any other place I could have been. The light hearted spirit of the lab has only come from your example and I hope that it continues to be that way far into the future.

I would like the thank Dr. White for being on my committee and willing to deal with all the scheduling changes.

I would like to thank Lt. Colonel Matthew Dunlop for allowing me to be a part of his research. I would not have survived this degree without your help and guidance. You were always willing to teach me about networking skills and protocols that I did not get in my classes and help me through my writing process. I'm glad that we were able to work through all the various simulation issues to get the results we needed, even if it was by working overnight and through weekends.

I would like to that the undergraduate workers in the lab, particularly Reese and Matt, for helping with VoIP testing. I enjoyed all the fun "nerdy" conversations on code optimization, movies and computers.

My friends deserve great thanks for putting up with me throughout my education. Without my friends in Blacksburg, I would have led a very boring life.

Last but not least, I need to thank my family. Without my parents, I would not be in the place that I am today. There are not enough words to express my deep appreciation for all that they have done for me. My siblings also have helped me through by reminding me to have fun and enjoy the little things.

Romans 8:28 - And we know that all things work together for good to those who love God, to those who are the called according to His purpose.

TABLE OF CONTENTS

Table of Contents iv
List of Figures vii
List of Tables ix
1. Introduction 1
2. Background 3
2.1 IPv6 3
2.1.1 Addressing 3
2.1.2 Header Format Comparison and Flow Labeling Capabilities 6
2.1.3 Support for Extensions 9
2.1.4 ICMPv6 10
2.1.5 NDP 13
2.2 Voice over IP 14
2.3 Moving Target IPv6 Defense 17
2.4 Chapter Conclusion 18
3. OPNET Modeler and the System-in-the-Loop Module 19
3.1 OPNET Modeler Overview 19
3.2 The System-in-the-Loop Module 21
3.3 The Advantages and Disadvantages of OPNET Modeler 23
3.4 The Use of Modeler in this Research 24
3.5 Chapter Conclusion 24
4. Simulation Tools and Related Research 26
4.1 Simulation and Emulation Tools 26
4.2 Related Research on Simulation Frameworks 28
4.3 Chapter Conclusion 30
5. Configuration and Network Topology Design 31
5.1 Live Network Configuration 31
5.1.1 Virtual Machine Configuration 31
5.1.2 Real Machine Configuration 35
5.2 Network Topology and Framework 37
5.2.1 Simulated Network Devices 37
5.2.2 Topologies 39
5.2.2.1 One Hop 39

5.2.2.2 Two Hop 40
5.2.2.3 Two Hop with Background Traffic 40
5.2.2.4 Campus Topologies 41
5.2.3 Traffic Determination 42
5.3 Chapter Conclusion 43
6. OPNET Modeler and SITL IPv6 Evaluation 44
6.1 Internal to Internal Evaluation 45
6.1.1 Traffic types 45
6.1.2 Testing Generation Traffic 49
6.1.3 Testing Flow Traffic 51
6.1.4 Testing Application Traffic 54
6.1.5 Conclusion for Internal to Internal 54
6.2 Internal to External Evaluation 55
6.3 External to Internal Evaluation 55
6.4 External to External Evaluation 57
6.4.1 Connectionless Traffic Test 57
6.4.2 Connection-Oriented Traffic Test 60
6.4.3 Extension Header Test 61
6.4.4 SITL Queuing Delay Tests 63
6.4.4.1 Hypothesis 1: Physical Machine Resources 64
6.4.4.2 Hypothesis 2: Bit Architecture 65
6.4.4.3 Hypothesis 3: Translation Function 66
6.4.4.4 Queuing Delay Test Conclusion and Benchmark Test 66
6.4.5 External to External Conclusion 67
6.5 Chapter Conclusions 67
7. IPv6 Application Simulation and Results 68
7.1 Methodology for VoIP 68
7.2 VoIP Results 69
7.3 Methodology for MT6D 72
7.4 Unexpected Behavior from the First Set of Results 72
7.5 MT6D Results 75
7.6 Chapter Conclusions 79
8. Conclusions and Future Work 80
References 82

APPENDIX 86
APPENDIX A: IPv6 Traffic Generator process function code 86
APPENDIX B: SITL Custom Translation Function 96
APPENDIX C: VoIP Metrics Python Script 116
APPENDIX D: Framework User Manual 118
APPENDIX D.1 Components of the Simulation Interface 118
APPENDIX D.2 SITL Configuration 124
APPENDIX D.3 Example 1: Running a Simulation 126
APPENDIX D.4 Example 2: Configuring a Traffic Profile 128

LIST OF FIGURES

Figure 2.1 Dot-decimal and Hexadecimal Notations of the Same 128 bits 4
Figure 2.2 IPv6 CIDR Notation 4
Figure 2.3 Steps to Create an IPv6 Address 5
Figure 2.4 IPv6 Header Format 7
Figure 2.5 IPv4 Header Format 7
Figure 2.6 NDP Interaction 14
Figure 2.7 SIP Establishment Process 16
Figure 3.1 OPNET Modeler’s Levels of Abstraction 20
Figure 3.2 The Three Configurations of SITL 22
Figure 5.1 Configuration of VMs on a Host Machine 33
Figure 5.2 SITL Attributes Menu 34
Figure 5.3 Comparison Ping Flood Test Results 35
Figure 5.4 End Hosts 38
Figure 5.5 Networking Devices 38
Figure 5.6 One Hop Topology 39
Figure 5.7 Two Hop Topology 40
Figure 5.8 Two Hop with Background Traffic Topology 41
Figure 5.9 Office Topology 42
Figure 6.1 Traffic Generation Menus: *Ethernet_ip_station* (left), *Ethernet_rpg_station* (right) . 46
Figure 6.2 Flow Attributes Menu 47
Figure 6.3 Profile and Application Modules 48
Figure 6.4 Application Configuration Menu 48
Figure 6.5 Process Model of *Ethernet_ip_station* 50
Figure 6.6 Ethernet Workstation Process Module 51
Figure 6.7 Flow Results and Configuration Menu 53
Figure 6.8 Configuration Menu with 50% Implicit to Explicit Ratio 53
Figure 6.9 Malformed Router Advertisements in Wireshark 56
Figure 6.10 Debugging Console with Router Prefix 56
Figure 6.11 Invalid ICMP Message Error 58
Figure 6.12 SITL Queuing Delay Variation in a Ping Test 59
Figure 6.13 Rise and Fall Pattern of the SITL Queuing Delay 60

Figure 6.14 Website Accessed Through Simulation 61
Figure 6.15 Log Showing Unsupported Destination Options Extension Header 62
Figure 6.16 Performance Monitor Report 65
Figure 7.1 MT6D Dropped Rotations 74
Figure 7.2 Updated MT6D with No Dropped Rotations 74
Figure 7.3 Average Throughput with No MT6D 75
Figure 7.4 Transfer Speeds of Simulated MT6D One Hop File Tests 76
Figure 7.5 Transfer Speeds of Live Network MT6D File Tests 76
Figure 7.6 Retransmissions for Simulated MT6D One Hop Tests 77
Figure 7.7 Packet Loss Comparisons between Live and Simulated Pings 78
Figure D.1 OPNET GUI 119
Figure D.2 Edit Attributes 119
Figure D.3 Statically Setting an IP 120
Figure D.4 Rapid Deployment 120
Figure D.5 Switching Scenarios 121
Figure D.6 Defining Statistics 122
Figure D.7 Choosing Statistics 122
Figure D.8 SITL Attribute Menu 124
Figure D.9 SITL Real-time Ratio 125
Figure D.10 Default Simulation Menu 127
Figure D.11 Application Configuration Menu 129
Figure D.12 Configuration Menus 130
Figure D.13 Deploy Defined Applications 131
Figure D.14 Deployment Menu 131

LIST OF TABLES

Table 2.1 Extension Header Values 10

Table 2.2 ICMPv6 Message Types from IANA 12

Table 5.1 MRTG Average Traffic 43

Table 6.1 IPv6 Evaluation Results 45

Table 6.2 Extension Header Test Results 62

Table 7.1 VoIP Live Network Results 70

Table 7.2 VoIP Latency Results 70

Table 7.3 VoIP Jitter Results 71

1. Introduction

The motivation for examining simulation software for Internet Protocol version 6 (IPv6) readiness comes from the need to be able to test applications with that protocol as IPv6 deployment proliferates throughout the world. Although IPv6 was introduced in 1998 with RFC 2460 [1], there remains a slow adoption rate for various reasons including software changes to include protocol support and hardware changes to the network infrastructure. One example of a needed software change is additional support in Intrusion Detection Systems to look for IPv6 addresses. An example of a hardware change to the network infrastructure is physical deployment of routers that support IPv6. Simulation schemes are used to ensure that these hardware and software changes are able to be operational in a network environment; therefore, the simulation software must be able to support IPv6 to test the functionality of existing software applications. Newly developed applications that use IPv6 also require testing before deployment on a network. One such new application is the Moving Target IPv6 Defense (MT6D) system [8]. The research presented in the following pages helps verify the functionality of this specialized IPv6 application. Simulation software was chosen based on certain requirements of MT6D, namely IPv6 support and communication between live and simulated networks. OPNET Modeler, which met these requirements, was chosen and subsequently evaluated as a separate entity. It was then used to create a framework to test one standard and one custom application, specifically Voice over IP (VoIP) and MT6D.

The methodology for creating the framework included research into simulation products, literature review for other studies done using network simulation and IPv6, an evaluation of IPv6 traffic support in OPNET, and determining the configuration options between live devices and simulation. Application testing follows the methodology of running the tests first on a live network then in simulation. The results from both tests are compared for two reasons: to determine that simulation topologies with background traffic are configured to match a live environment and to verify applications run without unexpected errors. Further tests can be done in simulation to pinpoint the cause of the failure if unexpected errors occur.

The thesis is organized as follows: Chapter 2 gives background information on IPv6, VoIP, and MT6D. Chapter 3 provides an introduction to OPNET Modeler. Chapter 4 gives information on other simulators as well as a review of related work and literature. Chapter 5 outlines the configuration of Modeler and SITL and the design of the network topology framework. Chapter 6 evaluates the simulation software for IPv6 readiness and presents the actions taken to overcome any shortcomings. Chapter 7 explains the testing of VoIP and MT6D in simulation and presents the results from these tests. Chapter 8 is the conclusion and discussion of future work.

2. Background

Background information on IPv6, VoIP, and MT6D is necessary to understand the work done in this thesis. IPv6 is the main network protocol the simulation software is being evaluated for and that is used by the applications. VoIP is the custom application for IPv6 chosen for simulation and has several components that need to be explained for a complete understanding. Finally, MT6D is the custom IPv6 application that has been simulated which has a unique process for creating addresses. In addition to address creation, MT6D has several configuration options available to the user including encryption. The essentials of IPv6 and the two test applications will be explained in the following sections.

2.1 IPv6

IPv6 is the latest generation of Internet protocol designed as a successor to Internet Protocol version 4 (IPv4) [1]. There are four areas that IPv6 extends IPv4: addressing, header format, support for extensions and flow labeling capabilities [1]. Not only are these areas different between IPv6 and IPv4, IPv6 uses different protocols to perform network discovery. These protocols are Internet Control Message Protocol version 6 (ICMPv6), used to relay informational and error messages, and the Neighbor Discovery Protocol (NDP), used to facilitate routing. The four extension areas, ICMPv6, and NDP will be described by comparing them to their IPv4 counterparts.

2.1.1 Addressing

IPv4 uses 32 bits to contain the address for each node. IPv6 increases the address size to 128 bits to “support more levels of addressing hierarchy, a much greater number of addressable nodes, and simpler auto-configuration of addresses” [1]. In other words, IPv6 supports larger networks and subnets than IPv4 and the allocation of the IP addresses is easier. The maximum amount of addresses in IPv4 is 2^{32}which is approximately 4.2 billion. In comparison, IPv6 has a maximum of 2^{128}addresses which is approximately 340 undecillion (10^{36}). This vastly larger address space requires the use of hexadecimal (hex) notation instead of IPv4's dot-decimal notation for readability purposes as hex notation reduces the number of characters that need to be written or remembered for the address. Leading zeros can be dropped and one set of repeated zeros can be truncated with a double colon ("::") to further enhance readability. The same 128-bit IP address in dot-decimal notation and in hexadecimal notation is shown in Figure 2.1. The hexadecimal representation is shorter than the dot-decimal notation.

$$Dot - decimal\ notation\text{: }128.32.0.156.23.147.0.193.1.29.0.0.0.0.0.1$$
$$Hexadecimal\ notation\text{: }8020\text{: }9c\text{: }1793\text{: }c1\text{: }11d\text{: : }1$$

Figure 2.1 Dot-decimal and Hexadecimal Notations of the Same 128 bits

Increasing the addressing space directly increases the addressing hierarchy. Addressing hierarchy refers to divisions between networks, i.e., subnets, which are signified by prefixes (a leading set of bits in the address). Prefixes are analogous to telephone area codes. The notation signifying the prefix remains the same as in IPv4 despite the increased number of addresses and is called the Classless Inter-Domain Routing (CIDR) notation. CIDR notation designates the prefix by adding a forward slash ("/") then the number of leading bits that determine the subnet at the end of the IP address. An IPv6 address using CIDR notation is presented in Figure 2.2. It signifies it belongs to the subnet 8020:9c:1793:c1 (note that leading zeros are dropped in hex notation) which is the leading 64 bits of the address.

$$8020\text{: }9c\text{: }1793\text{: }c1\text{: }11d\text{: : }1/64$$

Figure 2.2 IPv6 CIDR Notation

Another impact of the increased number of bits allocated to addresses is simpler auto-configuration. In IPv4, complete addresses are usually allocated to nodes by the network infrastructure. IPv6 uses a scheme called Stateless Auto-Address Configuration (SLAAC) in which the network provides the first 64 bits of the address, i.e., the subnet, and the node itself provides the latter 64 bits with a modified unique identifier. This modified unique identifier is created by taking the Media Access Control (MAC) address of the node, inserting 16 extra bits with the hexadecimal value of "FF:FE" in the middle of this address, and flipping the 7th most significant bit. The result is called the Modified Extended Unique Identifier 64 (EUI-64). The Modified EUI-64 is the same for a single node across all networks. The subnet prefix is given to the device through ICMPv6 messages generated from the network router which is described in section 2.4. Each part of creating the SLAAC address is shown in Figure 2.3. The first part is the MAC address, the second part is the Modified EUI-64 and the last part is the subnet prefix. Each new addition to the address is highlighted in red. Addresses can also be assigned through Dynamic Host Configuration Protocol version 6 (DHCPv6) that has a central management system which assigns addresses. This scheme is similar to DHCP addressing in IPv4 and is not necessarily simpler when done in IPv6.

$$MAC\ address: FC:75:12:43:75:7B$$

$$Modified\ EUI-64: F\boldsymbol{E}75:12\boldsymbol{FF}:\boldsymbol{FE}43:757B$$

$$IPv6\ address: \mathbf{2005:0:}\boldsymbol{C}\mathbf{11}\boldsymbol{D}\mathbf{:5}:FC75:12FF:FE43:757B$$

Figure 2.3 Steps to Create an IPv6 Address

In addition to the SLAAC address, nodes can have several other types of IPv6 addresses which are used on different areas of networks and denote different types of communication. The addresses that designate the area of network they are used on include the loopback address, the link-local address, and specific global addresses; these are all unicast address, one to one communication. The loopback address is an address that points back to the device itself and is used mainly for management and testing purposes. In IPv6, this address is "::1/128" on all devices. The link-local address, which is usually formed with the Modified EUI-64 and the subnet "fe80::/64", is an address that is used between two communicating nodes that are on the same local network [52]. A router is not necessary for nodes to communicate over the link-local address and routers cannot forward any of these packets. The link-local address also acts as the gateway for the node to the network for auto-addressing and Neighbor Discovery (discussed in section 2.1.5). Specific global addresses are addresses that are available to communicate on the Internet but designate a characteristic about the network. For example, 6to4 is a tunneling mechanism that allows IPv6 packets to be sent over an IPv4 network [53]. This mechanism is signified by the first 16 bits of the IPv6 address that have the hex value “2002.” If an address starts with that 16-bit value, then it knows to translate that packet into IPv4. Another example of a specific global address is a 6bone address [54]. 6bone was an IPv6 test bed network developed in 1996 to help facilitate IPv6 testing and transitioning. It used the prefix “3FFE::/16” in its global addresses. A third global address is the official IPv6 global network address which uses the prefix “2001::/16.” Entities that have stood up IPv6 networks can request an allocation of addresses containing this prefix from an Internet Registry, such as the American Registry for Internet Numbers (ARIN) [55].

Apart from these unicast addresses, there are IPv6 prefixes for multicast and anycast addresses. Multicast addresses are ones that are used to communicate to multiple destinations at the same time (one to many) and are used by routing protocols [52]. IPv6 multicast addresses for routing protocols have the prefix “FF02::/16.” Anycast addresses are addresses that can be assigned to multiple nodes and are sent to the closest node with that address (one to one of many) [52]. Anycast addresses are assigned from the unicast space and therefore do not have a distinguishing prefix.

2.1.2 Header Format Comparison and Flow Labeling Capabilities

The second area IPv6 extends IPv4 is in the header format. The IPv6 header format, shown in Figure 2.4, has fewer fields than the IPv4 header, shown in Figure 2.5 [1] [48]. While there are fewer fields, the IPv6 header contains the same functionality that the IPv4 header provides and has more features. The fields of the headers are examined below to highlight the similarities and differences between the two protocols.

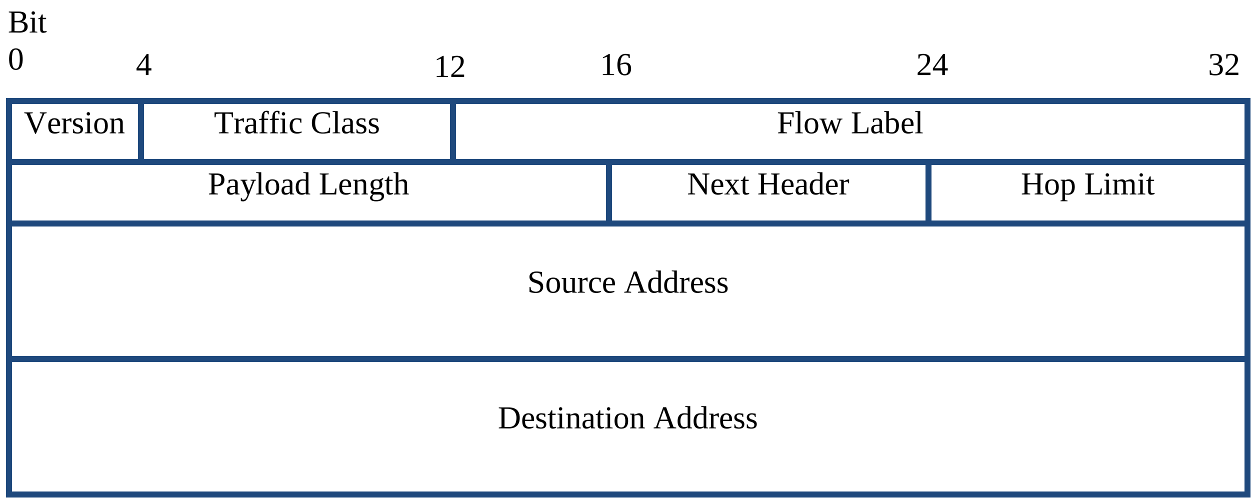


Figure 2.4 IPv6 Header Format

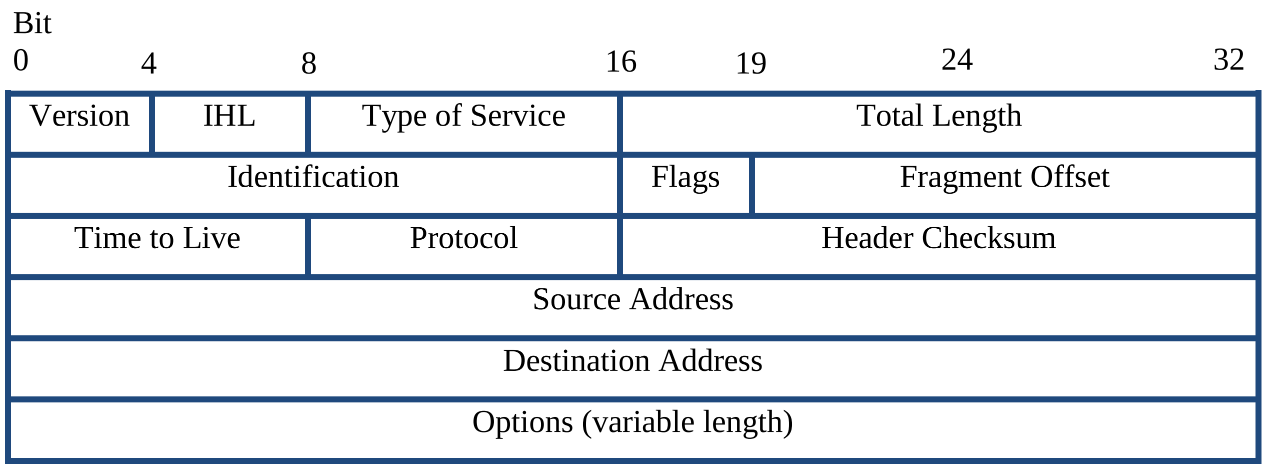


Figure 2.5 IPv4 Header Format

There are three fields in the IPv6 header format that are named the same in the IPv4 header format: Version, Source Address, and Destination Address. The Version field, a 4-bit value, in both headers signifies the protocol version of the packet. In the IPv6 header, the value of this field is always 6. The source and destination addresses fields only differ in size between the protocols; IPv4 addresses are 32 bits and IPv6 addresses are 128 bits. The remaining fields in the IPv6 header are the Traffic Class, Flow Label, Payload, Next Header, and Hop Limit which differ from the IPv4 header fields.

The Traffic Class field is an 8-bit value used to designate a priority to the IPv6 packet. This is a new field that is not included in the IPv4 header. Instead, the IPv4 header has the Internet Header Length (IHL) field which provides the length of the header. IPv4 needs the IHL field because its header is not a static length and can vary between 20 and 24 bytes. In contrast, the IPv6 header uses a fixed 40 bytes and does not need a field to indicate its size.

The Flow Label in the IPv6 header designates a label for a specific flow of packets that may have special handling requirements. A flow is a sequence of packets between one source and destination that may have specific handling requirements such as real time traffic. All packets within a specific flow must have the same label, source address, destination address, and hop limit [1]. Flow labeling is one of the four extension areas of IPv6 meaning that IPv4 does not completely provide this capability. The IPv4 header has the Type of Service field, 8 bits in size, which can be used to specify quality of service requirements but it is not widely used in current networks.

Designation of the payload length, which includes all higher layer headers and data, occurs in both IP headers but the fields are called by different names. In IPv6 the field is the Payload Length and in IPv4 it is called the Total Length; both fields are 16 bits long. In addition to payload length, a field for indicating the next protocol header in the packet occurs in both IPv6 and IPv4. The Next Header (IPv6) and Protocol (IPv4) fields contain a value that signifies the type of protocol that occurs next in the packet such Transmission Control Protocol (TCP) or User Datagram Protocol (UDP). In IPv6, the Next Header field can also point to an extension header as the next header in the packet. Extension headers are discussed in further detail in section 2.1.3. A third type of field in both headers represents the remaining allowed hops in a network and are the Time to Live and Hop Limit field in IPv4 and IPv6 respectively. These fields start with an initial value and decrement after every hop the packet encounters. The packet is discarded when these fields reach 0 to ensure that the packet will not continue in an endless loop in the network.

There are four fields in the IPv4 packet that the IPv6 designers did not carry over to the new header: Identification, Flags, Fragment Offset and Checksum. The first three relate to the fragmentation of a packet, meaning that if a packet is too large for a specific network, it can be split into parts and transmitted in these smaller pieces. Fragmentation does not occur in the basic IPv6 header; instead, this functionality was moved into the Fragmentation extension header by the designers. The Checksum field is used to indicate if any data was corrupted through transport. The reason this field was left out of the IPv6 header design is that checksums are already computed by Layer 2 protocols and therefore checksums generated in Layer 3 protocols are redundant. The designers felt that this redundancy did not need to occur.

2.1.3 Support for Extensions

IPv6 allows packets to send extra information through extension headers. The IPv4 options field was used to provide this capability, but IPv6 extension headers have more flexibility for both the length of the option information and for introducing new options in the future [1]. The extension headers are included as a part of the IPv6 packet payload instead of being in the IPv6 header and are indicated by the Next Header field in the IPv6 header. Current extension headers are defined for routing options, destination specific options, Internet Protocol Security (IPSec) headers, and mobility. Routing options extension headers are used to provide source routing which allows the originator of the packet determine the route through the network. Destination specific headers, called Destination Options, allow the sender of a packet to include specific information for the destination only. The Destination Options header is contrasted with the Hop-by-Hop extension header which contains information that must be examined at each hop the packet takes [1]. IPSec, an end to end security scheme, has two headers which can be used: the Authentication Header (AH) and the Encapsulation Security Payload (ESP) Header. Each header when used can provide authentication and integrity to the packets and the ESP Header provides encryption [56]. The mobility header is used to support Mobile IPv6, a protocol that allows devices to be reachable by a single address wherever they are in the Internet [51]. The extension headers and the corresponding value used in the Next Header field are displayed in Table 2.1.

Extension Header	Value
Hop-by-Hop Options	00
Destination Options	60
Routing Header	43
Fragment Header	44
Authentication Header	51
Encapsulation Security Payload Header	50
Mobility	135
No Next Header	59

Table 2.1 Extension Header Values

While each extension has its own defined header format with different fields, all contain a Next Header field and a Length field. The Next Header field in the extension header can be another extension or the transport layer protocol.

2.1.4 ICMPv6

ICMPv6 is a protocol that is used to transfer informational and error messages between two nodes and is essential for IPv6 communication [49]. These messages are generally used to determine the reachability of a node and if there are errors in the packet length or formation. They also are used as a part of NDP to set up routes and give addressing information to end host.

ICMPv6 is an update of ICMP, the IPv4 version of the protocol [49] [50]. The header for both protocols is the same: four fields that contain the message type, message code, checksum and the body of the message. The message type indicates what type of information, whether informational or error, is being sent. Examples of informational messages are echo requests and replies, commonly known as pings. An example of an error message is a parameter problem type which might indicate a difficulty like an invalid Next Header within the IPv6 header of a received packet. The code field is used to provide further granularity for a message. For instance, a parameter problem message can be caused by several different errors such as an unrecognized next header value, an unrecognized IPv6 option, or an unexpected header field and the code field can be used to narrow which error caused the message. ICMP uses the code field in the same manner. The third field is the checksum field which verifies that the message was not corrupted in transit. ICMPv6's calculation of the checksum differs from ICMP by including a "'pseudo-header' of IPv6 header field" along with the ICMPv6 message [49]. The fourth field contains the body of the message being sent and is the same in both ICMPv6 and ICMP.

ICMPv6 uses a different set of values for message type. For instance, ICMP uses the values 0 and 8 for echo replies and requests while ICMPv6 uses the 129 and 128, respectively. ICMPv6's message type values are separated into numerical ranges by error and informational types which does not occur with ICMP. Error message values are in the 0-127 and informational messages are in the 128-255 range. A complete list of the available ICMPv6 types provided by the IANA is in Table 2.2 [57]. Another value difference is the Next Header specification in the IP header. ICMP's protocol value is 1 while ICMPv6's is 58.

Type	Name
0	Reserved
1	Destination Unreachable
2	Packet Too Big
3	Time Exceeded
4	Parameter Problem
100	Private experimentation
101	Private experimentation
102-126	Unassigned
127	Reserved for expansion of ICMPv6 error messages
128	Echo Request
129	Echo Reply
130	Multicast Listener Query
131	Multicast Listener Report
132	Multicast Listener Done

133	Router Solicitation
134	Router Advertisement
135	Neighbor Solicitation
136	Neighbor Advertisement
137	Redirect Message
138	Router Renumbering
139	ICMP Node Information Query
140	ICMP Node Information Response
141	Inverse Neighbor Discovery Solicitation Message
142	Inverse Neighbor Discovery Advertisement Message
143	Version 2 Multicast Listener Report
144	Home Agent Address Discovery Request Message
145	Home Agent Address Discovery Reply Message
146	Mobile Prefix Solicitation
147	Mobile Prefix Advertisement
148	Certification Path Solicitation Message
149	Certification Path Advertisement Message
150	ICMP messages utilized by experimental mobility protocols
151	Multicast Router Advertisement
152	Multicast Router Solicitation
153	Multicast Router Termination
154	FMIPv6 Messages
155	RPL Control Message
156	ILNPv6 Locator Update Message
157-199	Unassigned
200	Private experimentation
201	Private experimentation
255	Reserved for expansion of ICMPv6 informational messages

Table 2.2 ICMPv6 Message Types from IANA

ICMPv6 messages are also used by NDP to help facilitate routing. The next section will discuss NDP and routing in more detail.

2.1.5 NDP

NDP is the protocol used by IPv6 to help nodes discover each other on a network, establish routes and receive subnet information from the network [2]. It provides the functionality of the Address Resolution Protocol (ARP) which is used by IPv4 to find nodes on a network and accomplishes this through the use of a set of ICMPv6 messages. The specific ICMPv6 message types used for NDP are 133-137 and are the Router Solicitation, Router Advertisement, Neighbor Solicitation, Neighbor Advertisement, and Redirect messages.

As a part of IPv6 addressing, NDP is used to provide subnet information from the network while establishing the route for the node in the network. The sequence for this process is as follows: A request for the subnet information is sent to the router using router solicitation messages when a node joins a network. The router will respond with an advertisement message which contains the subnet prefix information. The node will then complete its 128-bit IPv6 address and will be reachable by the network. A node will send a neighbor solicitation message for the destination node when it wants to communicate. The receiving node will reply with a neighbor advertisement message indicating it is reachable. The intended communication will then follow. The interaction between a node joining the network and router using NDP is illustrated in Figure 2.6.

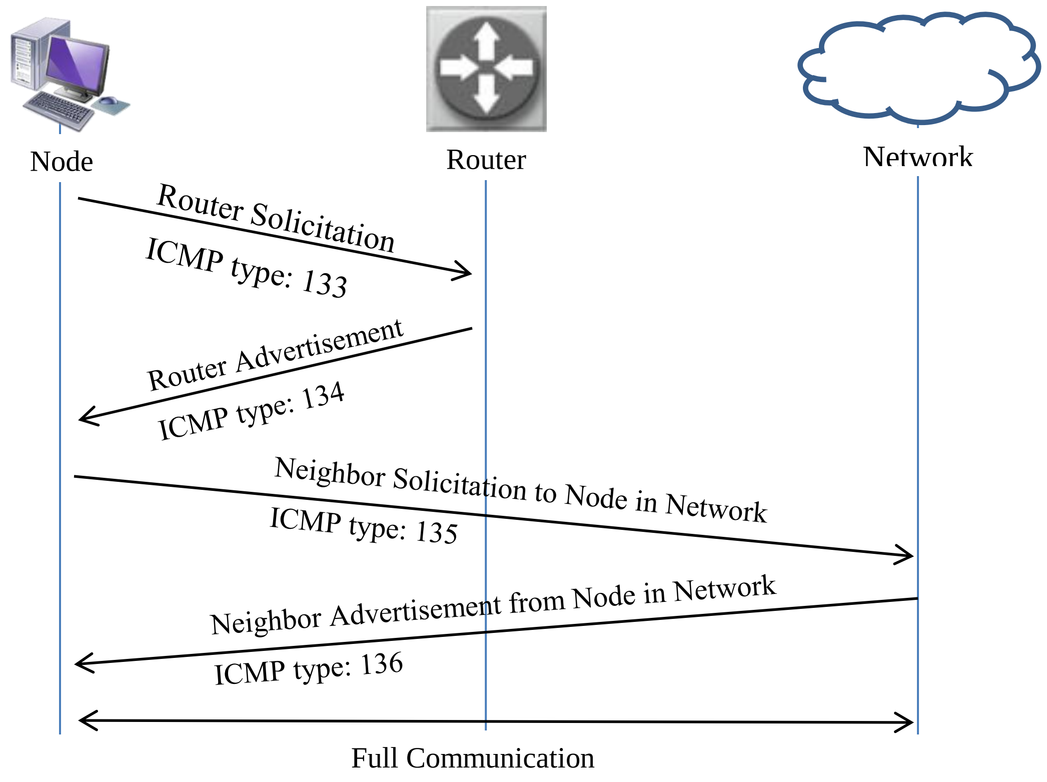


Figure 2.6 NDP Interaction

NDP improves upon IPv4’s route establishment because the protocol contains all the messages needed for this process whereas IPv4 has to use a combination of ARP and ICMP to accomplish network discovery. ARP is used to resolve IP addresses to their corresponding MAC addresses and ICMP is then used to discover routers. NDP also enables auto-addressing as previous discussed. More ways that ICMPv6 improves upon IPv4’s mechanisms can be found in its RFC [2].

2.2 Voice over IP

Voice over IP (VoIP) refers to voice and media communication services provided over the Internet protocol. There is no formal definition for VoIP by the U.S. Federal Communications Commission [31]. VoIP is used to support telephone calls over the internet versus the traditional analog telephone network. Companies such as Skype and magicJack provide VoIP services to home users and have seen growth in usage [5] [6]. In 2011, Skype usage grew 48% and the service accounted for twice the amount of international calls than traditional phone companies [32]. This shows that VoIP is a common application among household users.

VoIP services can be accomplished using a variety of different protocols over IP. Some protocols are open such as Real-time Transport Protocol (RTP). Other protocols are proprietary such as the Skype protocol which is based on peer-to-peer concepts [5]. Home users and companies should compare the options available when considering which standard to use. One way to compare the protocols is to consider the quality of service. VoIP's objective quality of service can be measured with a set of metrics including latency, packet loss and jitter. The Mean Opinion Score (MOS) and the R-Factor are specific metrics for VoIP that involve subjective ratings [7]' therefore, a combination of metrics should be used to obtain a complete measurement of VoIP's quality of service for the comparison between protocols.

The components of VoIP are the codecs, the signaling process, and media servers, gateways, and clients. Each of these plays a part in the successful establishment of a VoIP application. A codec is an encoding/decoding program that coverts analog audio into digital and vice versa. VoIP applications have a plethora of codecs to choose from to use, all with varying compressions and quality. Designers of VoIP applications choose codecs based on their determined requirements for quality and latency; therefore, each application is different. Media servers process data streams and handle connections between clients. Media gateways are devices that connect different types of networks together, like an IP network with a Public Switched Telephone Network (PSTN), so that clients on disparate networks can communication. Media clients are the nodes that are actually generating and receiving calls.

The signaling process establishes the setup and tear down required for each VoIP session. One of the standard signaling processes is the Session Initiation Protocol (SIP) [58]. SIP is an application layer protocol that uses a peer to peer connection for initiating a session. The components needed for SIP are the proxy server which listens for calls and facilitates the connection and clients which initiates the calls. Clients need to use identities specific to SIP to call each other; these identities are analogous to telephone numbers or email addresses and are bound to an IP address. SIP uses a similar request and response format to HTTP [58]. Initiation of a call starts with an "INVITE" message, containing source and destination addresses and the type of session, which is routed to the destination through the proxy servers. The servers must report back to the client that they has received the "INVITE" message and are finding the route to the destination by sending a "Trying" message. Once the "INVITE" message is received by the destination client, two messages can be sent: a "Ringing" message and an "OK" message. A "Ringing" message is always sent back to the call originator indicating that the client has been reached but has not answered the call request. An "OK" message is sent when the destination client has answered the call request; this completes establishing the connection. Apart from the "INVITE," "Trying," "Ringing," and "OK" messages, SIP has several other messages that are used to indicate a variety of responses, including errors; the full list can be found in the RFC [58]. The SIP session establishment is displayed in Figure 2.7.

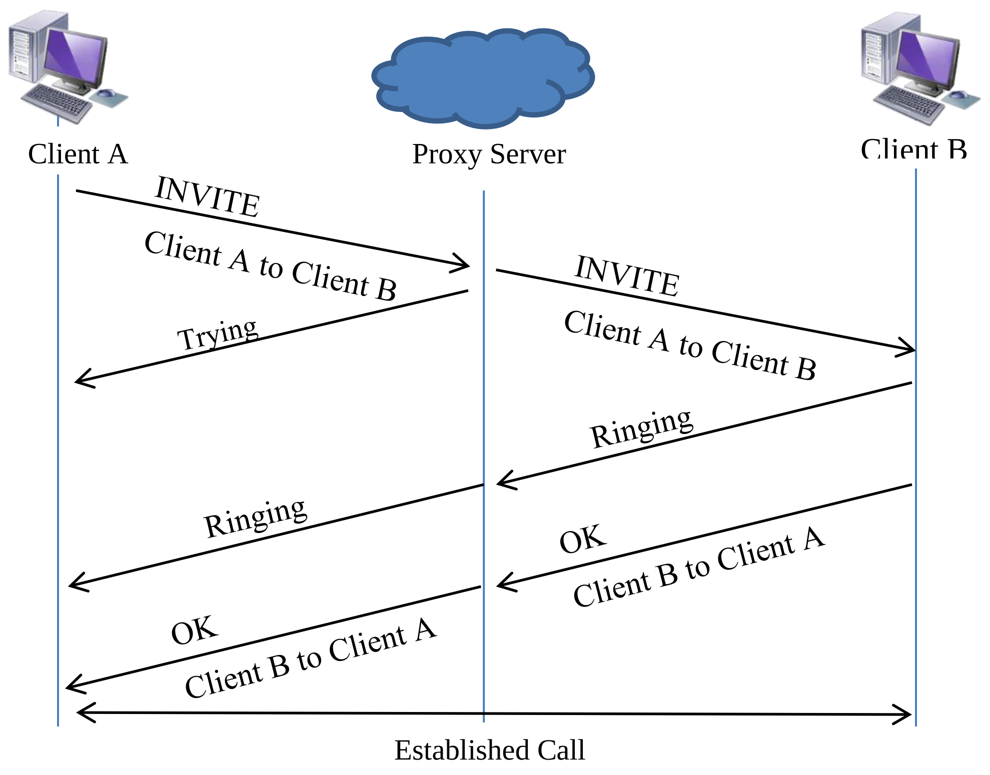


Figure 2.7 SIP Establishment Process

2.3 Moving Target IPv6 Defense

Research completed in the Virginia Tech Information Technology Security Lab (ITSL) revealed that there are several privacy issues pertaining to IPv6 addressing. One particular privacy issue found with SLAAC addressing is that network users can be tracked across subnets by potentially malicious people scanning for their Modified EUI-64 values. In response to these issues and vulnerabilities, a specialized application called a Moving Target IPv6 Defense (MT6D) was created by Dunlop [8]. MT6D was tested on Virginia Tech's IPv6 production network by Dunlop and in simulation as a part of this work.

MT6D is a scheme in which the vast IPv6 address space is taken advantage of by having users constantly rotate addresses, preventing an attacker from being able to find and correlate packets between two users. Users of MT6D keep track of their correspondent by calculating each other's next address rotation. Addresses are calculated with a shared key, the EUI-64 and a timestamp. The shared key and MAC address of the users must be exchanged before enabling MT6D. The exchange of this information is left to the discretion of the users. It is also essential that the two users have synchronized clocks to maintain the same timestamp. MT6D accomplishes clock synchronization by periodically querying a common time server between the devices.

To use MT6D, the application can be installed directly onto a physical machine or onto a separate device that sits between the machine and the network. MT6D was tested in simulation using a physical machine that had the application directly installed on it. Once installed, MT6D has four configuration files. Three configuration files maintain mappings of MAC addresses to human readable device names, human readable device names to IP addresses, and human readable device names to shared keys. The fourth configuration file maintains all MT6D options including address rotation time (adjustable to seconds), AES encryption, and dynamic subnet obscuration. The MT6D options configuration file must be the same between two users otherwise communication will fail. After configuring these files, MT6D can be enabled.

MT6D's execution process starts by making a virtualized network interface. This interface maintains the device's original IP address and generates all outgoing traffic. All packets from the virtualized interface are captured by the MT6D application and wrapped in the MT6D header. The MT6D header is an IPv6 header that uses the rotating addresses as the source and destination. If encryption is enabled, a Destination Options extension header is generated containing information about the encryption type. Furthermore, MT6D uses UDP as the transport layer protocol and the original packet is contained in the UDP header's payload. When a MT6D packet is received, the application strips the MT6D header off the packet, decrypts the original packet if necessary and forwards it to the virtualized network interface. The machine then responds to the packet normally.

Overall the MT6D application provides anonymity to users and can provide encryption for data. This scheme adds an extra layer of security to user's existing security infrastructure contributing to the defense-in-depth concept. More information on MT6D can be found in Dunlop's work [8].
.

2.4 Chapter Conclusion

An understanding of IPv6, VoIP, and MT6D is vital to comprehend the focus of testing in this thesis. The chapter described key attributes of IPv6 particularly addressing, extension headers, and network discovery. VoIP's main components were explained with a focus on establishing a session using SIP. The MT6D application process of providing privacy to users was expounded upon, including the configuration options.

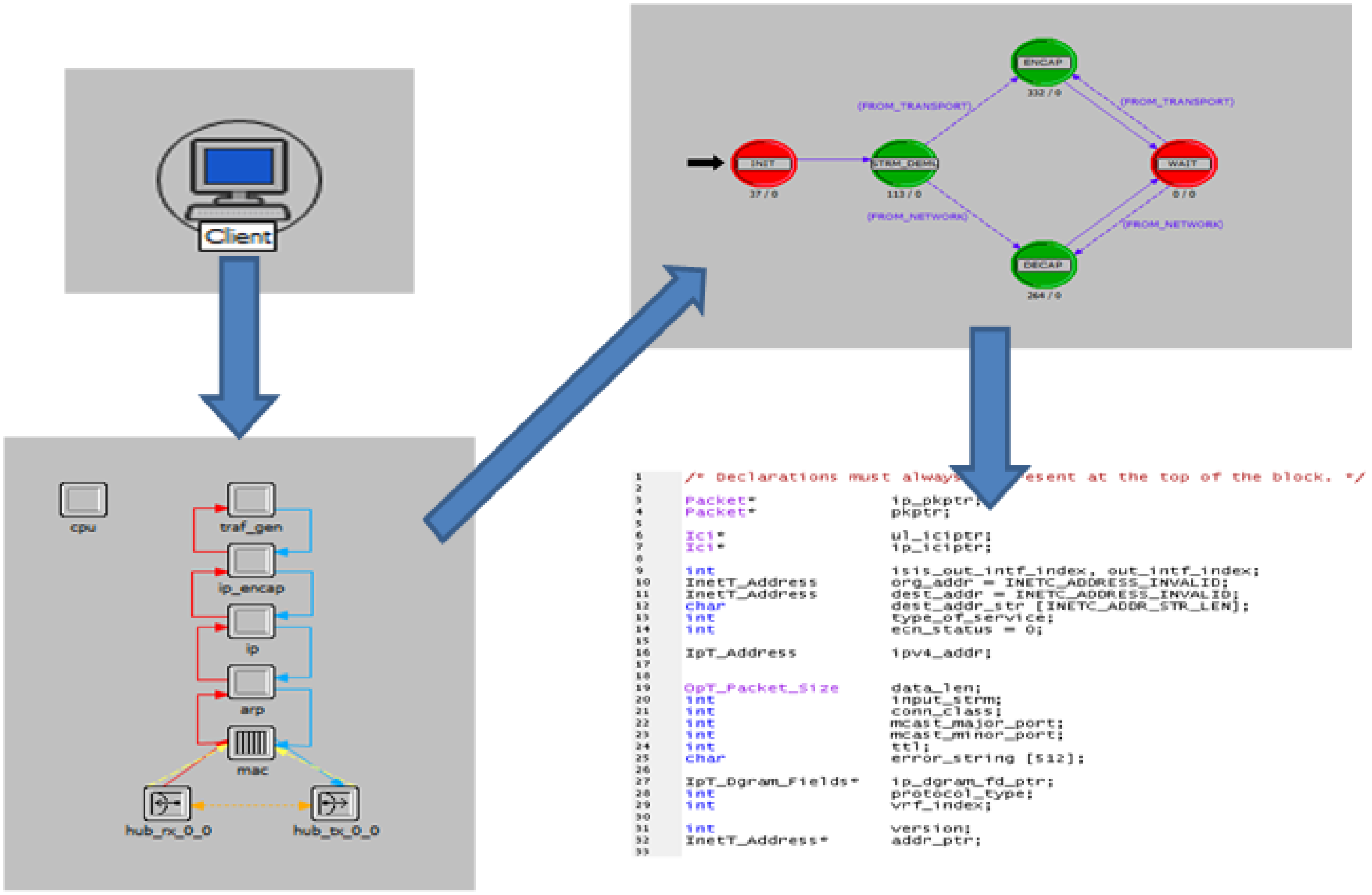
3. OPNET Modeler and the System-in-the-Loop Module

OPNET is a company that provides software solutions targeted at network management [3]. Modeler is their network simulator which is used in network research and development. Other products focusing on network monitoring, auditing and planning are available for purchase. This chapter describes OPNET Modeler and the System-in-the-Loop Module.

3.1 OPNET Modeler Overview

OPNET Modeler is a discrete event simulator that allows users to build and test network configurations and protocols. It is used by a variety of companies such as the Harris Corporation, Intel and Ericsson as well as the U.S. military [33]. The simulator is advertised to support a broad range of protocols and statistics gathering methods. It also provides a large set of simulated network devices and links, such as routers and switches, including devices from proprietary sources (Cisco, Juniper, etc.).

Modeler uses a graphical user interface to drag and drop a pre-defined library of nodes into a network topology. The user can then define attributes for each node using menu options. Each simulated node can be broken down into a set of connected processes which in turn consist of state diagrams. The states of each diagram are created using C/C++ code that use the OPNET coding libraries. This design allows for the user to have control over the node at all levels of abstraction as well as being able to define specialized nodes. Each level of abstraction for a node within the Modeler interface is shown in Figure 3.1. The top left corner displays a node in the main interface. The bottom left corner shows the next level of abstraction which is the process diagram of the node. The top right corner is the state diagram for a single process and the bottom right corner displays the code used to create the states. Each level can be modified by the user resulting in a custom model by adding or changing processes, states, and code. Link models are customizable as well if a user desires a different set of parameters for a link.



```
/* Declarations must always [illegible]esent at the top of the block. */

Packet*                 ip_pkptr;
Packet*                 pkptr;

Ici*                    ul_iciptr;
Ici*                    ip_iciptr;

int                     isis_out_intf_index, out_intf_index;
InetT_Address           org_addr = INETC_ADDRESS_INVALID;
InetT_Address           dest_addr = INETC_ADDRESS_INVALID;
char                    dest_addr_str [INETC_ADDR_STR_LEN];
int                     type_of_service;
int                     ecn_status = 0;

IpT_Address             ipv4_addr;


OpT_Packet_Size         data_len;
int                     input_strm;
int                     conn_class;
int                     mcast_major_port;
int                     mcast_minor_port;
int                     ttl;
char                    error_string [512];

IpT_Dgram_Fields*       ip_dgram_fd_ptr;
int                     protocol_type;
int                     vrf_index;

int                     version;
InetT_Address*          addr_ptr;

```

Figure 3.1 OPNET Modeler’s Levels of Abstraction

Modeler has a wide set of statistics available including capturing the amount of packets of a specific protocol or application received and sent by an interface, latency, link load, etc. Statistics can be displayed as is, as an average, or by a defined distribution. The user can specify whether to capture statistics on a specific node or on the network as a whole. Statistics, like node models, are customizable by the user by modifying the capture rate and method.

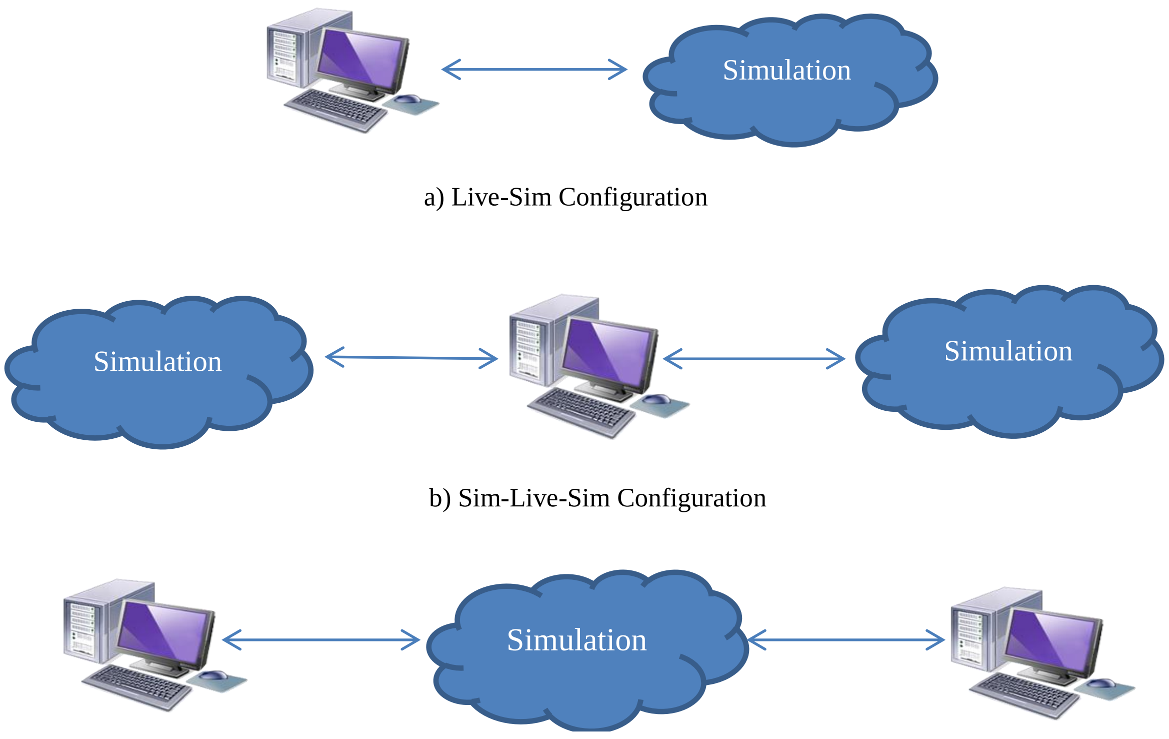
Modeler's features are extended through add-on modules, like IPv6 and wireless support, available from OPNET [1]. The System-in-the-Loop module (SITL) provides the capability to translate real network data into simulation [4]. This allows premade hardware prototypes and software to interact with a simulated network without having to create a simulation model of these prototypes. SITL is advertised to support a set of basic protocols which includes IPv6. This module is described in the following section.

3.2 The System-in-the-Loop Module

SITL is an add-on module to OPNET Modeler that allows real traffic to be sent to and from the simulation environment. This is a great benefit for applications and tools that are in hardware or are in the prototype stage. It can also be used when applications are complex and cannot be simplified into a simulated node.

SITL operates by working as a translator between the live network and the simulated network. The translation module is configured to pull and send packets with a specific network interface. A packet filter must be written to specify which packets should be translated into simulation from the live traffic. SITL uses the Berkeley Packet Format as the syntax for these packet filters. It is suggested by OPNET that one network interface per SITL process be used to make the filtering process easier to handle; however, it is not required. One network interface can be used for several SITL processes within the simulated network topology provided that a proper packet filter is written for each process. For this research, OPNET's suggestion was taken into consideration. It was determined that multiple network interface cards (NICs) would help to simplify the simulation process and reduce the configuration time. Two D-Link USB 2.0 Ethernet adapters were purchased to be used solely for the simulations.

SITL has three configurations to set up communication between live software or hardware and the simulated network. The configurations are named Live-Sim, Sim-Live-Sim and Live-Sim-Live and are illustrated in Figure 3.2. The names are an explanation of the networks through which the packets travel. Live-Sim is the configuration in which packets are sent from a live network and have a final destination to a simulated node. Sim-Live-Sim is the configuration in which packets are generated in simulation, sent through a live network and have a final destination back in simulation. An example of this is to have two simulated workstations that are connected through a live switch. Live-Sim-Live is the configuration in which packets are generated in a physical environment, sent through a simulated network and have a final destination of a real network/device. Live-Sim and Live-Sim-Live are the two main configurations used in this work.



a) Live-Sim Configuration

b) Sim-Live-Sim Configuration

c) Live-Sim-Live Configuration

Figure 3.2 The Three Configurations of SITL

The primary process in the SITL node is the translation function. The translation function is a set of three methods that change real packets into simulated ones and vice versa. The three methods are a testing method, a translation method from simulation to real, and a translation method from real to simulation. The testing method looks for characteristics in incoming and outgoing packets that determine what type of protocol is being used. It will return true or false depending on if the translation function is able to translate the detected protocol. For example, a translation function exists for TCP. The testing method looks for the protocol number within the IP packet header. If it detects the correct number, the method returns true and the TCP translation function is used. Testing methods are defined with a priority number and are executed in order by the SITL process. The first testing method to return true executes its corresponding translation function. Incoming and outgoing translation functions (in relation to the simulation) step through each bit of the header, using that information to create a packet, either into simulated packet form or as a real packet stream.

OPNET has pre-defined translation functions for Ethernet, IPv4, IPv6, ICMPv4, ICMPv6, ARP, TCP, UDP, OSPF, RIPv2 and WLAN [34]. Users can create their own translation function for other protocols as well. The customized functions must define three methods — testing, real to simulation, and simulation to real — and a method that registers the function along with its testing priority with the SITL process. One condition on this customization is that the protocol being translated must have a defined simulated packet form in Modeler. If one does not exist, then the user must first define the packet format. This can be done within the Modeler GUI.

3.3 The Advantages and Disadvantages of OPNET Modeler

There are advantages and disadvantages with using OPNET Modeler. The clear disadvantage is that OPNET is a proprietary simulator and costs money to use. A license must be purchased to run the Modeler software and each add-on module (IPv6, SITL) requires an additional license. OPNET has an educational program that gives out free licenses for some of their products to universities that qualify [35]. Paying for Modeler provides access to technical support and user forums which are otherwise not available to the general public. The forums are not particularly active; new posts occur every 1-2 weeks averaging 0-5 replies [36]. The lack of activity on the forums can be seen as a further disadvantage.

There are many advantages to using OPNET despite the cost. The first advantage is that proprietary network models from companies like Cisco are available for building network topologies. Other network simulators do not have these models available by default; users of other simulators have to create their own network models. The availability of these proprietary models helps users to create network topologies that closely match existing networks. For example, a system administrator could build a simulation using Cisco devices that exist in his/her live network. A second advantage is that the OPNET software is generally reliable. Open source projects can contain software bugs that may not be fixed in a timely manner. Since OPNET is proprietary, customers expect that patches will be made quickly to fix any known error. A third advantage is that Modeler does advertise to support a great amount of protocols. For example, in comparison to ns-3, Modeler has more routing protocols implemented [34] [37]. A fourth advantage is that OPNET provides training classes for their products for new users. This helps decrease the time to learn the details of the software and decrease the time to create effective simulations. Finally, a fifth advantage is that OPNET conducts its own conference that showcases new features and updates to products, customer research and development, and further educational outreach.

3.4 The Use of Modeler in this Research

OPNET Modeler and SITL were used to create a series of network topologies as a framework to assess IPv6 applications. A full evaluation of the software for IPv6 support was done through a set of tests. Chapters 5 and 6 will describe the network configurations, tests and results of this evaluation. The developed framework described in Chapter 5 was used to test a VoIP application and MT6D. The IPv6 application test methodology and results can be found in Chapter 7.

3.5 Chapter Conclusion

This chapter described OPNET Modeler and the SITL module. The main interface and components of simulation were covered and the configurations of SITL were detailed with examples of how they are used. The advantages and disadvantages of using Modeler were discussed with the advantages outweighing the main disadvantage of cost.

4. Simulation Tools and Related Research

Simulation tools are readily available for network and application testing and are used abundantly in academic research. Several major concepts from previous work on simulation frameworks, including the choice of simulation tool, were used as building blocks in creating the IPv6 simulation framework. Since OPNET Modeler was chosen as the simulator for the framework, a literature review of work done using Modeler and the SITL module in IPv6 was done. Section 4.1 gives the overview of simulation tools examined for the framework and Section 4.2 presents the literature review on simulation frameworks and OPNET.

4.1 Simulation and Emulation Tools

There are several tools available to create network simulations. Four tools researched before deciding to use OPNET Modeler as the tool to create the IPv6 framework. Tools were looked at with two criteria. First, it had to support IPv6. Secondly, it had to have the ability to tie in live network traffic. This second criterion was based on the fact that MT6D had an existing developed prototype. Any additional features counted positively towards the tool. Four simulation/emulation tools were considered based on the criteria: ns-3, OMNeT++, Common Open Research Emulator (CORE) and OPNET [18][19][20][3].

ns-3 is a discrete event simulator that is used widely in educational research. It is primarily used to simulate IP/wireless networks [18]. C++ or Python code is accepted to describe models to develop a simulation. The command line and code is the main avenue for running and building a simulation. It does not visualize any of the networks and does not have an integrated debugger. ns-3 does have a system in the loop capability that works as a scheduler between real network traffic and virtualized traffic which met one of the criteria. ns-3 version 3.10 was the stable release at the time of evaluating simulators for this work (January 2011). This version of ns-3 has several IPv6 classes, but lacks support in some areas. For instance, it does not support global routing for IPv6. Global routing was added by the developers to a later release during the summer of 2011 [38]. Another lacking area of the 3.10 release is that TCP and UDP are not supported with IPv6. Support for these transport layer protocols has been added in the 3.14 release of June 2012 [39]. Since ns-3 was found to lack in some areas of IPv6 support, it was not chosen for this framework.

OMNeT++ is a C++ based simulation library and framework to help build network simulators [19]. It is an open source project that is maintained and contributed to by the academic community. There are several pre-built tools; one such tool is the xMIPv6 project which supports Mobile IPv6. The project has subsequently been moved into the main INET framework. The framework documentation describes IPv6 and ICMPv6 support, but does not provide the same detail as ns-3's documentation [40]. OMNeT++ did not have a mechanism for live network communication at the time of comparison, therefore, OMNeT++ was not chosen as the simulation tool as it did not meet the second criterion. A secondary project, TTE4INET: the Time Triggered Ethernet Model for the INET Framework, was developed and released in February 2012 [21]. This project allows for real time communication with simulation which helps OMNeT++ meet the criteria; however, it was released after a majority of this research had been done.

CORE is an emulation project that was developed by the Boeing Corporation for the United States Navy [20]. The benefit of this tool is that the emulations can be run on multiple machines and can be connected to live networks. The network devices are emulated which makes it easy to customize devices to run and support various protocols including IPv6 and custom applications such as MT6D. One of the drawbacks of emulation is that these devices have to be individually configured as virtual machines (VMs). A machine needs to have sufficient resources to handle the additional RAM and hard disk space needed for VMs. For example, if there are 10 VMs with 1GB of RAM to be emulated, the emulation machine needs to have at least 10GB of RAM free. The additional time and resources needed to configure and host the emulations was not available; therefore, CORE was not chosen as the emulation tool.

The last tool evaluated was OPNET Modeler [3]. As discussed in the previous chapter, OPNET fits the two criteria of supporting real time traffic communication with simulation through SITL and advertised IPv6 support. The advantages of providing proprietary network models, training classes and meeting the two criteria made OPNET the choice for this framework.

4.2 Related Research on Simulation Frameworks

Simulating a network environment is not a new concept. Many researchers have looked to simulation as a viable way to test new network configurations and new network device models. There are several examples of simulation frameworks developed for live networks, new protocols and wireless configurations. The following frameworks helped develop this research by verifying areas that have been already tested and by revealing topics that have not been examined.

Two studies were found that created campus network frameworks using OPNET Modeler. One study was run by SUNY Fredonia [9] and the other was done by K.U. Leuven in Belgium [10]. The purpose of the SUNY Fredonia framework was to study the performance of the new Gigabit Ethernet backbone and of voice traffic over the network. They built their framework with a combination of Cisco and custom models and ran six scenarios. The study from K.U. Leuven created a simulation model of their campus network which included a five switch ring as its core topology. As with the SUNY Fredonia campus framework, voice traffic was used as a case study. Through these two studies, it was determined that voice traffic is a standard application to simulate over networks when looking at performance.

Various test bed studies and publications on IPv6 have been produced. One particular piece of literature gave an overview of IPv6 test beds in China circa 2004 [11]. China has several live research networks run by educational institutions and the government, four of which were described in this paper as being a part of countrywide IPv6 study. Another study was found on an IPv6 wireless sensor network in China [12]. The sensor network test bed was likewise a live network and it contained a series of 20 sensor nodes, a sink (gateway device), and an IPv6 server/database. The results of Huo et al. represented the hardware limitations of developing a physical test bed versus one in simulation. A third publication was found that expounded upon work on test beds in France which were used for a number of projects including testing IPv6 mobility, voice traffic and web browsing [13]. These test beds give an interesting perspective on live network testing versus pure simulation. The study from Montavont et al. contributed to the use of VoIP as a standard testing application for IPv6 networks for the framework.

Certain aspects of IPv6 in OPNET Modeler have been researched. An evaluation of IPv6 Simulation in OPNET was done in 2003 by Chang et al. which looked at the internal simulation IPv6 model [14]. It focused on the IPv6 ping model, IPv6 tunnels and routing protocols in particular. The example networks in this paper are very similar to the OPNET IPv6 tutorial and it was reasoned that this evaluation helped create the OPNET tutorial. The study did not adequately cover IPv6 traffic generation in simulation and so it was determined that the model should be re-evaluated. Another evaluation was found concerning Mobile IPv6 in OPNET [15]. It examined the Mobile IPv6 model's configuration and then did several tests with application traffic. They found that the implementation of the handover procedure of Mobile IPv6 had adverse effects to the congestion control of the transport layer which helped to indicate that there could be some other unseen problems within the OPNET IPv6 implementation. The most recent study examining IPv6 in OPNET was published in December 2011 and examined voice traffic performance in dual stack networks [16]. A topology was developed and used to study voice traffic in both IPv4 and IPv6. The study was done completely in simulation and used OPNET's ability to create a custom application to set the voice traffic.

A study that created a Virtual/Live Gateway process for IPv6 using OPNET provided information on the process of characterizing applications [17]. The purpose of the Virtual/Live Gateway was to serve as a communication mechanism between simulation and external data. It was concluded by the authors that the Virtual/Live Gateway process was suited for single packet, non-real time traffic. The authors suggested further research into developing the gateway to handle a wider range of application traffic and simulation environments. They also proposed further work into having the gateway generate real packets. No further publications on the proposed further work could be found.

4.3 Chapter Conclusion

This chapter described three simulation tools and one emulation tool that were considered for this work. It described the criteria used for consideration and how each tool met or did not meet the criteria. It also covered related research of simulation frameworks. The results of the literature review revealed that OPNET was the best suited simulator for this research but had not undergone a full IPv6 traffic evaluation and VoIP is generally used as a standard application in test beds.

5. Configuration and Network Topology Design

The choice of the configuration and topology for a simulation is dependent on the statistics being measured and the desired nodes in the network. The simulation measuring the effects of traffic on a router would be configured and designed differently than the simulation monitoring the effects of mobility on a router. In two major section of this chapter, the configuration options and topology choices made to make the IPv6 application simulation framework are explained. The first section describes the configuration between the simulator and the live application machines. The second major section describes the network topology for each scenario in the framework.

5.1 Live Network Configuration

There are two ways to set up the live network to interact with the simulated environment. The first way is through the use of Virtual Machines (VMs). These VMs can be hosted on the same machine as the simulation software or separately on a remote server. The second way is to use real machines that physically connect to the simulation network. This section describes the set up for each configuration and discusses their benefits and limitations.

5.1.1 Virtual Machine Configuration

VMs are a good alternative to use when there are limited physical machines available because they virtualize the components of a real machine, like the processor, buses, and peripherals, and a full operating system. Software testers are attracted to using VMs since they can easily test software on a variety of systems without having to configure and run a full physical machine. There are two main companies that provide free virtualization: VMWare and VirtualBox [41] [42]. VMWare is a leading virtualization company that provides services across all levels, from the home user to large businesses. . VMWare Player is their free virtualization solution for home users. VirtualBox is an open source virtualization solution from Oracle that provides the same services as VMWare player. It has more flexibility and configuration options than VMWare Player. VMWare has solutions that have the same flexibility and configuration control, but these solutions are not free.

VMs were used to do the IPv6 evaluation of OPNET Modeler and SITL. VMWare Player was chosen as the virtualization agent as it provided the most stable network configuration through initial trials. The VirtualBox solution was tried, but full communication with the machines and simulation was never achieved. The network bridging mechanism of VirtualBox was determined as the problem area as other means of networking (Network Address Translation and a host-only network) worked correctly.

There were three VMs made to be used for the IPv6 OPNET evaluation. Two VMs used the Ubuntu 11.10 operating system (OS) which supports IPv6 and can be easily configured to run a webserver. Wireshark [23], a packet capturing tool, was loaded onto these machines to help monitor packets across the network. The third machine ran Backtrack 5, a penetration testing Linux distribution [43]. Backtrack 5 contains several packet crafting tools that were used for the IPv6 extension header tests. Each VM has its virtualized network interface bridged with one of the USB NICs. The configuration between VMs and the simulation on the host machine is shown in Figure 5.1.

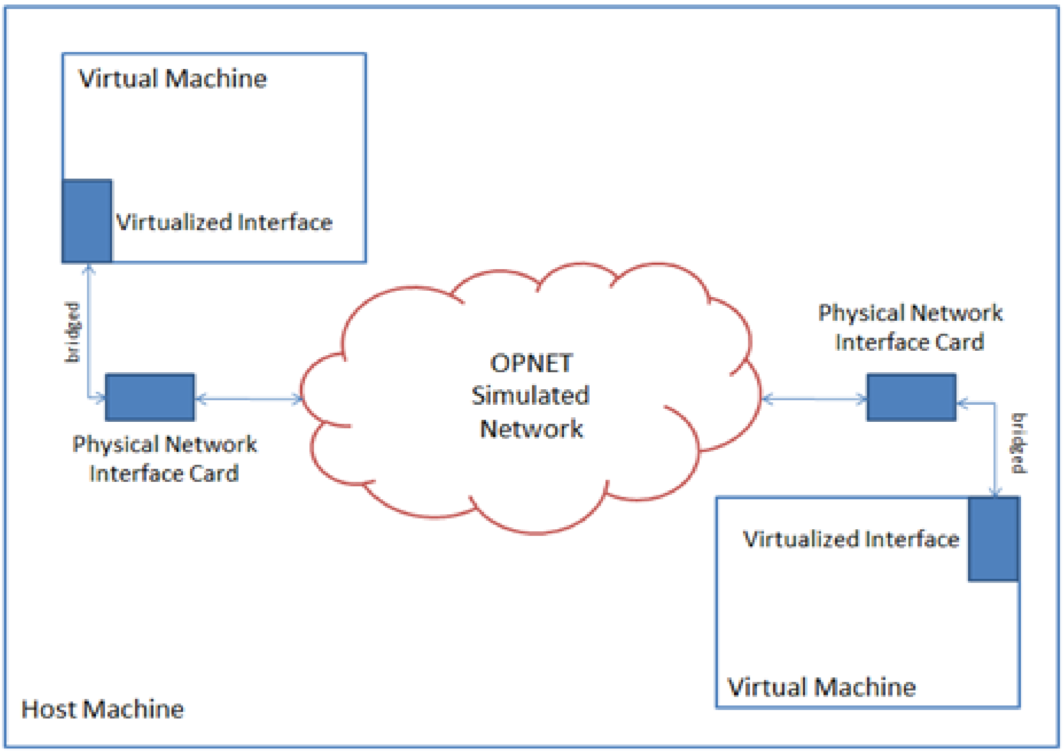
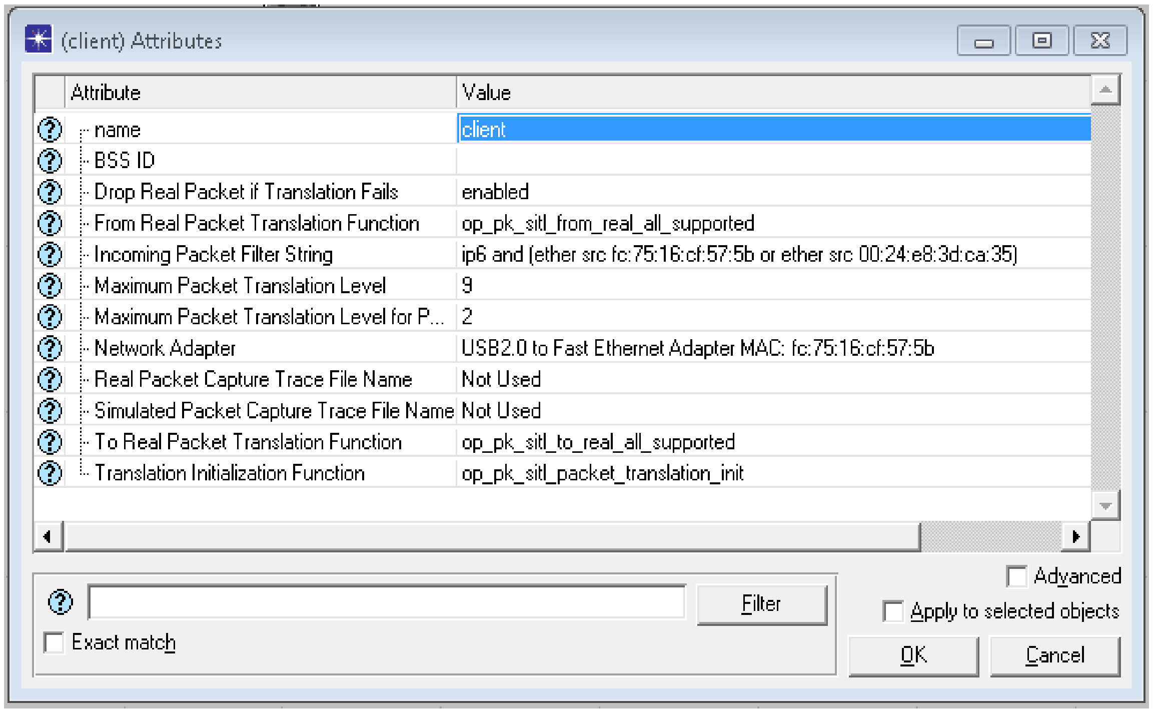


Figure 5.1 Configuration of VMs on a Host Machine

The SITL process was configured with the NIC bridged with the VM and packet filter. The packet filter followed the same syntax for each SITL node:

ip6 and (ether src <Virtualized NIC MAC address> or ether src <USB NIC MAC address>).

There are three main parts to this filter the first of which is “ip6.” This part is intuitive; the filter allows IPv6 packets through to simulation while dropping IPv4 packets. The second part is “ether src <Virtualized NIC MAC address>” which filters for packets containing the Virtualized NIC Ethernet address as the source address. The third part is ether src <USB NIC MAC address>.” It only allows packets with the USB NIC Ethernet address as the source address though to simulation. These three parts completely form the filter that allows for the correct packets to be translated into simulation and any unrelated packets to be dropped. The SITL NIC and packet filter set in the attributes menu is shown in Figure 5.2.



Attribute	Value
name	client
BSS ID	
Drop Real Packet if Translation Fails	enabled
From Real Packet Translation Function	op_pk_sitl_from_real_all_supported
Incoming Packet Filter String	ip6 and (ether src fc:75:16:cf:57:5b or ether src 00:24:e8:3d:ca:35)
Maximum Packet Translation Level	9
Maximum Packet Translation Level for P...	2
Network Adapter	USB2.0 to Fast Ethernet Adapter MAC: fc:75:16:cf:57:5b
Real Packet Capture Trace File Name	Not Used
Simulated Packet Capture Trace File Name	Not Used
To Real Packet Translation Function	op_pk_sitl_to_real_all_supported
Translation Initialization Function	op_pk_sitl_packet_translation_init

Figure 5.2 SITL Attributes Menu

There are two main benefits to using VMs to host an application and generate packets. The first benefit is that extra machines are not need. The entire set of components for simulation can be contained on one physical machine providing that the host machine has free space for the VM files which allows more control over the system settings, particularly network interface configuration. The second benefit is that configuring of a new VM is fast and simple. VMs are easily duplicated so only one full configuration of a VM is needed if all the VMs have the same parameters. If the OS is a variable in the simulation, then new VMs with different OSs can be configured quickly without having to worry about compatibility with physical hardware.

The VM configuration has limitations as well. The biggest limitation of using VMs is that the host machine must have enough memory resources to contain the memory it uses and the memory of the VMs. Since VMs take up space on a host machine, the simulation performance can be affected. One possibility is that the simulation has a smaller space to grow which affects the performance. Simulations with a large amount of events will take longer to execute and have the potential to abort the program entirely. Another effect is that there is a measureable delay when transferring packets between the network interfaces. This extra time of passing packets between the virtualized network interface to the USB NIC adds to latency. The additional latency was made clear in a comparison ping flood test between using VMs and real machines. The VM configuration averaged 14.88ms for a 50,000 packet ping flood over 10 iterations. The real configuration averaged 2.04ms with the same test. The results of this comparison test are summarized in Figure 5.3. The observed additional latency was one of the determining factors for not choosing the VM configuration for the IPv6 application testing.

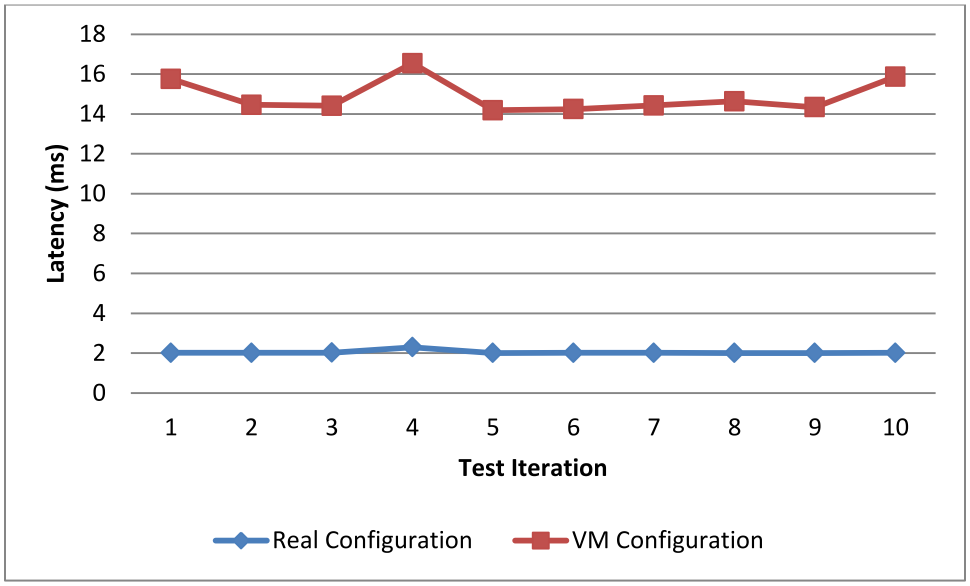


Figure 5.3 Comparison Ping Flood Test Results

5.1.2 Real Machine Configuration

The real machine configuration for hosting applications and generating traffic comes closer to mimicking a live environment. The network is the only item simulated in this configuration. This configuration was used for IPv6 application testing because it minimizes added effects from the configuration. The configuration is similar to the VM configuration with some slight differences, notably the connection between the devices and the simulated network.

The real machines used for this configuration were two Dell OptiPlex 960 machines loaded with the Ubuntu 11.10 OS. These machines were the same machines used in the live network testing of MT6D and were used to reduce the differences between tests, making the comparison of the results easier [8]. Each machine was loaded with packet capture software, the MT6D application and the open source VoIP application Mumble [27]. The SITL process was configured the same way as the VM configuration and the packet filter followed the same format of using the “ip6” filter and the two Ethernet source filters.

There were two options on how to connect the live machines and the simulation machine. The first option was to use a router or switch to connect all the machines. There are two drawbacks to using a router or switch. The first drawback is that the device has to be IPv6 capable. The second is that the device adds an additional hop to the testing network. The router connection option was not chosen because it was desired to keep the entire network in simulation. It was also not chosen because the additional hop was unable to be monitored and have statistics collected from it as can be done in simulation. The second connection option was to use crossover cables. Crossover cables are specialized Ethernet cables that allow a direct connection between two machines without an intermediate device. The crossover cable option was chosen as it did not add additional, unwanted complexity to the network.

The real machine configuration has its benefits and limitations. The primary benefit has been stated several times: this configuration mimics real network testing more closely than virtualizing applications. The simulation machine’s physical resources are used solely for the simulation software tolerating larger simulations. One drawback of using real machines is that there are more physical devices to configure and manage. The connection options between the live and simulation machines could be seen as a limitation as well if more than two physical devices need to be included in simulation. This is a limitation because the first connection adds an unwanted hop in the network and the second connection type is limited by the number of available USB ports on the simulation machine. The benefits outweigh the potential issues with connection and the real machine configuration was chosen for application testing.

5.2 Network Topology and Framework

The network topology is the basic building block of the network framework. Several different topologies were created and used as a part of different scenarios for testing the IPv6 applications. This section will explain each of the topologies, the individual components and settings, and the details of how the amount of simulated background traffic was determined.

5.2.1 Simulated Network Devices

There are three main components to the network: the end host device, the networking device, and the links between the devices. Each of these components can exist as proprietary models, generic models or customized models depending on the requirements of the network. Proprietary and generic models are provided by OPNET in complete working form. Customized models can be coded from scratch or be made from a template. Parameters such as available ports and protocols supported are defined by the user if a template is made. A combination of templates and coding can be used to create a custom model as well. For the network a topologies, a mix of proprietary and generic models were used. The specifics of these models are discussed in this section.

There are three types of end host models used to create the topologies in the framework. The first type of model is the generic simulated Ethernet workstation which can generate TCP and UDP application traffic and IPv6. It has one open Ethernet connection that can support a 10 Megabits per second (Mbps), 100 Mbps or 1000 Mbps link. There were other workstations available to use in the framework such as ones proprietary to Dell, but the generic version fit all the needs for the network. The second end host model is the Ethernet IP Station model and is a simplified version of the workstation device. It has a separate process that makes IP packets at a specified size and frequency instead of generating specific application traffic. It supports one Ethernet connection like the workstation. The third type of end host is the SITL Gateway. In simulation, it acts as an end host even though more than one node could exist beyond it. It supports a single Ethernet connection. The icons used in Modeler for each of these models are shown in Figure 5.4.

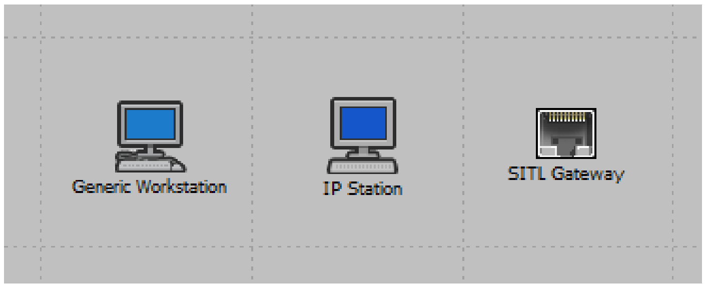


Figure 5.4 End Hosts

The networking devices make up the majority of models available in Modeler because of the wide range of proprietary models provided by OPNET. For instance, there are over 200 Cisco models for switches and routers and over 60 3Com devices, not to mention devices from Juniper, Alcatel, and more. The devices chosen for the framework scenarios were a mix of proprietary devices, specifically from Cisco, and generic models. If a specific device was known to exist in the Virginia Tech live network, the closest match to it in simulation was chosen. The four models used in the topologies were a generic router, a generic switch, a Cisco 7500 series router, and a Cisco 6500 series switch. The Cisco devices were chosen because they are used in the live campus network. The networking device icons used in Modeler are shown in Figure 5.5.

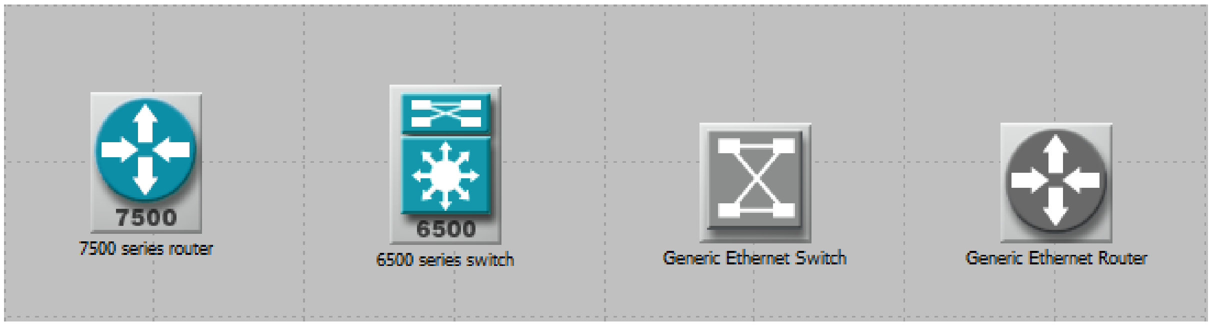


Figure 5.5 Networking Devices

There are 4 different links that were used in setting up the topologies. They are all Ethernet links with variable operation throughput, specifically, 10 Mbps, 100 Mbps, and 1000 Mbps. The nodes connected to a link have to support the specified throughput. The fourth link is a specialized

Ethernet link to connect the SITL module to another device and supports throughput at 10 Gigabits per second. All of these links are full-duplex links.

5.2.2 Topologies

The framework was designed to start with simple scenarios and build in complexity. There are a total of eight topologies that were made for this framework that fall into four main categories. This section will present each network, its purpose and which tests were run with it.

5.2.2.1 One Hop

The One Hop topology is the simplest network in the framework. It consists of a single Cisco 7500 series router connected to two SITL nodes with the specialized SITL link. The type of router can be switched for a generic router or one that matches more closely to another live network, providing the router supports IPv6. In Modeler, IPv6 has been enabled for most network nodes.

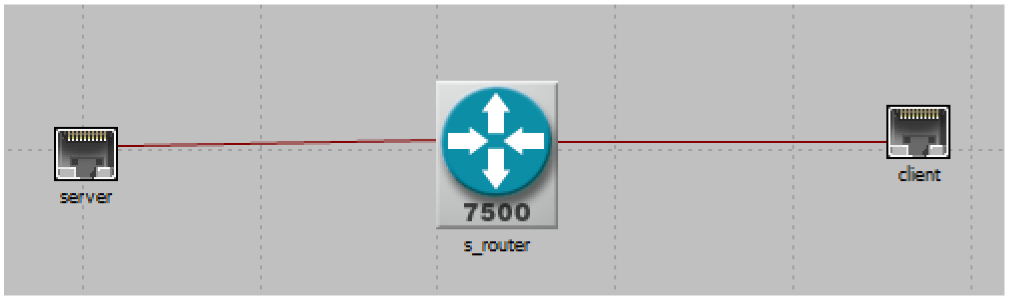


Figure 5.6 One Hop Topology

This topology, displayed in Figure 5.6, is used for all of the test scenarios including SITL IPv6 evaluation and the IPv6 application tests. Its purpose is to determine the effect routing has on a packet while excluding the effects of other traffic.

5.2.2.2 Two Hop

The Two Hop topology builds on the One Hop topology by adding an additional router in between the SITL nodes. This topology, displayed in Figure 5.7, is used in the IPv6 application tests. Its purpose is to understand the effect of an additional hop without the added effects of background traffic.

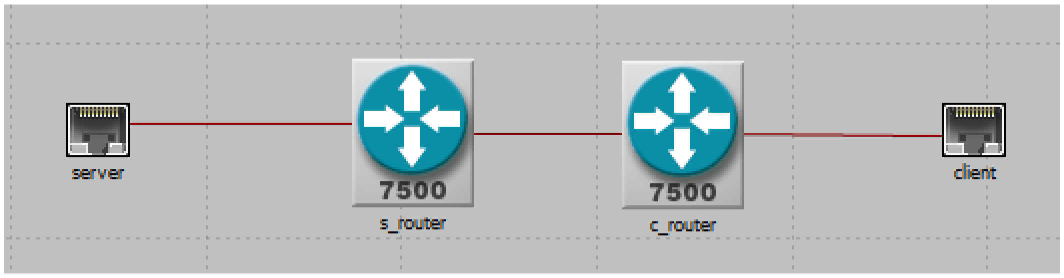


Figure 5.7 Two Hop Topology

5.2.2.3 Two Hop with Background Traffic

The Two Hop with Background Traffic topology contains the same core network of the Two Hop topology with the addition of simulated end host nodes that simulate background traffic. The addition of the simulated end hosts was included to mimic hosts on a live network. Application traffic is the type of simulated traffic used in this topology. Two modules named Application Configuration and Profile Configuration define Application traffic. More details on the specifics of traffic generation can be found in Chapter 6. This topology is shown in Figure 5.8.

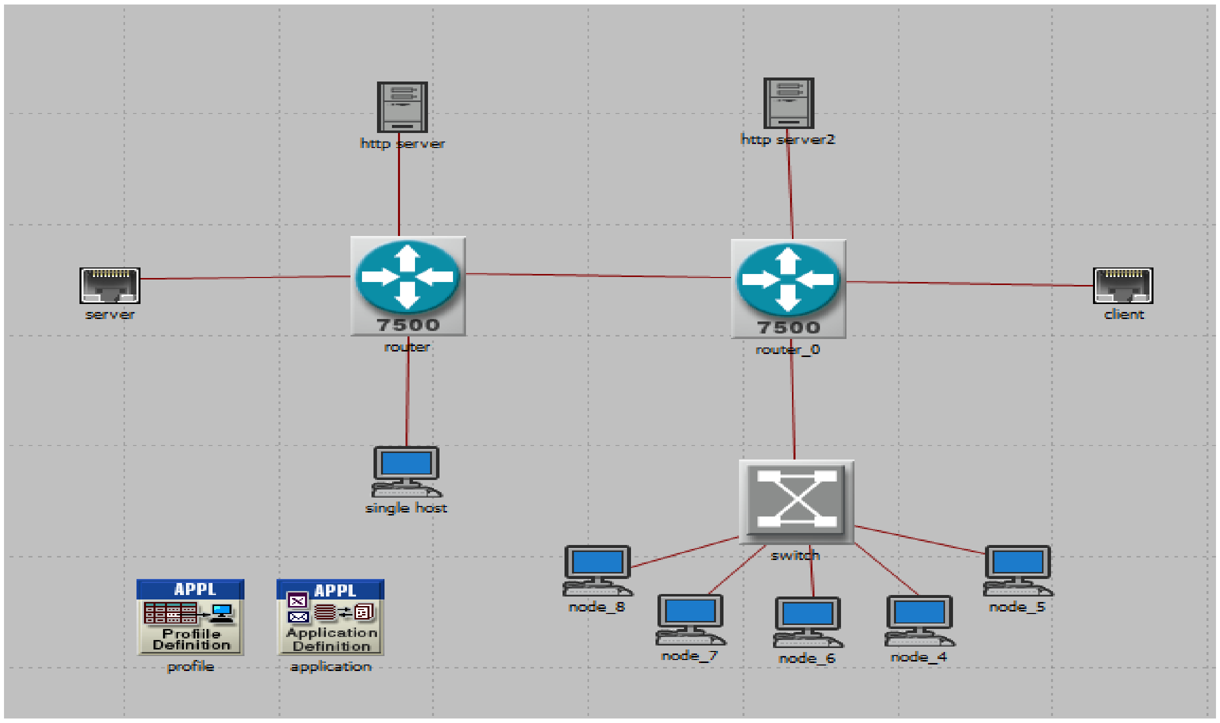


Figure 5.8 Two Hop with Background Traffic Topology

The Top Hop with Background Traffic topology was used in both the IPv6 evaluation of OPNET and in the IPv6 application testing to find the effects, if any, that unrelated traffic would have on the network and applications.

5.2.2.4 Campus Topologies

The Office topology, shown in Figure 5.9, is the base scenario of the campus related topologies. It contains a single switch with ten simulated users in addition to the SITL nodes. Application traffic has been deployed to each of the simulated users. The Floor topology is the next campus scenario following the Office layout. In the Floor topology, there are 120 total users connected to a single floor switch. The users are separated into sets of 40 connected to a single switch. The amount of users chosen for the Floor topology was the maximum that the switch model could manage. Each user does not have to have traffic defined and can act as though it does not exist for when the simulation runs making the real amount of simulated nodes user defined. The

Building and Section of Campus topologies are built from multiple Floor layouts. A generic building was defined to have 3 floors. The Section of Campus layout mimics a set of academic buildings on the Virginia Tech campus. Information about the number of floors on a building was obtained (if available) on the Virginia Tech website [44].

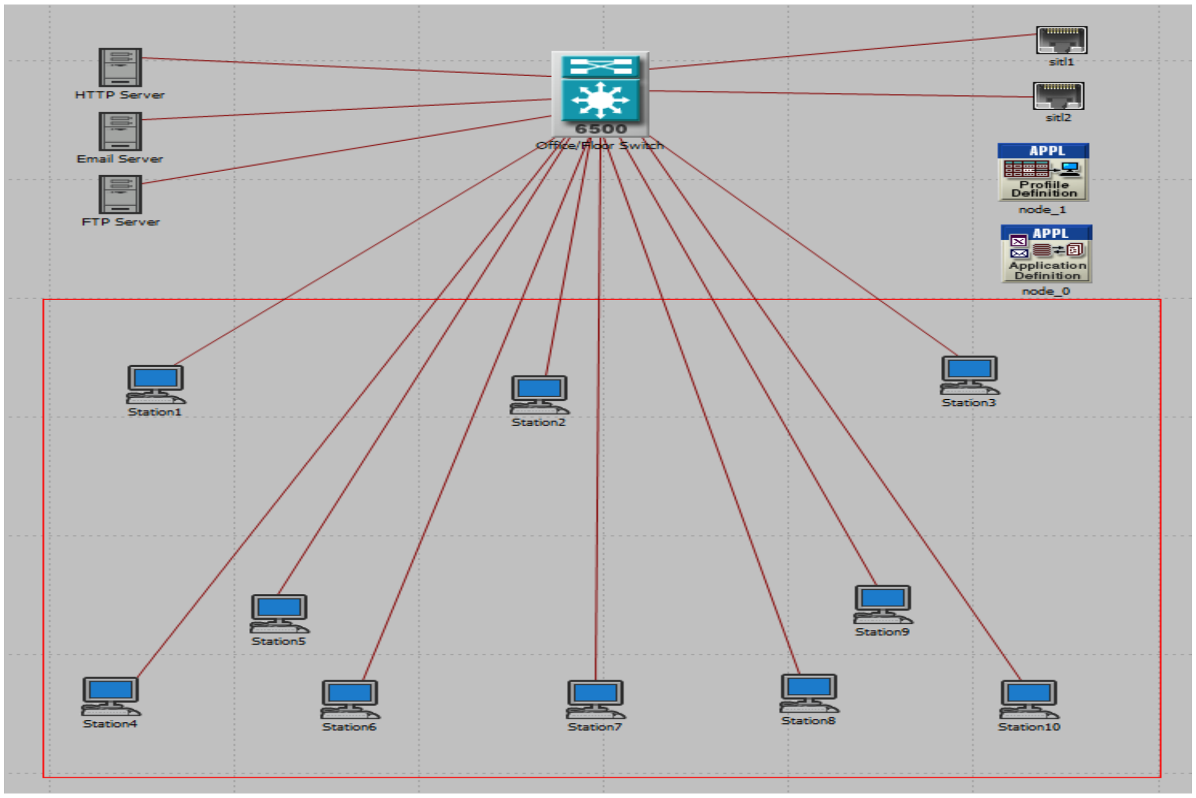


Figure 5.9 Office Topology

These layouts were used in IPv6 application testing to identify the effects of background traffic on the application and as well of the effect of an increased number of hops. There were some drawbacks to the larger configurations that will be discussed in Chapter 7.

5.2.3 Traffic Determination

Some of the designed topologies include traffic generating nodes. The campus framework was designed to be a close representation of the live campus network; therefore, the simulated background traffic was adjusted to resemble the campus network. The adjustment was done by examining the Multi Router Traffic Grapher (MRTG) website maintained by the Communication and Network Services Department of Virginia Tech [45]. This website shows link load information for each network router for incoming and outgoing traffic that is averaged over daily, weekly, monthly and yearly periods of time. The yearly average for each available link on the Burruss Hall Core Router was aggregated into a spreadsheet in which the average of the links was calculated. The overall average results of these calculations is shown in Table 5.1.

Burruss 6509-1	Incoming Traffic (kb/s)	Outgoing Traffic (kb/s)
Total Average	19133.05942	15374.26316
Named Buildings Link Average	5539.244	5616.068
Abbreviated/Other Named Link Average	36514.56	25743.835

Table 5.1 MRTG Average Traffic

The compiled information from the MRTG website was used to create an application traffic profile for each workstation so the overall load on the router link would match the average found by the calculations. HTTP traffic was chosen as the default application traffic as web traffic is common on campus.

5.3 Chapter Conclusion

The framework is the center point of testing IPv6 application. The configuration choices made in creating the simulation framework were described as well as the two different options of configuring applications with simulation through SITL. Each topology that makes up the framework was displayed and noted in which tests it was used. The determination of the amount of background traffic used in the simulations was also discussed. Following the construction of the framework, the evaluation and application tests were executed.

6. OPNET Modeler and SITL IPv6 Evaluation

Previous evaluations of IPv6 in OPNET Modeler have not sufficiently covered SITL's IPv6 support and IPv6 traffic in general; therefore, an evaluation of these areas was done before application testing. This chapter covers the scenarios used to assess IPv6, focusing on traffic generation, addressing and packet translation within the SITL module. It also presents solutions and attempts to fix the deficiencies of Modeler and SITL revealed by the evaluation.

The results of this chapter are summarized in Table 6.1 noting success or failure for the four communication areas examined in this evaluation. These areas describe the direction of traffic flow from source to destination. The first communication area has three subcategories as there are three ways to generate simulated traffic. In this table and the remainder of the chapter, "Internal" refers traffic generated within simulation. "External" refers to traffic generated by a real device outside of simulation. "Out of the Box" refers to using the simulation product as it was installed. "Local Solution" is a solution that has been found to make the simulation work if it did not work "Out of the Box." All simulations for this chapter were run on a Dell OptiPlex running Windows 7 Enterprise 32-bit with a 2.8 GHz processor and 4 GB of RAM unless otherwise specified and the Virtual Machine configuration (explained in Chapter 4) for SITL was used unless otherwise noted.

Communication Type	Out of the Box	Local Solution*
Internal to Internal: Generation Traffic	No	No Solution
Internal to Internal: Flow Traffic	No	Use Explicit Traffic Only
Internal to Internal: Application Traffic	Yes	Not Applicable
Internal to External	No	No Solution
External to Internal	No	Static Assignment of Address and Gateway
External to External	No	Code Change to "ipv6_nd" Module

***Solutions found as a result of the research described in this thesis**

Table 6.1 IPv6 Evaluation Results

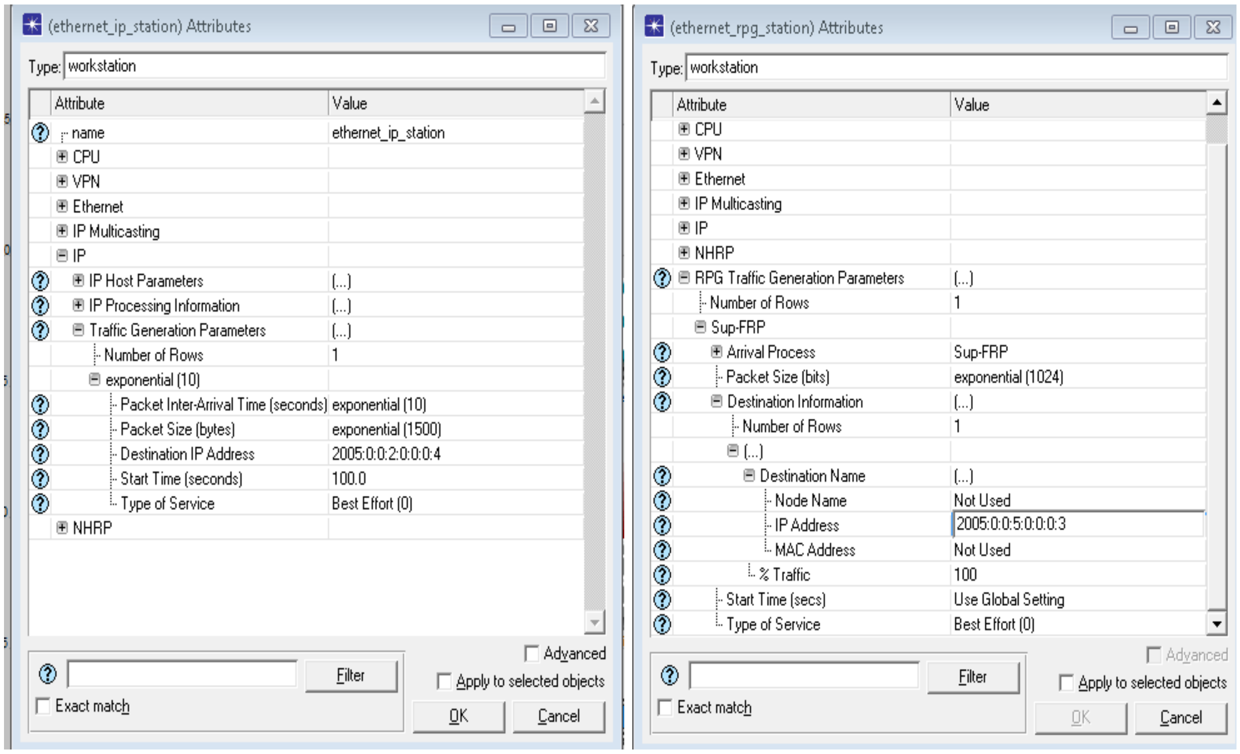
6.1 Internal to Internal Evaluation

The Internal to Internal Evaluation will cover the assessment of the IPv6 model completely within simulation. Chang et al. [14] did an evaluation of this model in 2003 but did not cover traffic generation. The focus for these test scenarios was to examine the different traffic types and their success in operating in IPv6.

6.1.1 Traffic types

Modeler has two types of traffic: explicit and implicit. Explicit traffic is traffic that generates a discrete event for every packet traversing the network and takes a significant portion of memory resources when simulations are large. Implicit traffic, traffic in which loads are simulated analytically, is used for these larger simulation situations. According to OPNET documentation, using implicit traffic results in fewer discrete events [3]. The implicit loads influence explicit traffic and impose effects on simulated devices.

There are three ways to create traffic in simulation. The first way is through traffic generator modules. These modules have a specific process that creates packets at a user defined frequency with a user defined size. The two modules in Modeler that have this process are the *ethernet_ip_station* and the *ethernet_rpg_station.* These two stations both generate packets but differ in how the frequency of the packet generation is determined. In the *ethernet_ip_station* module, frequency is decided based on statistical distributions such as exponential, poisson, etc. The *ethernet_rpg_station* module determines frequency by using fractal point processes to more accurately model the burstiness of real network traffic [22]. The configuration menu for these nodes is shown in Figure 6.1. The traffic generated by these modules is explicit. This type of traffic is referred as "generation traffic."


(ethernet_ip_station) Attributes

Type: workstation

Attribute	Value
name	ethernet_ip_station
⊞ CPU	
⊞ VPN	
⊞ Ethernet	
⊞ IP Multicasting	
⊟ IP	
⊞ IP Host Parameters	(...)
⊞ IP Processing Information	(...)
⊟ Traffic Generation Parameters	(...)
Number of Rows	1
⊟ exponential (10)	
Packet Inter-Arrival Time (seconds)	exponential (10)
Packet Size (bytes)	exponential (1500)
Destination IP Address	2005:0:0:2:0:0:0:4
Start Time (seconds)	100.0
Type of Service	Best Effort (0)
⊞ NHRP	

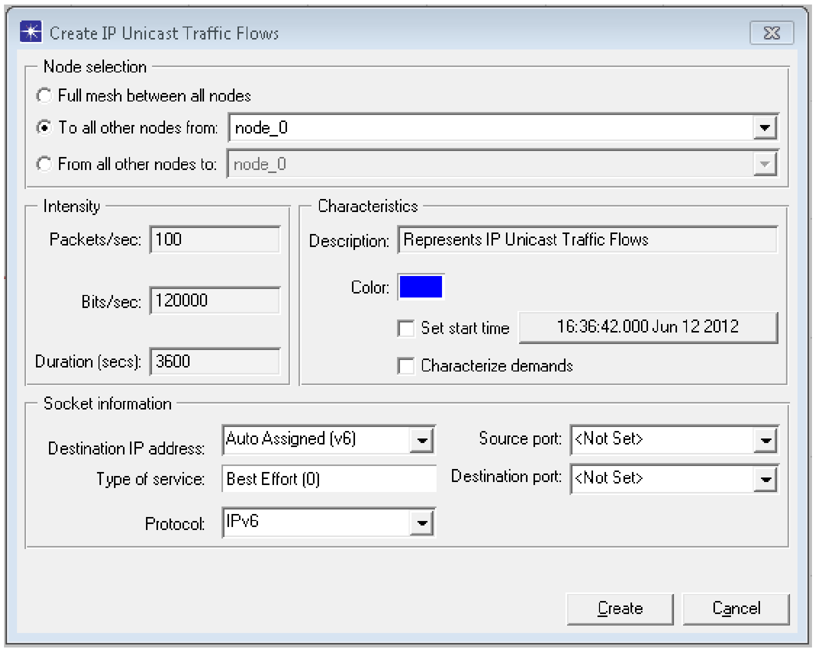
(ethernet_rpg_station) Attributes

Type: workstation

Attribute	Value
⊞ CPU	
⊞ VPN	
⊞ Ethernet	
⊞ IP Multicasting	
⊞ IP	
⊞ NHRP	
⊟ RPG Traffic Generation Parameters	(...)
Number of Rows	1
⊟ Sup-FRP	
⊞ Arrival Process	Sup-FRP
Packet Size (bits)	exponential (1024)
⊟ Destination Information	(...)
Number of Rows	1
⊟ (...)	
⊟ Destination Name	(...)
Node Name	Not Used
IP Address	2005:0:0:5:0:0:0:3
MAC Address	Not Used
% Traffic	100
Start Time (secs)	Use Global Setting
Type of Service	Best Effort (0)

Figure 6.1 Traffic Generation Menus: *Ethernet_ip_station* (left), *Ethernet_rpg_station* (right)

The second way to generate traffic is through traffic flows. A traffic flow is a series of packets sent from a source to a destination. A flow specifies which nodes are the source and destination, whether the flow is unicast or multicast, the size and frequency of the packets, and the type of traffic (explicit or implicit). Traffic flows are created through the traffic center menu in Modeler, shown in Figure 6.2, which provides the options to set the flow parameters. Traffic flows can be used to create both types of traffic. The percentage of explicit and implicit traffic generated can be set for each flow in further menus. This type of traffic is known as “flow traffic.”


Create IP Unicast Traffic Flows

Node selection
( ) Full mesh between all nodes
(●) To all other nodes from: node_0
( ) From all other nodes to: node_0

Intensity
Packets/sec: 100
Bits/sec: 120000
Duration (secs): 3600

Characteristics
Description: Represents IP Unicast Traffic Flows
Color: 🟦
☐ Set start time 16:36:42.000 Jun 12 2012
☐ Characterize demands

Socket information
Destination IP address: Auto Assigned (v6)
Type of service: Best Effort (0)
Protocol: IPv6
Source port: <Not Set>
Destination port: <Not Set>

Figure 6.2 Flow Attributes Menu

The third way of creating traffic is through the Profile and Application Definition modules. The Application Definition module defines application traffic with the type of application (HTTP, FTP, etc.), the amount of traffic sent per request, and when the requests occur in simulation. The Profile Definition module creates profiles containing a set of defined application traffic. The profiles are then assigned to desired end hosts. Servers also have to be assigned to serve the application traffic to the end hosts. The available applications in Modeler are HTTP, FTP, Database, Email, Print, Remote Login, Video, Voice and custom (a user defined application). Each of the applications can be set with a low, medium, or high traffic demand. A user also has the option to create a custom demand. The traffic generated by these modules is considered to be explicit. The two module icons are displayed in Figure 6.3 and the Application Configuration menu is displayed in Figure 6.4. This type of traffic is known as “application traffic.”

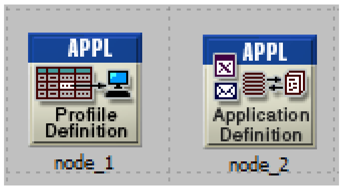
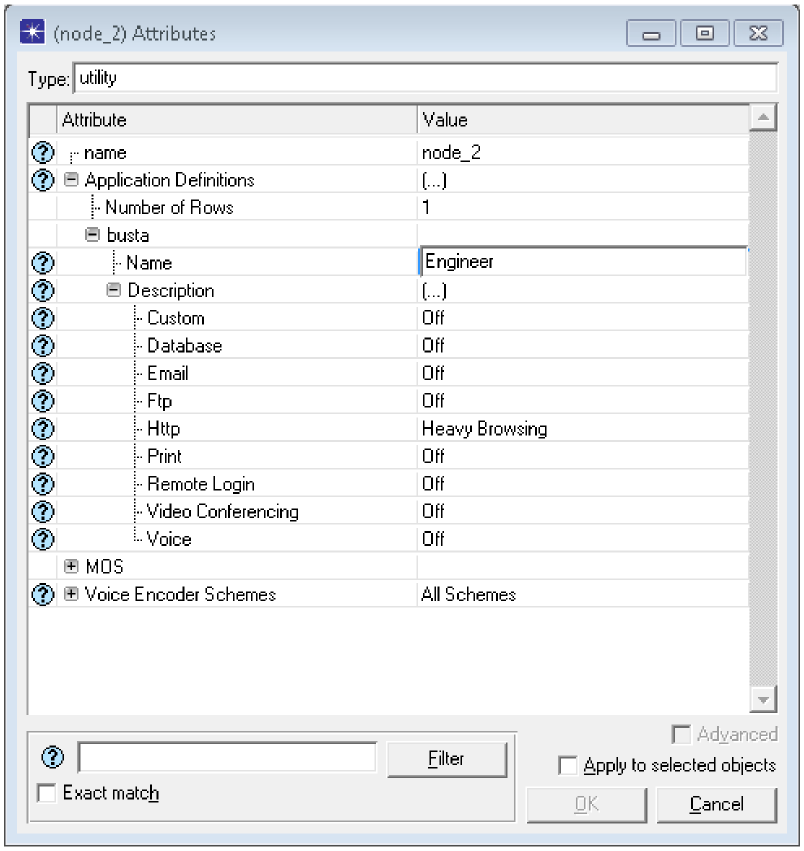


Figure 6.3 Profile and Application Modules



(node_2) Attributes

Type: utility

Attribute	Value
name	node_2
Application Definitions	(...)
Number of Rows	1
busta	
Name	Engineer
Description	(...)
Custom	Off
Database	Off
Email	Off
Ftp	Off
Http	Heavy Browsing
Print	Off
Remote Login	Off
Video Conferencing	Off
Voice	Off
MOS	
Voice Encoder Schemes	All Schemes

Figure 6.4 Application Configuration Menu

Modeler has a few traffic generating options for links. One option is a specialized link that creates a traffic flow between two connected devices that has the same parameters as flows created through the traffic center menu. Another option is the creation of ping traffic. Ping traffic is assigned with its own link and has three parameters: the source and destination addresses, the protocol used (IPv4 or IPv6), and the number of sent pings. Link traffic was deemed IPv6 ready based on the previous work by Chang et al. [14].

6.1.2 Testing Generation Traffic

The Two Hop with Background Traffic topology was used to test generation traffic. The *ethernet_ip_station* and the *ethernet_rpg_station* modules were used as end hosts in the network. The deployment of IPv6 addresses to the end hosts was done through the rapid assignment option in Modeler's Protocols menu and the traffic generator's default attribute settings for the frequency of packet arrival and packet size were used for each node. The default settings are an exponential distribution of 10 seconds for frequency and an exponential distribution of 1500 bytes for packet size. The last parameter, destination address, was statically assigned to the process, completing the set up for the generator modules. Statistics capturing for sent and received IPv6 traffic was enabled on each node. The first simulation was run was with the Optimized kernel to make a quick determination of support. The second simulation was run was with the Development kernel to be able to use the debugger to obtain further details of the simulation.

Statistics from the Optimized kernel simulation showed that no IPv6 traffic was sent or received by the traffic generating nodes. Since OPNET advertises IPv6 support, this result was unexpected. The debugger was an essential tool to figuring where the packet generation was failing. During the simulation with the Development kernel, the debugger showed that the given destination IPv6 address was invalid. Further investigation using the debugger showed that the packet was being destroyed in the internal IP encapsulation process. This process is a part of both the *ethernet_ip_station* and *ethernet_rpg_station* models. The full packet was never formed so no statistics could be gathered. The process model of *ethernet_ip_station* is shown in Figure 6.5 with the red arrow pointing out the IP encapsulation process in which the packet generation fails.

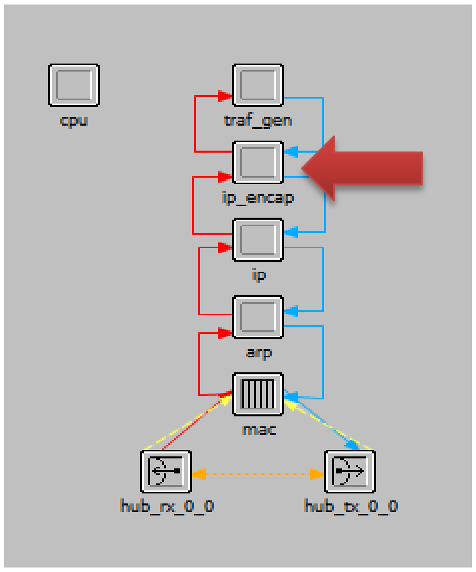


Figure 6.5 Process Model of *Ethernet_ip_station*

Despite the packet being destroyed in the IP encapsulation process, the real source of the invalid input was caused by the traffic generation process, *traf_gen* which uses several data types in its source code that pertain to IPv4 only. First, the variable that contains the IP address was an integer type named IpT_Address. A 32-bit integer is too small to store an IPv6 address. Modeler has a placement structure for IpT_Address called InetT_Address which is a structure that contains a variable for protocol type (IPv4 or IPv6) and a union data type which contains an integer for an IPv4 address and a pointer for an IPv6 address. *The traf*_gen process needed to be updated with the InetT_Address variable to enable support IPv6. The second improper data type was the communication structure between processes. The *traf_gen* process used the IPv4 communication structure to pass information to the IP encapsulation process. Modeler provides an IPv6 capable communication structure that could be used in the generation process. These two data type changes were made to the code which can be found in Appendix A. Despite these changes, a correct IPv6 packet could not be properly formed in simulation by the traffic generation process. The debugger revealed that the IPv6 address was now valid but the communication process continued to fail. This is an area for future work and a bug report was filed with OPNET's Technical Support group.

The IPv6 generation traffic test was deemed a failure because IPv6 packets could not successfully be generated and transported between processes.

6.1.3 Testing Flow Traffic

Flow traffic was tested second after generation traffic. The setup of for this assessment was similar to the generation traffic tests, the one difference being the type of end hosts in the Two Hop with Background Traffic topology. The end hosts were Ethernet workstations instead of *ethernet_ip_stations*. Ethernet workstations use a different set of processes to generate traffic than the *ethernet_ip_stations*. The Ethernet workstation process module is shown in Figure 6.6.

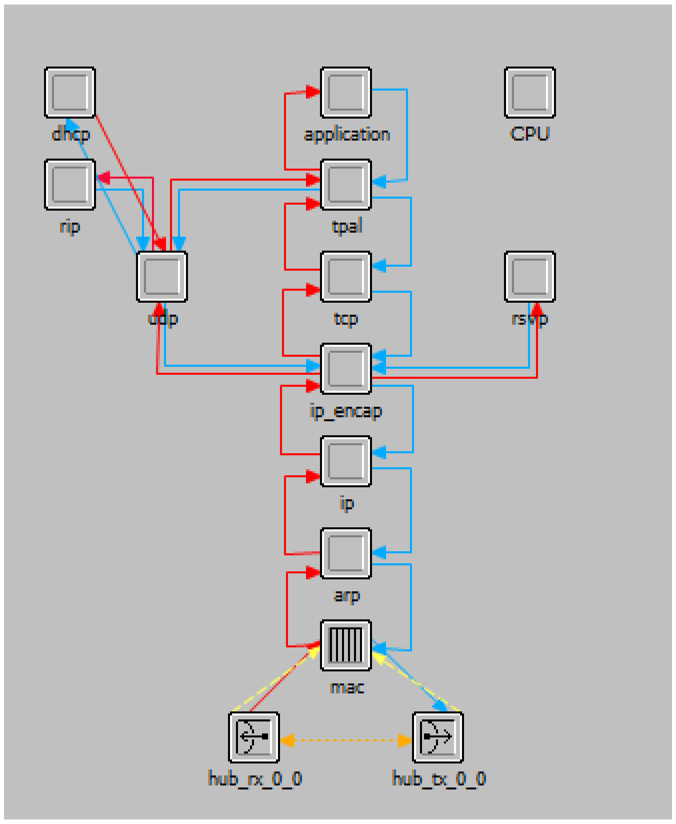


Figure 6.6 Ethernet Workstation Process Module

The IPv6 addresses were assigned with the rapid deployment method and statistics were configured to capture on each node for sending and receiving of IPv6 packets.

Flow traffic was created using the traffic menu. Specifically, a set of IP Unicast traffic was deployed between all nodes meaning that each node sent a load of traffic to every other node in the network. The load was set at 100 packets per second, with 1500 bytes per packet using IPv6. The last item of configuration was the setting the ratio of explicit to implicit traffic. Three scenarios with different ratios were run: one with all implicit traffic, a second with a 50% ratio of explicit and implicit traffic, and one with all explicit traffic. These scenarios were run for 5 minutes using the Optimized kernel.

The only successful scenario was the one ran with all explicit traffic. The statistics showed that with an all explicit test, the correct value of 100 packets per second was captured. The 50% ratio scenarios results showed half of the expected packet flow, 50 packets per second. The 50% ratio results are displayed as a graph in Figure 6.7. The configuration menu showing the 50% explicit/implicit ratio is displayed in Figure 6.8. The statistics for the all implicit scenario show zero IPv6 traffic sent or received. These results indicate that IPv6 traffic is only recognized when it is explicit traffic. Since explicit traffic creates more events in simulation, these results also imply that IPv6 simulations will require more memory resources and will have limitations on their size.

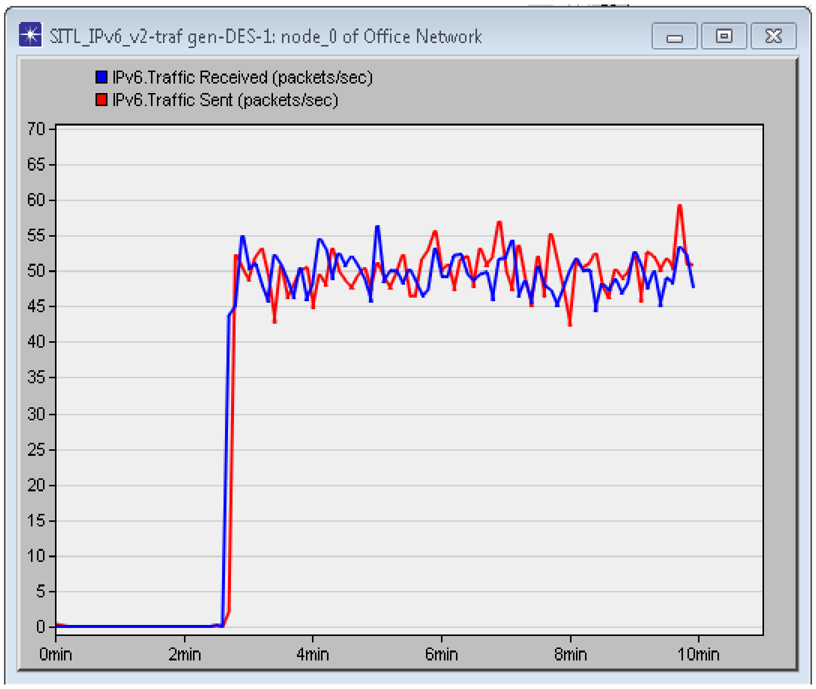
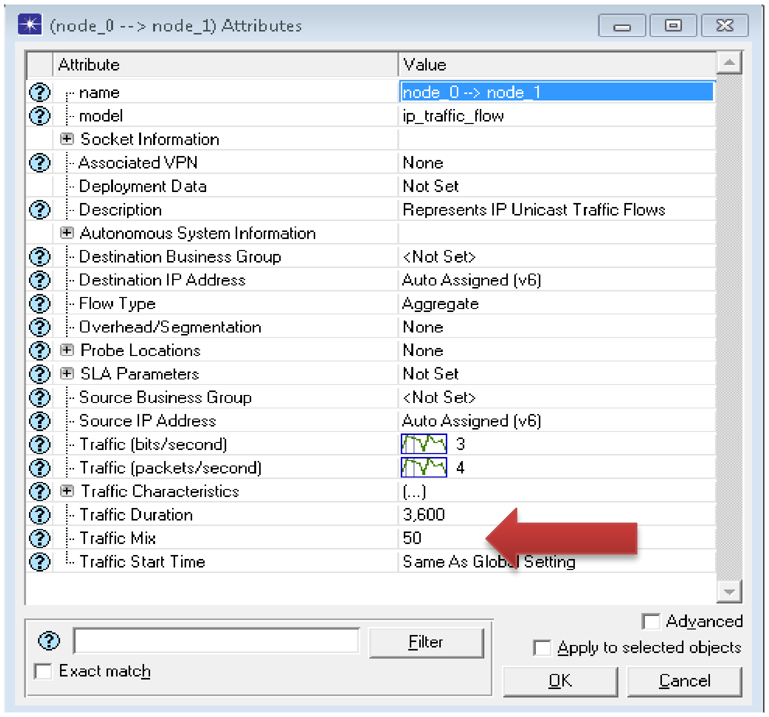


Figure 6.7 Flow Results and Configuration Menu



Attribute	Value
name	node_0 --> node_1
model	ip_traffic_flow
Socket Information	
Associated VPN	None
Deployment Data	Not Set
Description	Represents IP Unicast Traffic Flows
Autonomous System Information	
Destination Business Group	<Not Set>
Destination IP Address	Auto Assigned (v6)
Flow Type	Aggregate
Overhead/Segmentation	None
Probe Locations	None
SLA Parameters	Not Set
Source Business Group	<Not Set>
Source IP Address	Auto Assigned (v6)
Traffic (bits/second)	3
Traffic (packets/second)	4
Traffic Characteristics	(...)
Traffic Duration	3,600
Traffic Mix	50
Traffic Start Time	Same As Global Setting

Figure 6.8 Configuration Menu with 50% Implicit to Explicit Ratio

The flow traffic evaluation was deemed a partial success because flow traffic works for IPv6 in explicit mode only but does not work in implicit mode.

6.1.4 Testing Application Traffic

Application traffic was tested in the same manner as the generation and flow traffic types using the Two Hop with Background traffic topology. In this case, the end hosts were a combination of Ethernet workstations and Ethernet servers. Application traffic requires the use of the Profile and Application Definition modules as discussed previously. Three application types were chosen and configured: HTTP, FTP and Database. Within the Application Definition module, the HTTP application was configured to be a set of five “large images.” A “large image” averages 6,000 bytes in size. The FTP application was configured to retrieve 10MB sized files. The Database application was configured to be a high load which is defined as repeated 32kB sized transactions. These three applications were combined into one profile named “IPv6 test” in the Profile Definition module. “IPv6 test” was assigned to all of the Ethernet workstation end hosts and each application was assigned a dedicated Ethernet server. The simulation was then run for 5 minutes using the Optimized kernel while capturing IPv6 traffic in the statistics.

The results show that these tests were successful. IPv6 traffic was sent and received as expected. This evaluation confirms that application traffic is the only method that works without any unexpected errors. It is interesting to note that all studies found concerning IPv6 tests in Modeler mention application traffic as their mode of traffic generation.

6.1.5 Conclusion for Internal to Internal

This section evaluated three traffic generation types available in OPNET Modeler. The traffic generation was contained in simulation, meaning there was no interaction with a live network. From the simulation tests it was found that two out of the three traffic types could support IPv6. One of those methods, flow traffic, had a limitation in that it only registered explicit traffic. It was determined that Internal to Internal communication with IPv6 is partially successful because of these results.

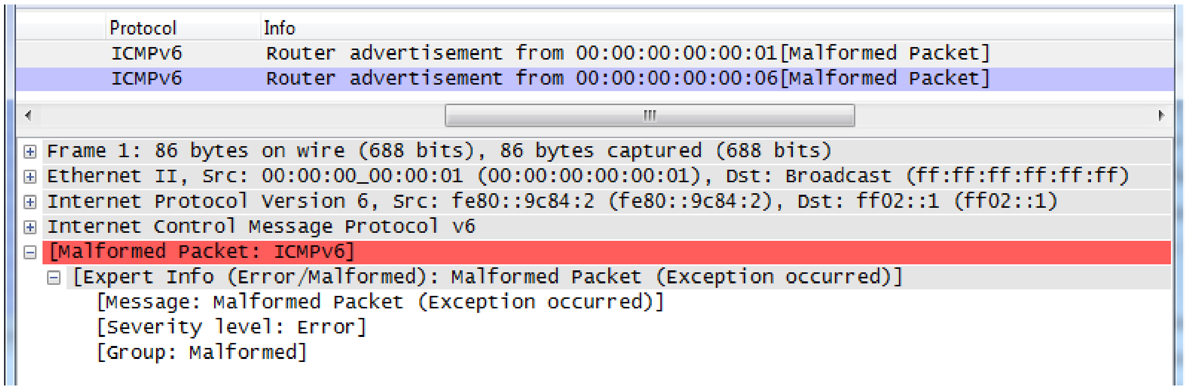
6.2 Internal to External Evaluation

Internal to External evaluation tests traffic generating internally in simulation that is sent to an external node through the SITL process. The Two Hop with Background traffic topology was used for these tests but traffic generation was limited based on the previous conclusions on IPv6 capable traffic type from the Internal to Internal tests. The first attempt to send traffic from a simulated node to an external machine was with using application traffic. This attempt failed because the SITL process could not be assigned a profile, nor could it be assigned as an application server. Flow traffic likewise failed because the SITL process was not an option for the destination of a flow. A last attempt was launched using the ping traffic link model. The ping traffic link model did not allow for an IPv6 address to be entered manually; only addresses defined within the simulation space were accepted as input. No other option for generating IPv6 traffic in simulation to be sent to a live destination was found. The Internal to External evaluation was deemed a failure because of the lack of IPv6 Internal traffic generation support.

6.3 External to Internal Evaluation

The External to Internal evaluation tested communication sent from a real device to a simulated device through the SITL module. The One Hop topology was used to test this communication with the router acting as the simulated node receiving traffic. ICMPv6 traffic, specifically NDP and ping messages, was the type of traffic sent into simulation. For SITL to execute properly, the simulation kernel has to be set to run at real time speed. The simulation was run for one hour after setting the correct packet filter and interface for the SITL process. The one hour time period was used to ensure there was enough time to send a variety of packets to the simulated node before the simulation completed. The Development kernel was chosen for simulation so that the debugger could be used.

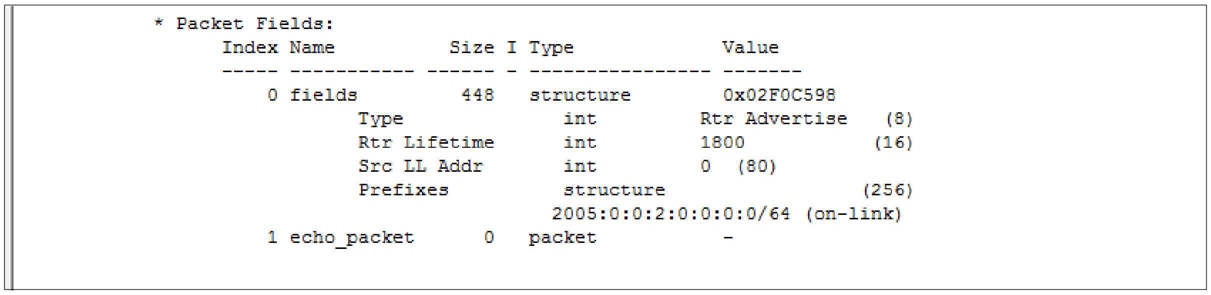
Communication was not successful on the first try of sending a ping message into simulation. It was discovered that the live machine was not receiving a Router Advertisement from the simulated router. A Router Advertisement is needed because it contains the subnet prefix for the network which is used by the live machine to form its full IPv6 address. The packet in the debugger was reported successfully translated despite the live machine not receiving it. Wireshark, packet capturing software [23], was used during a second trial to see if the packet was truly leaving the simulation and being sent to the live machine. Wireshark reported that the Router Advertisement packet from simulation was being translated but it was malformed. The Wireshark packet and the debugging console are presented in Figure 6.9 and Figure 6.10. The packet in Wireshark did not contain any subnet information, but the simulation packet has this information. To facilitate communication, an address and gateway was statically assigned to the real device. The gateway for the interface had to be assigned as the first simulated device's address connected to the SITL process. The simulation was run again; this time it was successful.



```
Protocol   Info
ICMPv6     Router advertisement from 00:00:00:00:00:01[Malformed Packet]
ICMPv6     Router advertisement from 00:00:00:00:00:06[Malformed Packet]

Frame 1: 86 bytes on wire (688 bits), 86 bytes captured (688 bits)
Ethernet II, Src: 00:00:00_00:00:01 (00:00:00:00:00:01), Dst: Broadcast (ff:ff:ff:ff:ff:ff)
Internet Protocol Version 6, Src: fe80::9c84:2 (fe80::9c84:2), Dst: ff02::1 (ff02::1)
Internet Control Message Protocol v6
[Malformed Packet: ICMPv6]
  [Expert Info (Error/Malformed): Malformed Packet (Exception occurred)]
    [Message: Malformed Packet (Exception occurred)]
    [Severity level: Error]
    [Group: Malformed]
```

Figure 6.9 Malformed Router Advertisements in Wireshark



```
* Packet Fields:
    Index Name          Size I Type             Value
    ----- ----------- ------ - ---------------- -------
        0 fields         448    structure        0x02F0C598
                Type              int          Rtr Advertise    (8)
                Rtr Lifetime      int          1800            (16)
                Src LL Addr       int          0   (80)
                Prefixes          structure                   (256)
                                2005:0:0:2:0:0:0:0/64  (on-link)
        1 echo_packet      0    packet           -
```

Figure 6.10 Debugging Console with Router Prefix

This solution, while working, is not complete because it is does not solve the malformed packet translation. Static assignment also prevents dynamic routing. A look into SITL's documentation revealed that there are no IPv6 routing protocols supported for SITL translation at this time [34]. The External to Internal evaluation showed that SITL does not work "out of the box" but communication is possible and was achieved only after statically setting addresses.

6.4 External to External Evaluation

External to External communication is the communication of two real devices through a simulated network using the SITL module. There were several scenarios run to examine the full capability of translating IPv6 in and out of simulation. The first scenario used ping traffic and is referred to as the connectionless traffic test. The second scenario used transport layer traffic (TCP and UDP) through configuring HTTP applications and is referred to as the connection-oriented traffic test. The third scenario was a packet crafting exercise involving crafting specific packets to figure which IPv6 extension headers were supported. This scenario is referred to as the extension header test. Based on the results from these tests, a fourth scenario was developed to test the effects of the machine bit architecture on the simulation and simulation effects on machine resources. The One Hop Topology was used throughout the following situations.

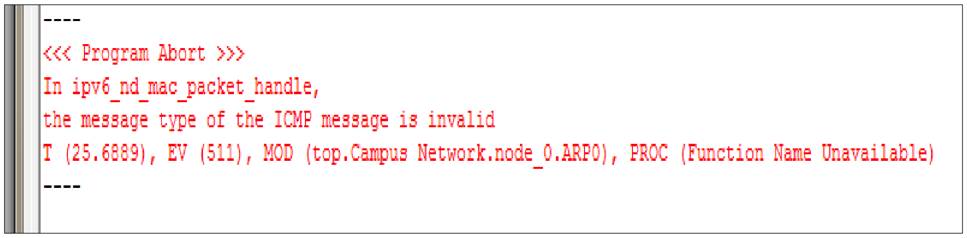
6.4.1 Connectionless Traffic Test

The connectionless traffic test required the real devices to be set with a static address and default gateway as was discovered from the External to Internal test. The simulation was run with the Development kernel in debugging mode after the appropriate settings were set in the SITL processes. The debugging mode displays packet information for each incoming and outgoing communication which helps to track which messages are being translated correctly and if any parts of a packet have been dropped.

Initially, the simulation crashed at the first ping message received from the SITL process. It aborted the simulation with an "improper ICMPv6 message type" error shown in Figure 6.11.

This was unexpected because the External to Internal test was successful. The only difference between this test and the previous is the destination address. This error needed to be solved so that application testing would be possible and the error message provided the first clue to pinpointing the problem.

The error message contained the location of the issue, specifically a process in Modeler called *ipv6_nd*. The source code for this process was modified to include a series of print statements to see the point in which it fails. Through these statements, it was found that when the ICMPv6 value was parsed by this process, it contained extra information in the upper bits of the variable. It was reasoned that this extra information came from a translating error in the SITL process because other simulation models did not create this same error. Since the translation code is not available to modify, the solution was to mask out the upper bits in the ICMPv6 variable. This was done with masking the variable with the value 0xFF in the *ipv6_nd_mac_packet_handle* function. The mask eradicated the information contained in the higher bits while maintaining the correct value for the message type in the lower 8 bits.



```
----
<<< Program Abort >>>
In ipv6_nd_mac_packet_handle,
the message type of the ICMP message is invalid
T (25.6889), EV (511), MOD (top.Campus Network.node_0.ARP0), PROC (Function Name Unavailable)
----
```

Figure 6.11 Invalid ICMP Message Error

Using the masked value allowed connectionless traffic to be successfully sent between two live hosts without aborting the simulation and the ping test was executed. The round trip time results displayed some interesting characteristics. Throughout the tests, the round trip times would vary at being consistently low, between 2 and 5ms, and jumping to 15-30ms. At the high round trip times, the values followed a decreasing pattern. The pattern started at the high round trip time, usually in the 30ms range and decrease around 3ms for 5 iterations. The round trip time would then jump back to the 30ms range and repeat. Since this pattern is not common in live networks, it was determined that it was being caused by Modeler. To find cause, the simulation was run again while capturing SITL related statistics including queue size, queuing delay and translation delay. It was found that the higher round trip time and decreasing pattern was coming from the queuing delay. In Figure 6.12, the SITL queuing delay is shown for one of the simulation ping tests. The delay is low for a period of time (when the round trip times average 2-5ms) and then suddenly changes to follow a rise and fall pattern (when the round trip times average 15-30ms). The rise and fall pattern is more clearly illustrated in Figure 6.13. There was no evident cause for the change in the SITL queuing delay so further tests were devised to investigate. These tests are discussed in section 6.4.4.

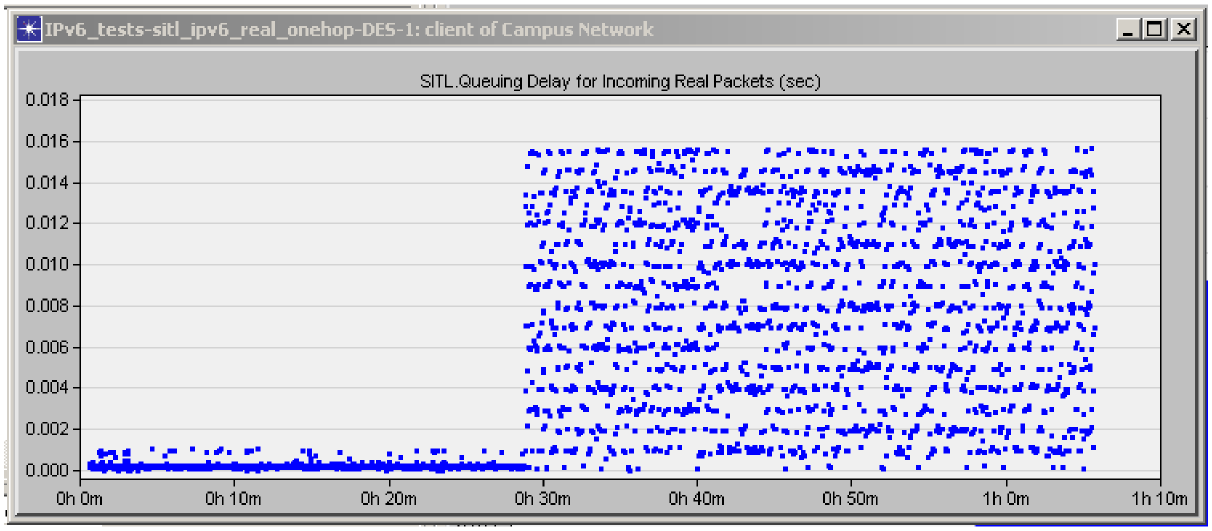


Figure 6.12 SITL Queuing Delay Variation in a Ping Test

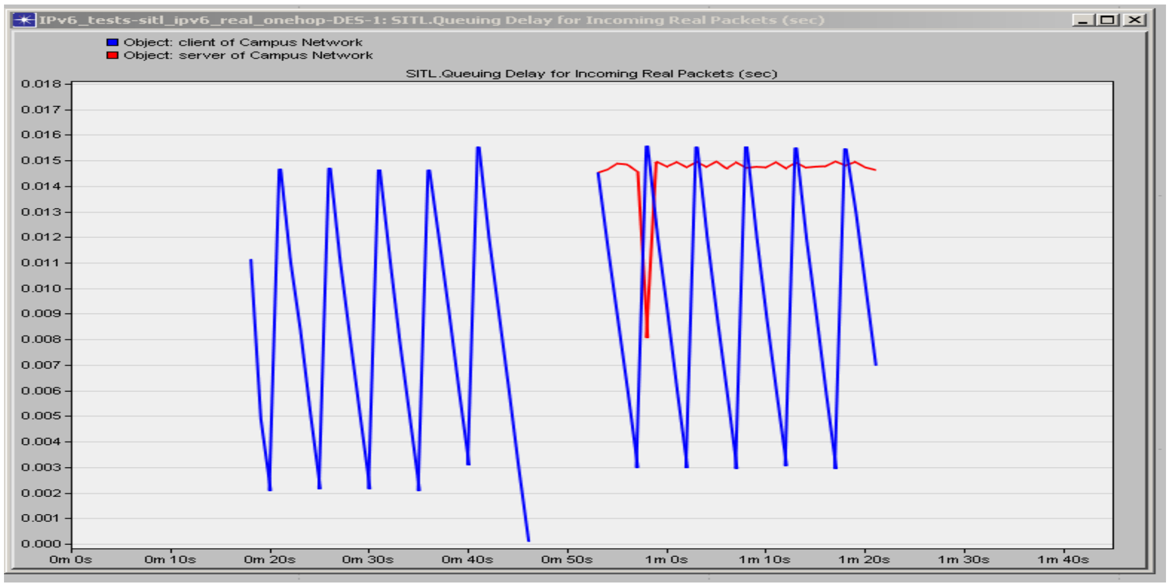
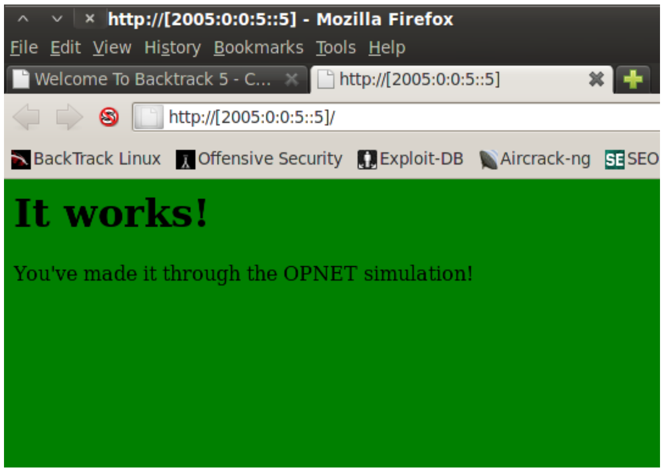


Figure 6.13 Rise and Fall Pattern of the SITL Queuing Delay

6.4.2 Connection-Oriented Traffic Test

The purpose of the Connection-Oriented tests was to confirm application traffic could be successfully transferred over the simulated network. The application traffic was tested with both TCP and UDP and required additional setup on the real devices for the applications. One device was designated as the application server and ran an Apache server that hosted a simple webpage and a set of binary files of various sizes. The webpage was used to test TCP and the binary files were used to test UDP through files transfer. After the server device was configured, the tests were performed.

A single simulation ran for one hour to ensure that all files were transferred completely and the webpage could be accessed. During the simulation, the webpage, shown in Figure 6.14, was accessed successfully through a web browser on the client machine. The file transfers were accomplished using the Wget [23] software package. The sizes of the files transferred varied from 1 kilobyte to 50 megabytes (MB). These files were transferred successfully without error. An additional media file was transferred to test the transfer of a large file. The media file was an 800MB mp4 video. The video was played on the client after the transfer with no visible or audible problems. Streaming media through simulation was not tested at this time.


http://[2005:0:0:5::5] - Mozilla Firefox
File Edit View History Bookmarks Tools Help
Welcome To Backtrack 5 - C...
http://[2005:0:0:5::5]
http://[2005:0:0:5::5]/
BackTrack Linux
Offensive Security
Exploit-DB
Aircrack-ng
SEO

It works!

You've made it through the OPNET simulation!

Figure 6.14 Website Accessed Through Simulation

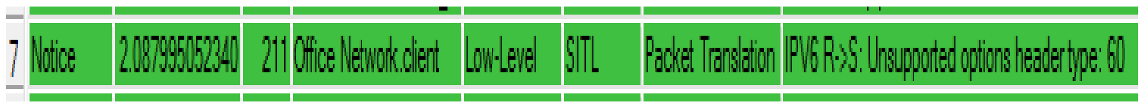
These tests were regarded as a complete success.

6.4.3 Extension Header Test

The Extension Header test was done to verify that the entire IPv6 packet header was being translated correctly by SITL. This particular evaluation was done because MT6D uses the Destination Options extension header as a way to transfer an encryption option, an optional feature of the application but enabled by default. The IPv6 header can contain other extension headers as discussed in Chapter 2. These extension headers contain information for IPSec, Mobility, and others. They are an essential part in how IPv6 contains optional information; therefore SITL should support extension header translation.

The method of testing extension headers was to craft packets with each possible option send it to a simulated node. The extension header was determined to be supported based on the output of the simulation event log; this log was written to if the header was unsupported, i.e., not translated. Packets were crafted using Scapy [25]. Scapy is a free packet manipulation program written in Python that supports IPv6. One packet per extension header was crafted, sent to the simulated node, and the log result was recorded. An example log result showing that the

Destination Options extension header (type 60) was unsupported is shown in Figure 6.15. Wireshark was used to confirm that each packet was crafted correctly and that the translation function was the cause.



```
7  Notice  2.087995052340  211  Office Network.client  Low-Level  SITL  Packet Translation  IPV6 R->S: Unsupported options header type: 60
```

Figure 6.15 Log Showing Unsupported Destination Options Extension Header

The full results of this test containing all extension headers are displayed in Table 6.2.

Extension Header	**Result**
Hop-by-Hop Options	Fail
Destination Options	Fail
Routing Header	Fail
Fragment Header	Fail
Authentication Header	Fail
Encapsulation Security Payload Header	Fail
Mobility	Success

Table 6.2 Extension Header Test Results

Only Mobility extension headers are supported by the SITL translation function. This posed an issue with fully testing MT6D because it uses Destination Options. SITL does allow for the flexibility of creating a custom translation function for protocols which could solve the lack of extension header support. The translation function is the core of the SITL process as it translates real packets to and from the simulated packet formats. A custom translation function was written to attempt to translate Destination Options packets so that MT6D’s encryption option could be enabled for testing. The custom translation simulation tests were run using the real device configuration for SITL.

The translation function method was to examine the incoming IPv6 header's next header field. If this field had the value of "60" then this signified that the next header was a Destination Option extension header. The result of this condition was used to determine if the new translation function should be used. If condition was false, the default translation function was used. The custom translation encapsulated the rest of the packet after the network layer as the information in further OSI layers was not needed in simulation. The custom code for this translation function is located in Appendix B. This code successfully translated the packet in and out of simulation; however, the receiving machine returned an ICMPv6 message containing a Parameter Problem type indicating there was a problem in translation. Packet traces were captured on both live machines and the Destination Option packet was identified in both of these traces. In the traces, the packet was identical with an expected change in the hop limit field. Additional changes to the custom translation code did not yield other results and it was not clear as to why the receiving machine did not accept the packet. Based on real network tests, the encryption option for MT6D did not add significant overhead and therefore application tests were done with encryption disabled. More information on the application tests can be found in Chapter 7. Improving the translation function is an area for future work.

Based on the lack of support for the majority of the extension headers, this part of the evaluation was deemed a failure.

6.4.4 SITL Queuing Delay Tests

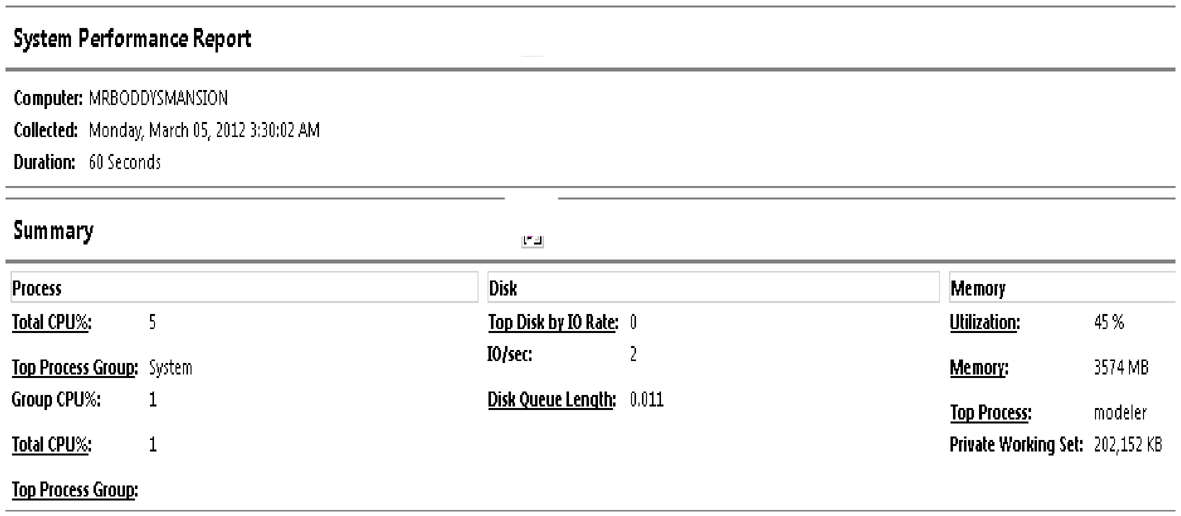
The External to Internal tests showed that the SITL queuing delay displayed unusual behavior. The reason for this behavior was sought because it affects packet latency. A number of hypotheses were developed on why the queuing delay would vary at random points in time. The first hypothesis was that it was caused by the availability of the simulation machine's physical resources. The second hypothesis was that the delay could be affected by the bit architecture of the simulation machine. The third hypothesis was that the variation was being caused by the

SITL translation function itself. Each of these hypotheses was tested and a benchmark test was developed from this work.

6.4.4.1 Hypothesis 1: Physical Machine Resources

The amount of physical resources available for the simulation is an important factor to building a simulation. The hypothesis was made that if there was a shortage of RAM and/or disk space, then the SITL queuing delay would increase as the simulation would have to wait for computation space to be freed. The simulation machine was configured to have no other processes running except for the essential processes of the operating system and the simulation. Graphics were set to run at a minimal level. After these adjustments were made, a schedule for performance monitoring was created using the Performance Monitoring tool for Windows. This tool ran a performance snapshot of the machine every 15 minutes (per the defined schedule), logging metrics such as RAM usage, disk space, and CPU usage.

A series of traffic was set up to run through the simulation to stress the system and was named “barrage traffic” as it continually fired traffic at the simulation. This barrage traffic was a sequence of 100,000 packet ping floods that occurred one after the other for a user defined amount of time. For the resource test, this time period was 8 hours. A simulation using the One Hop topology was run for this time period using the Optimized kernel and the queuing delay statistic was captured on the SITL processes. The performance monitoring reports showed that RAM and CPU usage stayed consistently low, under 10%. The summary of one of the reports is shown in Figure 6.16.


System Performance Report

Computer: MRBODDYSMANSION
Collected: Monday, March 05, 2012 3:30:02 AM
Duration: 60 Seconds

Summary

Process		Disk		Memory	
Total CPU%:	5	Top Disk by IO Rate:	0	Utilization:	45 %
		IO/sec:	2		
Top Process Group:	System			Memory:	3574 MB
Group CPU%:	1	Disk Queue Length:	0.011		
				Top Process:	modeler
Total CPU%:	1			Private Working Set:	202,152 KB
Top Process Group:					

Figure 6.16 Performance Monitor Report

The queuing delay still fluctuated in its decreasing pattern indicating that low system resources were not the cause.

6.4.4.2 Hypothesis 2: Bit Architecture

Another possible cause was bit architecture of the simulation machine. OPNET has both a 32-bit kernel and a 64-bit kernel available for use. Up until this point, all simulations were being run on a 32-bit Windows 7 Enterprise machine. Using the 64-bit OPNET kernel could make a difference in the SITL queuing delay. For this test, a second Dell OptiPlex with 16GB of RAM and a 3.4GHz processor was configured with the Windows 7 Enterprise 64-bit OS. OPNET Modeler was installed on this machine and the 64-bit kernel was selected to run simulations.

The testing scenario was built with the One Hop topology and the barrage traffic tests were executed with a time period of one hour. The results from this test were unexpected. The SITL queuing delay was twice as large as the delay on the 32-bit machine. The tests were repeated with the same results. A possible reason for this outcome is that the SITL module is not built for use on the 64-bit kernel which can be confirmed through examining which dynamic link libraries are used during the simulation. This is an area of future work. The overall result is that the bit architecture does not cause the SITL queuing delay variance but instead compounds the latency.

6.4.4.3 Hypothesis 3: Translation Function

The third possibility for the queuing delay variance is that the SITL translation function could be causing the delay. This hypothesis was tested by using the custom translation function written for IPv6 extension headers. The custom function would use the default translation for the packet after testing for the Destination Option, considering ping packets do not have extension headers. Using the custom test function makes the translation function on a whole different enough to test this hypothesis. The test was executed on the 32-bit machine with the barrage traffic tests for a period of one hour.

The results showed that the queuing delay stayed fairly consistent, averaging 5ms, indicating that the variance is most likely caused by the translation function. The conclusion that translation function causes the varying delay is reasonable but cannot by confirmed true without access to the SITL source code. Since this code is not made available from OPNET, there is still the possibility that the queuing delay variance is caused by another source.

6.4.4.4 Queuing Delay Test Conclusion and Benchmark Test

There could be other unknown factors affecting the SITL queuing delay because the three hypotheses tested did not reveal a definitive answer. The hypothesis of the translation function being the cause is the most likely from the test results. This is a coding issue that needs to be resolved by OPNET.

The barrage traffic test became a benchmark test during this time period. It is suggested that this test be used when investigating unexpected latency causes in an OPNET simulation. The continued traffic flow is consistent and ICMPv6 is a standard protocol that must be supported in any simulation. To reiterate the definition of this benchmark test, the barrage traffic test is a sequence of 100,000 packet ping floods that occur within a user defined time period. The suggested time period is one hour, which equates to 6-7 iterations of the ping flood depending on the latency of the network being tested.

6.4.5 External to External Conclusion

The External to External evaluation covered connectionless, connection-oriented and extension header traffic. It also explored latency caused by the SITL queuing delay which was noticed during the Connectionless traffic test. The final results from each test are that the connectionless traffic is that it is a success with fixes made in the *ipv6_nd* process, the connection-oriented traffic was a complete success, and the extension header test was a failure due to the lack of support and the custom translation function not solving the issue. Overall, the support of IPv6 in External to External communication is a success after applying changes to OPNET code but there is minimal extension header support.

6.5 Chapter Conclusions

This chapter covered an evaluation of IPv6 support in OPNET Modeler and the SITL module. IPv6 support exists within Modeler and SITL but there are several areas that need to be improved and fixed. Possible solutions to these deficient areas were presented including source code changes and a custom translation function. A benchmark test called the barrage traffic test was presented to test causes for variable latency in simulation. Part of this work was summarized in the publication “An Evaluation of IPv6 in Simulation Using OPNET Modeler” [26].

7. IPv6 Application Simulation and Results

The framework of topologies described in Chapter 5 was used to simulate two IPv6 applications. VoIP was chosen to represent a standard application and MT6D represented a custom application. The evaluation of IPv6 in Modeler found issues with the software and the solutions used to solve some of these problems were used to help facilitate this testing. MT6D simulation extends the work done by Dunlop [8]. This chapter will discuss the methodology for the simulations and the results for each application.

7.1 Methodology for VoIP

The methodology to test the VoIP application started with finding a standard voice communication. No standard musical sample used for benchmark VoIP testing was found which allowed freedom of choice in finding a suitable sample. A recorded musical selection was chosen to test the simulation to help be consistent across all tests. The chosen piece was Frederic Chopin’s Waltz in D flat major, more commonly known as the Minute Waltz, because it has an intricate melody, variations in volume and speed, and is a relatively short 93 seconds long. These qualities help in judging the quality of the sound being transferred because they keep the listener active. The simulation for testing the VoIP application was configured after determining the communication to be transferred. The three topologies chosen for VoIP testing were the One Hop, Two Hop and Two Hop with Background Traffic. For each topology, the music sample was played 5 times. These tests were run on the Virginia Tech IPv6 live network for comparison.

The Live Network configuration was used to connect the SITL model to the live network. The main reason for using this configuration was that USB microphones had to be used and the simulation machine did not have enough USB ports to support all of the peripherals. The second reason was that VoIP software was preconfigured on the live machines. The software used was Mumble, an open source voice chat that supports communication over IPv6 [27]. Mumble uses a custom protocol that includes voice pre-processing to remove noise and a low latency focus.

Both live machines ran the Mumble client and one machine acted simultaneously as the server. It did so by running Murmur, Mumble's server component.

VoIP has several metrics to record quality specifically packet loss, latency, jitter and two specialized values (R-Factor, MOS). Of these metrics, R-Factor and MOS were not obtained as certain factors needed for calculating the values, such as signal to noise ratio, were not available. Instead, a mean opinion of the quality was given by the members of the Virginia Tech ITSL as a substitute for the subjective metrics. Packet loss, latency and jitter were found by capturing packets on both machines and running a Python script that calculated these values from information in the traces. The definition used for the calculation of jitter is the variance of one way latency between two packets [29]. The Python script can be found in Appendix C.

7.2 VoIP Results

The live network average results for packet loss, latency and jitter are displayed in Table 7.1. The live results were used to compare with the simulation results. Since the live network contains traffic from other sources, the metrics of latency and jitter are affected. The variance of the latency is also a result of the being in the live network. There was no recorded packet loss. Two suppositions were made after looking at these results. First, the isolated network simulations should have a lower latency and jitter. Second, the simulation that includes simulated background traffic should affect latency and jitter to reflect the live network results.

Iteration	**Latency**	**Jitter**	**Packet Loss**
1	5.236553329	0.009035993	0
2	1.707461217	0.008349882	0
3	3.325838112	0.009644571	0
4	5.367741503	0.007616035	0
5	1.347498317	0.009643967	0
Average (ms)	3.397018495	0.00885809	0

Table 7.1 VoIP Live Network Results

The results of the simulation of the VoIP application displayed the behavior expected from looking at the live network results. The latency in the One Hop and Two Hop topologies (which are isolated networks), was extremely low with no packet loss. The values showed an increase in latency as expected with the introduction of simulated background traffic. The average latency was slightly higher than the live network but this was accounted for because the simulated traffic is based on the yearly average from the live network device (see Chapter 5 for more details on the determination of the magnitude of simulated traffic). There was no packet loss in the Two Hop with Background Traffic topology. The latency results are displayed in Table 7.2.

Iteration	**One Hop (ms)**	**Two Hop (ms)**	**Two Hop with Background Traffic (ms)**
1	1.959609454	1.936823181	3.505379865
2	1.205350084	1.857744399	3.593226925
3	1.486840805	1.838684012	3.646007423
4	1.855967771	1.868996167	3.671304864
5	1.910731942	2.325934282	3.6944148
Average	1.683700011	1.965636408	3.622066775

Table 7.2 VoIP Latency Results

Jitter was consistent across all topologies averaging 0.14 milliseconds which is significantly higher than the live network average result. The reason for increased jitter can be accounted for by the translation and queuing delays in SITL. As found in the IPv6 evaluation of SITL in Chapter 6, the queuing delay has a peculiar pattern when using the default translation function. This varying pattern would affect jitter. The simulation results for jitter can be found in Table 7.3.

Iteration	**One Hop (ms)**	**Two Hop (ms)**	**Two Hop with Background Traffic (ms)**
1	0.144140305	0.144176839	0.150815644
2	0.139052024	0.138130192	0.131669074
3	0.132289739	0.134167707	0.131669074
4	0.148806248	0.142763594	0.14830928
5	0.135799586	0.127802556	0.143646153
Average (ms)	0.14001758	0.137408178	0.141221845

Table 7.3 VoIP Jitter Results

The overall quality of the sound was rated as acceptable by the members of the ITSL. The melody was recognizable throughout the testing including when the music changed tempo and volume. On a scale from 1-5, the ITSL gave an average quality score of 4. This score was qualified with the statement: “For the Speex codec which Mumble uses, the quality was very good”.

Overall the VoIP application testing in simulation showed that the application followed expected behavior. Mumble, the VoIP application, does not use any specialized extension headers and operates using UDP; therefore, SITL’s default translation function supports the packets. This fact, in combination with the results, confirms that the developed framework has valid scenarios in which testing standard IPv6 applications is feasible. The scenario involving background traffic shows that the application is affected by outside influence. This influence should be taken into account when deploying VoIP into a production network.

7.3 Methodology for MT6D

MT6D testing methodology follows the same tests performed by Dunlop in his live network research of MT6D [8]. The first step was to configure MT6D and the live machine for simulation. Each machine was loaded with the MT6D application. MT6D has a set of configuration options which includes encryption and the address rotation time. For these tests, MT6D encryption was disabled because of the lack of extension header support in SITL. According to Dunlop, the latency incurred by the encryption option of MT6D was minimal; therefore, disabling this option should not greatly affect the simulation results [8]. The following address rotation times were tested: 5 seconds, 10 seconds, 15 seconds, 30 seconds and no rotation. For each rotation time, a set of simulations were run. A control test for the testing script was run by simulating the script without using MT6D. The testing script was a combination of the tests used to evaluate Modeler and SITL in Chapter 6. The full testing script included a series of a 1,000 packet standard ping, 10,000 packet ping flood, 50,000 ping flood, and files transfer using Wget of sizes 500kB, 1MB, 10MB, 50MB, 500MB, 1GB. This series of tests was run for 10 iterations and an average from these tests was recorded. The metrics for this application were latency, packet loss and packet retransmissions. Results were obtained through packet traces captured on the live machines. The second step of configuring the test was choosing the simulation networks and configurations. MT6D was run on the Live Configuration for SITL and the topologies used for these test were the One Hop, Two Hop and Campus topologies. The simulations were run with the Optimized kernel over a time period of 10 hours. This long time period was chosen because the average time to run the test script was 8 hours. The average time was determined through calculating the time each file transfer and ping communication would take with the simulation network speed (noted during the Modeler evaluation).

7.4 Unexpected Behavior from the First Set of Results

The first set of results obtained using the One Hop network displayed high latency and high packet loss for every address rotation time. These results were unexpected because the values were higher than the live network tests. From the results of the VoIP simulation, the isolation of the network should reflect reduced latency in simulation. Further investigation was needed to find the cause of the increase values. The topologies were checked for any inconsistencies in the node and link configurations and it was found that the links used in simulation had a lower bandwidth. These links were increased to the next level, namely 1Gbps and 10Gbps. The increased bandwidth links decreased the latency in a second set of tests to a range closer to the live network tests.

The change of the links helped improve latency but further action was taken to find ways to improve the values more. Packet traces were examined for any indication of anomaly. The packet traces revealed that the Destination Options extension header was still being built for MT6D packets even though encryption was disabled. SITL drops the extension header while translating but there is a cost in this translation effort. The discrete event log from the simulation confirmed that these headers were being captured and subsequently dropped. This error in the MT6D source code was fixed and the tests performed with the updated application showed that the Destination Options header was no longer built when encryption was disabled. The application error could account for both part of the increased latency and packet loss seen in the first set of simulation results.

Statistics captured from within Modeler were examined for items that could affect latency and packet loss as well. These statistics included the SITL queuing delay, translation delay, IPv6 packets received by each node, and IPv6 packets sent by each node. The queuing delay and translation delay added to the latency but these were already taken into consideration when examining the results. The IPv6 packet statistics showed a significant amount of dropped packets. These dropped packets occurred in chunks indicating that this was not due to a normal network occurrence. The IPv6 packet statistics for the client running a 100,000 packet ping flood is shown in Figure 7.1. The blue trace is the traffic received from the external (live) network and the red trace is the traffic sent to the external (live) network. The figure shows four 10 second periods of dropped packets. To make sure this pattern of dropped packets was not an anomaly, another ping flood test was run. This test displayed the same packet dropping characteristics.

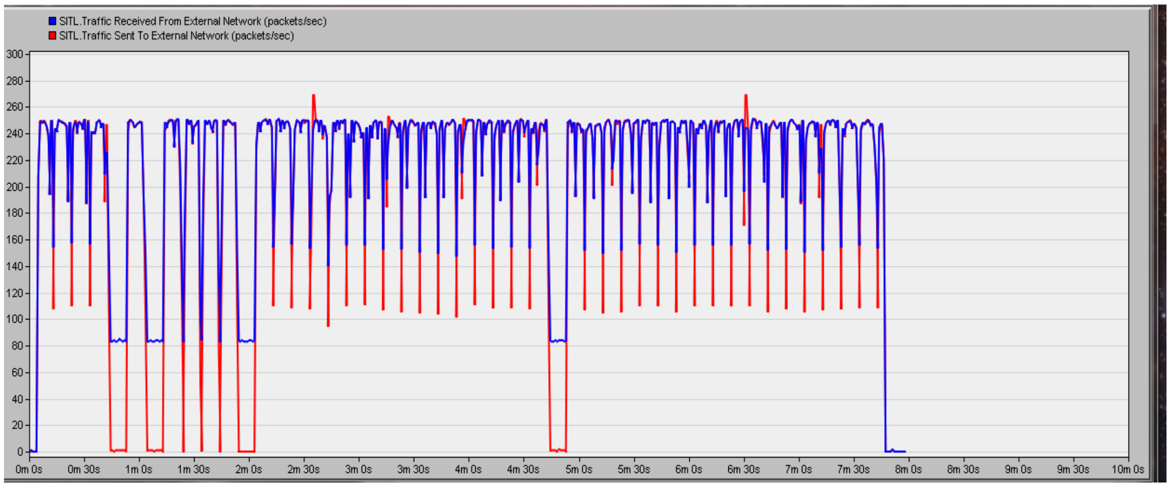


Figure 7.1 MT6D Dropped Rotations

It was postulated that the reason for this packet loss pattern was a timing issue within MT6D. The source code for MT6D was examined but no clear timing issue was found. After some additional testing on the live network and debugging the code, an error was found in how the application was binding addresses. This error was fixed in the code by binding the addresses more frequently. The updated MT6D code was tested with a 100,000 packet ping flood to see if the dropped packets still happened. The results are shown in Figure 7.2 and it is clear that there are no longer significant periods of dropped packets.

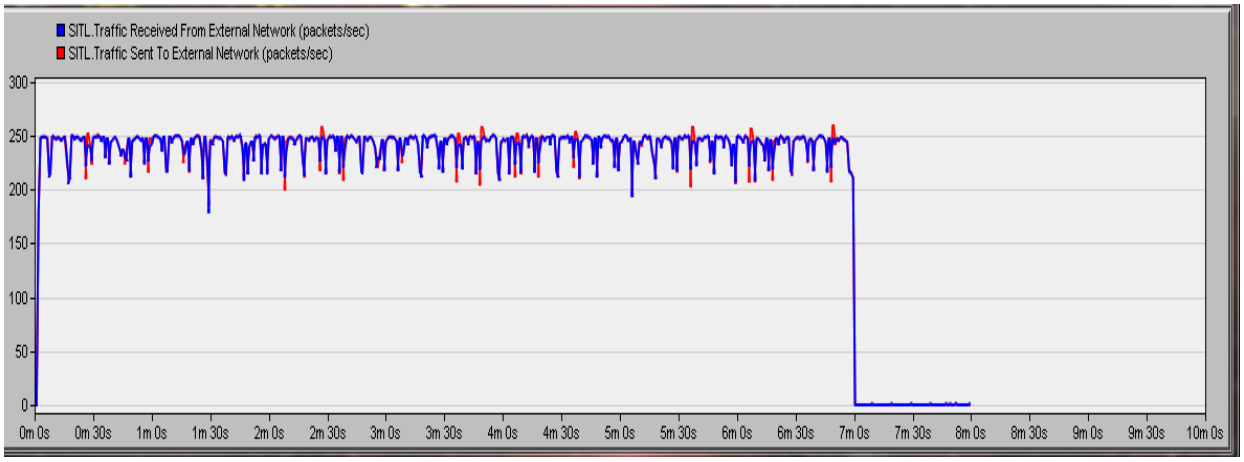


Figure 7.2 Updated MT6D with No Dropped Rotations

These two issues found within the MT6D and the simulation links accounted for the increased latency and packet loss seen in the first set of results. After the changes to the MT6D application were made and the simulation topologies updated, the series of tests were repeated for a more accurate set of results. The MT6D coding changes also caused live networking testing to be repeated. The updated live results were used for future comparisons.

7.5 MT6D Results

The results from MT6D presented several items of note. In the control test (testing script run without MT6D), the maximum throughput averaged 3MB per second for file transfers despite the simulated links being able to carry up to 10Gbps. The throughput for each file is graphed in Figure 7.3. This result was contrasted with the live network which averaged well over 10MB per second. This shows that the simulated network had limitations for representing a true network.

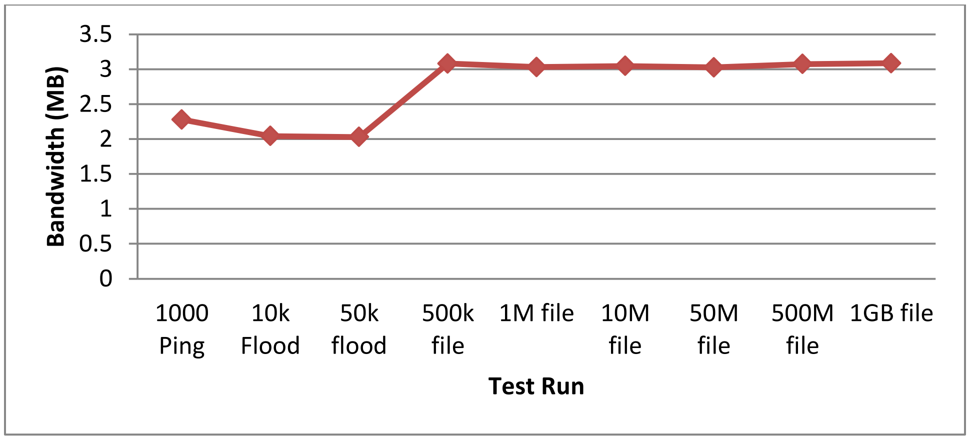


Figure 7.3 Average Throughput with No MT6D

MT6D enabled testing had a maximum average throughput of 2MB per second in both simulated and live network tests as displayed in Figures 7.4 and 7.5, respectively, showing that MT6D causes a bandwidth limitation regardless of the network throughput. This conclusion could not be made without testing MT6D on a different network, specifically in simulation.

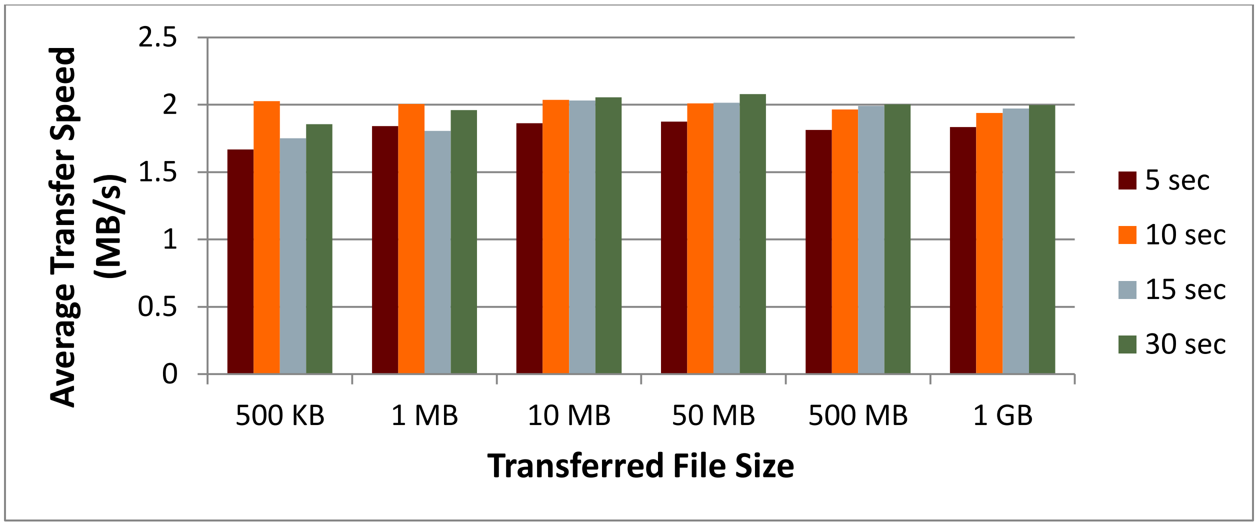


Figure 7.4 Transfer Speeds of Simulated MT6D One Hop File Tests

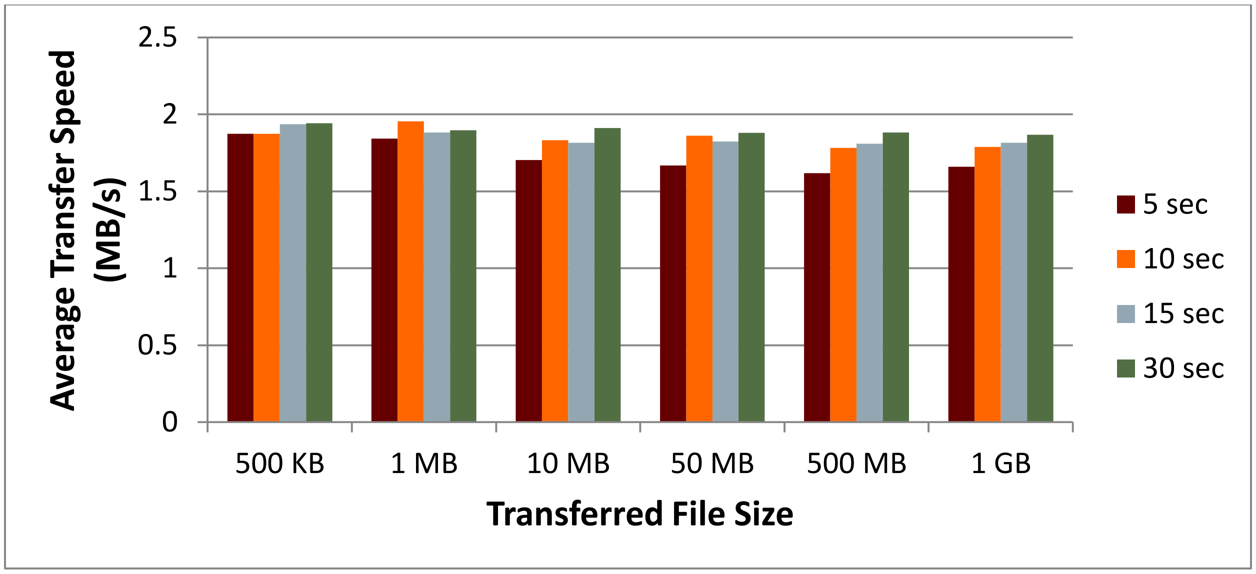


Figure 7.5 Transfer Speeds of Live Network MT6D File Tests

It is clear from these figures that the rotation time plays a part in the maximum speed of the communication transfer. To find the why this was the case, the retransmissions of each file transfer were examined. The retransmission data is an indication of the packets that are being dropped during communication which slows down the complete transfer. This data, shown in Figure 7.6, revealed that smaller rotation times resulted in a larger percentage of packets had to be retransmitted. The trend of higher retransmission rates at faster address rotation times appeared in both the live network and simulation tests. Using simulation helping to confirm that the cause of the retransmissions was within the MT6D application a not a cause of the network. This data was used to determine that MT6D has a limitation in how fast address rotations can happen. The limit is currently set at 10 seconds but there is ongoing work to optimize MT6D's code that could improve this limit [30].

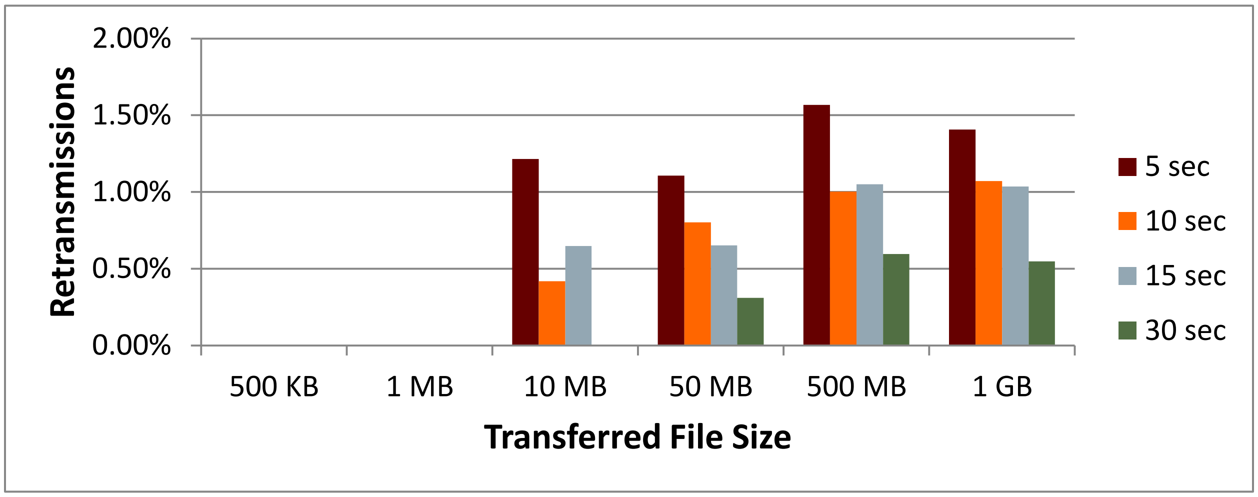


Figure 7.6 Retransmissions for Simulated MT6D One Hop Tests

These trends appear throughout the different simulation topologies. A lower throughput was noted for each hop added to the network.

An interesting statistic that appeared in the live network tests but not in simulation was that there was a higher packet loss shown in the 1,000 packet standard ping test. The standard ping had a packet loss of 10.16% in the live network test but only 0.16% loss in the simulated tests using a 10 second address rotation time. The discrepancy in the results did not occur in any of the ping flood tests. These results are graphed in Figure 7.7. After comparing packet traces of the live network tests to the simulated tests, the loss was determined to be coming from live network. The router in the live network dropped a NDP packet on each new address rotation [8]. Considering the rotation occurred every 10 seconds, this resulted in a 10% packet loss. The simulation helped to confirm this was a network configuration error and not a software bug within MT6D.

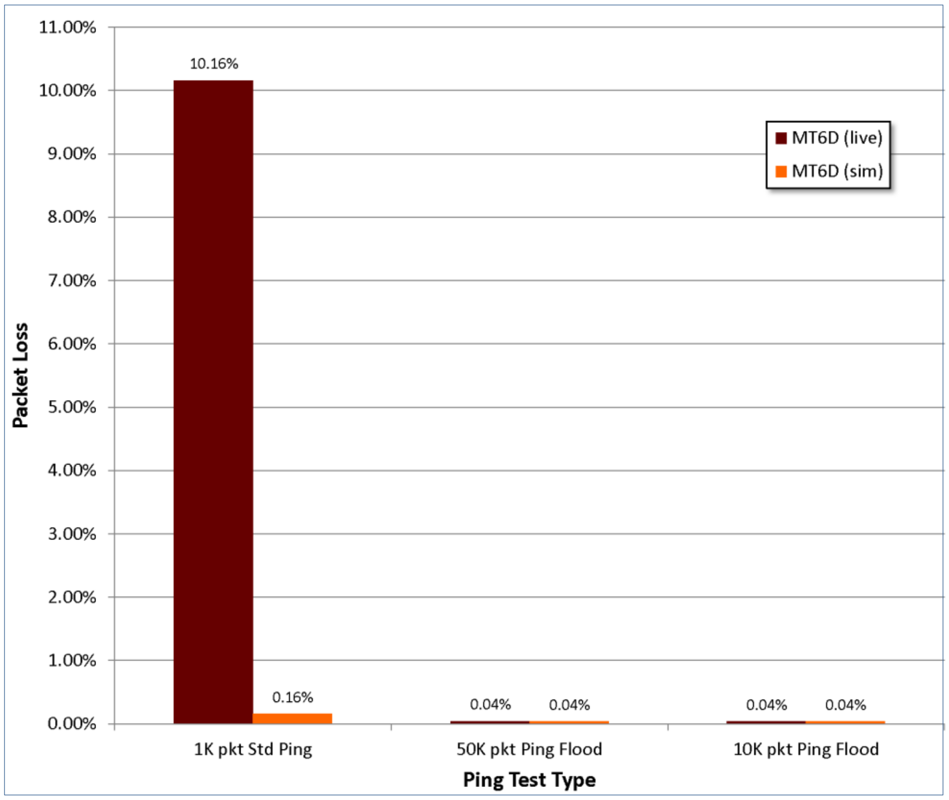


Figure 7.7 Packet Loss Comparisons between Live and Simulated Pings

Testing MT6D on the Campus Topologies with background traffic enabled ran into some unforeseen obstacles. On the first iteration of background traffic testing, the simulation crashed and not results were obtained. Continued research showed that this was due to improper configuration on the real devices. These devices had several network interface cards installed. For the simulation tests, all interfaces were disabled except for the one used to connect to the simulation. It was found that during this set of tests, one of the interfaces was enabled with an IPv4 address and was sending out packets. The packet filter on the SITL module was set for IPv6 only and so these IPv4 packets were not dropped cleanly causing the simulation crash.

Secondly, a threshold was discovered in how many nodes could be simulated while maintaining a real time communication flow with the live devices. The amount of simulated traffic also contributed to the number of nodes that could be simulated. The threshold of simulated nodes is dependent on the amount of memory available on the simulation machine. It was found that the campus topology could simulate the 120-user Floor scenario using background traffic without crashing. The throughput of a 1GB file transfer was 1.86MBps, which falls within the live network results range. .The upper limit of node simulation that allows real communication using SITL is 1800 nodes and was found through varying the amount of nodes in simulation monitoring for memory errors. This limit translates to one simulated building topology using the maximum amount of users per floor and shows that the amount of simulated nodes generating background traffic needs to be kept in the hundreds range for effective simulation.

Overall, the testing of MT6D on the simulation framework proved to be extremely useful. There were two programming issues found within the application code by using the simulation. The simulation also confirmed a configuration issue within the live network routers. Without simulating MT6D, these errors could have gone overlooked or been attributed to the influence of the live network. The series of different topologies helped to test the effects of multiple hops on the application, isolation from unrelated traffic, and background traffic itself. MT6D testing revealed a simulation limitation of OPNET when using SITL. These tests, in combination with the live network testing, proved that MT6D is a working IPv6 security application.

7.6 Chapter Conclusions

Chapter 7 described the methodology of using the developed IPv6 framework to test two applications: a VoIP application and MT6D, a custom IPv6 security application. The results from these tests helped to prove the validity of using the developed simulation framework to test IPv6 application. The simulations helped to refine the MT6D application by finding two software issues that were overlooked in live network testing. The work in this chapter is summarized in two publications: “Using Simulation to Evaluate a Custom IPv6 Application” and “Validating a Custom IPv6 Security Application using OPNET Modeler” [46] [47].

8. Conclusions and Future Work

The goal of this research was to create a simulation framework for testing IPv6 applications motivated from the need to simulate the custom IPv6 application MT6D. The availability of the Virginia Tech production IPv6 network provided the possibility to compare simulation results with live network results. Another motivation for this research is the timing of IPv6 deployment IPv6 in the world. The last IPv4 block of addresses was assigned in 2011 cementing the reasoning that IPv6 is needed to support the growing number of internet capable devices. The IPv6 simulation framework can help administrators plan for this change and test specific applications for functionality. The developed framework provides a set of scenarios and customizable traffic loads to help evaluate the applications for IPv6 readiness that will be used on a network.

OPNET Modeler was evaluated for IPv6 capabilities in addition to creating the framework. The results from this evaluation shows that the SITL module and Modeler's traffic generation is lacking complete IPv6 support despite advertising it. The results of the evaluation brought forth an interesting question: what is the meaning of being IPv6 capable? While this is not a new question, it became clear there is potential for miscommunication to the consumers of IPv6 products and devices. The complete results of the OPNET IPv6 evaluation are located in Table 6.1. During the evaluation, the Barrage Traffic benchmark test was introduced for evaluating latency and the resource usage between 32-bit and 64-bit architectures.

Two applications, one standard and one custom, were simulated on using the developed IPv6 simulation framework. The VoIP application, which represented a standard application, showed that the framework behaves as the live network does; latency increases with the introduction of background traffic. VoIP testing reiterated the findings of the SITL's IPv6 evaluation through its jitter results: SITL incurs a queuing and translation delay that must be accounted for when analyzing simulation results. MT6D, the custom application, showed the effectiveness of the framework at pinpointing software bugs in the application source code and verifying live device configuration errors. The framework confirmed that MT6D is a valid, functioning IPv6 application.

There are several areas of possible future work. First, a solution needs to be found for the traffic generation issues within Modeler because there are scenarios in which application independent traffic would be useful in simulation. Currently, only application traffic can be used in IPv6. Second, the SITL module needs to add support for all extension headers. A custom translation function was written to add support for the Destination Options extension header; however, errors were encountered so it was not fully functional. For SITL to claim IPv6 support, extension headers need be implemented and added to SITL's default translation function. Third, this framework could be adapted to support wireless communication. Currently, the framework simulates a wired network. Wireless devices are increasingly being used over wired ones which is particularly evident in mobile phones. An evaluation of IPv6 support in the wireless module in OPNET would need to be done as a part of making a wireless framework.

In conclusion, the need to adopt IPv6 in live networks is increasing every day. Applications need to be tested for IPv6 support before deployment on a live network and simulation is a good option to accomplish that need. An IPv6 simulation framework was presented in which applications can be tested in different scenarios. It is flexible for users to modify the simulated devices and traffic to fit their personal network characteristics. An evaluation of OPNET Modeler and the SITL module was also performed resulting in the conclusion that OPNET has much to do to improve their IPv6 support. Lastly, this work confirmed the validity of MT6D as an IPv6 custom application.

REFERENCES

[1] S. Deering and R. Hinden, “Internet Protocol, Version 6 (IPv6) Specification,” RFC 2460 (Draft Standard), Internet Engineering Task Force, Dec. 1998, updated by RFCs 5095, 5722. [Online]. Available: http://www.ietf.org/rfc/rfc2460.txt

[2] T. Narten, E. Nordmark, W. Simpson and H. Soliman, “Neighbor Discovery for IP version 6 (IPv6),” RFC 4861 (Draft Standard), Internet Engineering Task Force, Sept. 2007, updated by RFCs 5942. [Online]. Available: http://www.ietf.org/rfc/rfc4861.txt

[3] “OPNET Modeler,” http://www.opnet.com/solutions/network rd/modeler.htm.

[4] “OPNET System-in-the-Loop (SITL) module,” http://www.opnet.com/solutions/network_rd/system_in_the_loop.html.

[5] “Skype,” http://www.skype.com/intl/en-us/home.

[6] “magicJack,” http://www.magicjack.com/.

[7] “Measure VoIP Statistics,” Blue Coat. https://bto.bluecoat.com/packetguide/8.4/nav/tasks/measurement/voip-metrics.htm.

[8] M. Dunlop, “Achieving Security and Privacy in the Internet Protocol Version 6 Through the Use of Dynamically Obscured Addresses,” Ph.D. Dissertation, Dept. of Electr. And Comp. Eng., Virginia Tech, Blacksburg, VA, 2012.

[9] J. A. Zubairi and Mike Zuber, "SUNY Fredonia Campus Network Simulation and Performance Analysis Using OPNET" in Proc. online OPNETWORK2000, Washington D.C., Aug 2000.

[10] J. Potemans, J. Theunis, B. Rodiers, B. Van den Broeck, P. Leys, E. Van Lil and A. Van de Capelle, “Simulation of a Campus Backbone Network, a case-study”, OPNETWORK 2002, Washington D.C., August 2002.

[11] Hua Ning; , "IPv6 test-bed networks and R&D in China," Applications and the Internet Workshops, 2004. SAINT 2004 Workshops. 2004 International Symposium on, vol., no., pp. 105- 111, 26-30 Jan. 2004 doi: 10.1109/SAINTW.2004.1268573 URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1268573&isnumber=28387

[12] Hongwei Huo; Hongke Zhang; Yanchao Niu; Shuai Gao; Zhaohua Li; Sidong Zhang; , "MSRLab6: An IPv6 Wireless Sensor Networks Testbed," Signal Processing, 2006 8th International Conference on , vol.4, no., 16-20 Nov. 2006 doi: 10.1109/ICOSP.2006.346060
URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=4129752&isnumber=4129618

[13] Montavont, N.; Boutet, A.; Ropitault, T.; Tsukada, M.; Ernst, T.; Korva, J.; Viho, C.; Bokor, L.; , "Anemone: A ready-to-go testbed for IPv6 compliant intelligent transport systems," ITS Telecommunications, 2008. ITST 2008. 8th International Conference on, vol., no., pp.228-233, 24-24 Oct. 2008 doi: 10.1109/ITST.2008.4740262 URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=4740262&isnumber=4740207

[14] Y. Chang, H. Chao, J.Chen and R. Jhang, “Evaluation of IPv6 Simulation Using OPNET IPv6 Model Suite”, OPNETWORK 2003, Washington D.C., August 2003.

[15] D. Le, X. Fu and D. Hogrefe, “Evaluation of Mobile IPv6 Based on an OPNET Model,” Online. URL: http://user.informatik.uni-goettingen.de/~le/papers/ICYCS05.pdf.

[16] Md. Tariq Aziz, Mohammad Saiful Islam and Md. Nazmul Islam Khan. Article: Throughput Performance Evaluation of Video/Voice Traffic in IPv4/IPv6 Networks. International Journal of Computer Applications 35(2):5-12, December 2011. Published by Foundation of Computer Science, New York, USA

[17] D. Green, R. Mayo, and R. Reddy, "IPv6 application performance characterization using a virtual/live testbed," in Military Communications Conference, 2006. MILCOM 2006. IEEE, Oct. 2006, pp. 1 –4.

[18] "ns-3," Online. URL: http://www.nsnam.org/

[19] "OMNeT++," Online. URL: http://www.omnetpp.org/

[20] "Common Open Research Emulator (CORE)," Online. URL: http://cs.itd.nrl.navy.mil/work/core/

[21] "TTE4INET: TTEthernet Model for INET Framework," Online. URL: http://tte4inet.realmv6.org/

[22] J. Potemans, B. Van den Broeck, Y.Guan, J. Theunis, E. Van Lil and A. Van de Capelle, "Implementation of an Advanced Traffic Model in OPNET Modeler", OPNETWORK 2003, Washington D.C., August 2003.

[23] "Wireshark," Online. URL: http://www.wireshark.org/

[24] "GNU Wget," Online. URL: http://www.gnu.org/software/wget/

[25] "Scapy," Online. URL: http://www.secdev.org/projects/scapy/

[26] B. Clore, M. Dunlop, R. Marchany, J. Tront. "An Evaluation of IPv6 in Simulation Using OPNET Modeler," in Proc. of the 8th Advanced International Conference on Telecommunications, 2012, pp. 111-115.

[27] "Mumble," Online. URL: http://mumble.sourceforge.net/

[28] "Speex," Online. URL: http://www.speex.org/

[29] "Understanding Jitter in Packet Voiced Networks," Cisco. Online. URL: http://www.cisco.com/en/US/tech/tk652/tk698/technologies_tech_note09186a00800945df.shtml

[30] R. Moore. "Optimizing MT6D for the Stream Control Transfer Protocol," Online Wiki. URL: http://redmine.cirt.vt.edu/

[31] FCC 05-116. U.S. Federal Communications Commission. URL: http://transition.fcc.gov/cgb/voip911order.pdf

[32] "International Call Traffic Growth Slows as Skype's Volumes Soar ," Telegeography. 9 Jan. 2012. Online. URL: http://www.telegeography.com/press/press-releases/2012/01/09/international-call-traffic-growth-slows-as-skypes-volumes-soar/index.html

[33] "Customer Case Studies Network R&D," OPNET. Online. URL: http://www.opnet.com/customer_case_studies/network_rd/index.html

[34] "OPNET Modeler Documentation," OPNET. Online. URL: http://www.opnet.com/

[35] "OPNET University Program," OPNET. Online. URL: http://www.opnet.com/university_program/index.html

[36] "OPNET User Forums," OPNET. Online. URL: http://userforums.opnet.com/

[37] "ns-3 3.14 Documentation," ns-3. Online. URL: http://www.nsnam.org/docs/release/3.14/models/ns-3-model-library.pdf

[38] “NSOC2011 Accepted Projects,” ns-3 Wiki. Online. URL: http://www.nsnam.org/wiki/index.php/NSOC2011AcceptedProjects

[39] “ns-3: API and Model Change History,” ns-3. Online. URL: http://code.nsnam.org/ns-3.14/raw-file/4676959a1e99/CHANGES.html

[40] “INET Framework for OMNeT++,” OMNeT++. Online. URL: http://inet.omnetpp.org/doc/inet-manual-DRAFT.pdf

[41] VMWare. Online. URL: http://www.vmware.com/

[42] VirtualBox. Online. URL: https://www.virtualbox.org/

[43] Backtrack Linux. Online. URL: http://www.backtrack-linux.org/

[44] “Campus Residence Halls,” Virginia Tech. Online. URL: http://www.housing.vt.edu/halls/

[45] “MRTG Traffic and Performance Analysis,” Virginia Tech. Online. URL: http://mrtg.cns.vt.edu/

[46] B. Clore, M. Dunlop, R. Marchany, J. Tront, “Using Simulation to Evaluate a Custom IPv6 Application”, OPNETWORK 2012, Washington D.C., August 2012.

[47] B. Clore, M. Dunlop, R. Marchany, J. Tront, “Validating a Custom IPv6 Security Application using OPNET Modeler”, in Military Communications Conference, 2012. MILCOM 2012.

[48] “Internet Protocol” RFC 791 (Draft Standard), Internet Engineering Task Force, Sept. 1981. Online. Available: http://www.ietf.org/rfc/rfc791.txt

[49] A. Conta, S. Deering, M. Gupta, “Internet Control Message Protocol (ICMPv6) for the Internet Protocol Version 6 (IPv6) Specification” RFC 4443 (Draft Standard), Internet Engineering Task Force, Mar. 2006. Online. Available: http://www.ietf.org/rfc/rfc4443.txt

[50] J. Postel, “Internet Control Message Protocol” RFC 792 (Draft Standard), Internet Engineering Task Force, Sept. 1981. Online. Available: http://www.ietf.org/rfc/rfc792.txt

[51] C. Perkins, D. Johnson and J. Arkko. “Mobility Support in IPv6” RFC 6275 (Draft Standard), Internet Engineering Task Force, July 2011. Online. Available: http://www.ietf.org/rfc/rfc6275.txt

[52] R. Hinden and S. Deering. “IP Version 6 Addressing Architecture” RFC 4291 (Draft Standard), Internet Engineering Task Force, Feb 2006. Online. Available: http://www.ietf.org/rfc/rfc4291.txt

[53] B. Carpenter, K. Moore, and B. Fink, “Connecting IPv6 Routing Domains Over the IPv4 Internet,” Cisco. Online. URL: http://www.cisco.com/web/about/ac123/ac147/ac174/ac197/about_cisco_ipj_archive_article09186a00800c830a.html

[54] R. Fink and R. Hinden. “6bone (IPv6 Testing Address Allocation) Phaseout” RFC 3701 (Draft Standard), Internet Engineering Task Force, Mar 2004. Online. Available: http://www.ietf.org/rfc/rfc3701.txt

[55] “Request Resources,” ARIN. Online. URL: https://www.arin.net/resources/request.html

[56] S. Kent and R. Atkinson. “Security Architecture for the Internet Protocol” RFC 2401 (Draft Standard), Internet Engineering Task Force, Nov 1998. Online. Available: http://www.ietf.org/rfc/rfc2401.txt

[57] “ICMPv6 Parameters,” IANA. Online. URL: http://www.iana.org/assignments/icmpv6-parameters/icmpv6-parameters.xml

[58] J. Rosenberg et al. “SIP: Session Initiation Protocol” RFC 3261 (Draft Standard), Internet Engineering Task Force, Jun 2002. Online. Available: http://www.ietf.org/rfc/rfc3261.txt

APPENDIX

APPENDIX A: IPV6 TRAFFIC GENERATOR PROCESS FUNCTION CODE

```
//Main code from the traf_gen process provided by OPNET
//Changes were made to modify the address reference to an accepted IPv6 format
static void
ip_traf_gen_dispatcher_sv_init (void)
	{
	/** Initializes all state variables used in this process model. **/
	FIN (ip_traf_gen_dispatcher_sv_init ());
	/* Obtain the object identifiers for the surrounding module, */
	/* node and subnet.
		*/
	my_objid		= op_id_self ();
	my_node_objid	= op_topo_parent (my_objid);
	my_subnet_objid	= op_topo_parent (my_node_objid);

	/* Create the ici that will be associated with packets being	*/
	/* sent to IP.
		*/
	ip_encap_req_ici_ptr = op_ici_create ("inet_encap_req"); // this is a change from the
IPv4 version

	traf_delay = op_stat_reg ("IP Traffic Generator.Delay (secs)",
OPC_STAT_INDEX_NONE, OPC_STAT_LOCAL);
	traf_sent_bps = op_stat_reg ("IP Traffic Generator.Traffic Sent (bits/sec)",
OPC_STAT_INDEX_NONE, OPC_STAT_LOCAL);
	traf_sent_pps = op_stat_reg ("IP Traffic Generator.Traffic Sent (packets/sec)",
OPC_STAT_INDEX_NONE, OPC_STAT_LOCAL);
```

```
	traf_rcvd_bps = op_stat_reg ("IP Traffic Generator.Traffic Received (bits/sec)", 
OPC_STAT_INDEX_NONE, OPC_STAT_LOCAL);
	traf_rcvd_pps = op_stat_reg ("IP Traffic Generator.Traffic Received (packets/sec)", 
OPC_STAT_INDEX_NONE, OPC_STAT_LOCAL);


	FOUT;
	}
static void ip_traf_gen_dispatcher_register_self (void)
	{
	char				proc_model_name [128];
	OmsT_Pr_Handle		own_process_record_handle;
	Prohandle			own_prohandle;

	/** Get a higher layer protocol ID from IP and register this  **/
	/** process in the model-wide process registry to be discovered **/
	/** by the lower layer.
		**/
	FIN (ip_traf_gen_dispatcher_register_self ());

	/* Register RPG as a higher layer protocol over IP layer			*/
	/* and retrieve an auto-assigned protocol id.					*/
	higher_layer_proto_id = IpC_Protocol_Unspec;
	Ip_Higher_Layer_Protocol_Register ("ip_traf_gen", &higher_layer_proto_id);

	/* Obtain the process model name and process handle.				*/
	op_ima_obj_attr_get (my_objid, "process model", proc_model_name);
	own_prohandle = op_pro_self ();

	/* Register this process in the model-wide process registry		*/
	own_process_record_handle = (OmsT_Pr_Handle) oms_pr_process_register
```

```
		(my_node_objid, my_objid, own_prohandle, proc_model_name);

	/* Set the protocol attribute to the same string we used in	*/
	/* Ip_Higher_Layer_Protocol_Register. Necessary for ip_encap*/
	/* to discover this process. Set the module object id also.	*/
	oms_pr_attr_set (own_process_record_handle,
				"protocol",		OMSC_PR_STRING, "ip_traf_gen",
				"module objid",	OMSC_PR_OBJID,	my_objid,
				OPC_NIL);
	FOUT;
	}

static void ip_traf_gen_packet_flow_info_read (void)
	{
	int					i, row_count;
	char			temp_str[INETC_ADDR_STR_LEN]; // this is a change from the
IPv4 version
	Objid			trafgen_params_comp_objid;
	Objid			ith_flow_attr_objid;
	double			start_time;
	char			dist_name [512];
	//unsigned char family;
	/** Read in the RPG Start Time simulation attribute and the packet flow	**/
	/** configuration attributes.
		**/

	FIN (ip_traf_gen_packet_flow_info_read (void));

	/* Get a handle to the Traffic Generation Parameters compound attribute.*/
	op_ima_obj_attr_get (my_objid, "Traffic Generation Parameters",
&trafgen_params_comp_objid);
```

```
        /* Obtain the row count
                        */
        row_count = op_topo_child_count (trafgen_params_comp_objid,
OPC_OBJTYPE_GENERIC);

        /* If there are no flows specified, exit from the function                    */
        if (row_count == 0)
                {
                ip_traf_gen_flow_info_array = OPC_NIL;

                FOUT;
                }

        /* Allocate enough memory to hold all the information.
        */
        ip_traf_gen_flow_info_array = (IpT_Traf_Gen_Flow_Info*) op_prg_mem_alloc (sizeof
(IpT_Traf_Gen_Flow_Info) * row_count);

        /* Loop through each row and read in the information specified.               */
        for (i = 0; i < row_count; i++)
                {
                /* Get the object ID of the associated row for the ith child.         */
                ith_flow_attr_objid = op_topo_child (trafgen_params_comp_objid,
OPC_OBJTYPE_GENERIC, i);

                /* Read in the start time.
                        */
                op_ima_obj_attr_get (ith_flow_attr_objid, "Start Time", &start_time);

                /* Schedule an interrupt for the start time of this flow.             */
```

```
			/* Use the row index as the interrupt code, so that we can handle	*/
			/* the interrupt correctly.
				*/
			op_intrpt_schedule_self (start_time, i);

			/* Read in the packet inter-arrival time and packet size				*/
			op_ima_obj_attr_get (ith_flow_attr_objid, "Packet Inter-Arrival Time",
dist_name);
			ip_traf_gen_flow_info_array[i].packet_interarrival_time =
oms_dist_load_from_string (dist_name);

			op_ima_obj_attr_get (ith_flow_attr_objid, "Packet Size", dist_name);

			ip_traf_gen_flow_info_array[i].packet_size = oms_dist_load_from_string
(dist_name);

			/* Read in the destination IP address.
	*/
			op_ima_obj_attr_get (ith_flow_attr_objid, "Destination IP Address", temp_str);

			//CHANGE!!

			ip_traf_gen_flow_info_array[i].dest_address = inet_address_create (temp_str,
InetC_Addr_Family_v6); // this is a change from the IPv4 version



			//END CHANGE

			/* Read the type-of-service */
			op_ima_obj_attr_get (ith_flow_attr_objid, "Type of Service",
				&ip_traf_gen_flow_info_array[i].tos);
```

```
		}

	FOUT;
	}

static void ip_traf_gen_generate_packet (void)
	{
	int			row_num;
	Packet		*pkt_ptr;
	char		dest_address_str [INETC_ADDR_STR_LEN]; // this is a change from the
IPv4 version
	double		pk_size;

	/** A packet needs to be generated for a particular flow	**/
	/** Generate the packet of an appropriate size and send it	**/
	/** to IP. Also schedule an event for the next packet		**/
	/** generation time for this flow.					**/

	FIN (ip_traf_gen_generate_packet ());

	/* Identify the right packet flow using the interrupt code	*/
	row_num = op_intrpt_code ();

	/* Schedule a self interrupt for the next packet generation	*/
	/* time. The netx packet generation time will be the current*/
	/* time + the packet inter-arrival time. The interrupt code	*/
	/* will be the row number.
	*/
	op_intrpt_schedule_self
		(op_sim_time () +
                              oms_dist_outcome
(ip_traf_gen_flow_info_array[row_num].packet_interarrival_time), row_num);

        /* Create an unformatted packet. The packet size should be */
        /* the specified packet size. Multiply the value by 8 to    */
        /* convert it into bits.                                                        */
        pk_size = 8 * (int) oms_dist_outcome
(ip_traf_gen_flow_info_array[row_num].packet_size);
        pkt_ptr = op_pk_create (pk_size);

        op_stat_write (traf_sent_bps, pk_size);
        op_stat_write (traf_sent_bps, 0.0);
        op_stat_write (traf_sent_pps, 1.0);
        op_stat_write (traf_sent_pps, 0.0);

        /* Print out a trace message.                                                   */
        if (LTRACE_RPG_ACTIVE)
                {
                inet_address_print (dest_address_str,
ip_traf_gen_flow_info_array[row_num].dest_address); // this is a change from the IPv4 version
                op_prg_odb_print_major ("Sending a packet to the address",dest_address_str,
OPC_NIL);
                }

        //printf("address in traf_gen: %s\n", dest_address_str);

        /* Set the destination address in the ici.                                  */
        op_ici_attr_set (ip_encap_req_ici_ptr, "dest_addr",
ip_traf_gen_flow_info_array[row_num].dest_address);
```

```
	op_ici_attr_set (ip_encap_req_ici_ptr, "Type of Service", 
ip_traf_gen_flow_info_array[row_num].tos);

	//attempts were made to manually set the version of the ICI but this crashed the 
simulation



	/* Install the ici
	*/
	op_ici_install (ip_encap_req_ici_ptr);

	/* Send the packet
	*/
	/* Since we are reusing the ici we should use                */
	/* op_pk_send_forced. Otherwise if two flows generate a    */
	/* packet at the same time, the second packet genration will*/
	/* overwrite the ici before the first packet is processed by*/
	/* ip_encap.
	*/
	op_pk_send_forced (pkt_ptr, 0);

	/* Uninstall the ici.
	*/
	op_ici_install (OPC_NIL);

	FOUT;
	}

static void ip_traf_gen_packet_destroy (void)
	{
	Packet*				packet_ptr;
	Ici*			ip_encap_ind_ici_ptr;
	char			src_address_str [INETC_ADDR_STR_LEN];
	IpT_Address		src_address;
	double			pk_size;

	/** Get a packet from IP and destroy it. Destroy	**/
	/** the accompanying ici also.				**/

	FIN (ip_traf_gen_packet_destroy (void));

	/* Get the ip_encap_ind_v4 accompanying the packet	*/
	ip_encap_ind_ici_ptr = op_intrpt_ici ();

	/* Print a trace message				*/
	if (LTRACE_RPG_ACTIVE)
		{
		/* Get the source address from the ici.		*/
		op_ici_attr_get (ip_encap_ind_ici_ptr, "src_addr", &src_address);
		//ip_address_print (src_address_str, src_address);
		//op_prg_odb_print_major ("Received a packet from",	src_address_str,
OPC_NIL);
		}

	/* Destroy the ici.					*/
	op_ici_destroy (ip_encap_ind_ici_ptr);

	/* Get the packet and destroy it			*/
	packet_ptr = op_pk_get (0);
	pk_size = (double) op_pk_total_size_get (packet_ptr);

	op_stat_write (traf_rcvd_bps, pk_size);
op_stat_write (traf_rcvd_bps, 0.0);
op_stat_write (traf_rcvd_pps, 1.0);
op_stat_write (traf_rcvd_pps, 0.0);
op_stat_write (traf_delay, op_sim_time () - op_pk_creation_time_get (packet_ptr));

op_pk_destroy (packet_ptr);

FOUT;
}
```

APPENDIX B: SITL CUSTOM TRANSLATION FUNCTION

```
//This file is a translation function for extension headers in general. Considering that my tests are
passing
//through the simulation and have little interaciton with the other nodes, the options and
information inside do
// not have to be translated at this time.
//Use the SITL example for RTP to model this after
//translate first 40 bytes

#include <opnet.h>
#include <ip_dgram_sup.h>
#include <ethernet_support.h>
#include <ip_addr_v4.h>



#include "winsock.h"
#include "sitl_packet_translation.h"

// Extension header items that I care about are the next header and length, the rest will be
encapsulated
typedef struct SitlT_EH_Header {
        OpT_uInt8     Next_Header;
        OpT_uInt8     Header_Length;
        } SitlT_EH_Header;

typedef struct IPv6_Small_Header {
        OpT_uInt32    ver_traf_flow;
        OpT_uInt16    payload_length;
        OpT_uInt8     next_header;
        OpT_uInt8     hop_limit;
        } IPv6_Small_Header;

typedef struct Ipv6T_Address //originally in ip_addr_v4.c
        {
        OpT_uInt32                      addr32 [4];
        } Ipv6T_Address;

typedef struct Ipv6T_Address_bytes //originally in ip_addr_v4.c
        {
        OpT_uInt8                       addr8 [16];
        } Ipv6T_Address_bytes;

typedef struct IPv6_40_Header { //have to figure out the address situation
        OpT_uInt32    ver_traf_flow; //contains the version, traffic class and flow
        OpT_uInt16    payload_length;
        OpT_uInt8     next_header;
        OpT_uInt8     hop_limit;
        Ipv6T_Address source;
        Ipv6T_Address dest;

        } IPv6_40_Header;

static Pmohandle                        Ipv6I_Addr_Pmh;



// I'm saying the the extension header is only 2 bytes
#define SITL_EH_LENGTH 2
#define IPv6_LENGTH 40
//sounds like I'm canadian
int sitl_test_from_real_to_simulated_eh (SitlT_SCDB* scdb_ptr);
int sitl_test_to_real_to_simulated_eh (SitlT_SCDB* scdb_ptr);
int sitl_translate_from_real_to_simulated_eh (SitlT_SCDB* scdb_ptr);
int sitl_translate_from_simulated_to_real_eh(SitlT_SCDB* scdb_ptr);

//test for the next header
#define DEST_OPTION 60
#define AH 51
#define ESP 50
//#define RTP_PORT_NUMBER 6980

// The three functions declared DLLEXPORT below need to be set as attributes
// on the SITL node where the conversion needs to take place

// Initialization function registers the EH translation functions with SITL
DLLEXPORT int sitl_translation_init_eh(void)
	{
	FIN (sitl_translation_initialization_eh ());
	op_pk_sitl_packet_translation_init();
	op_pk_sitl_register_translation_function (
		SITL_TRANSLATION_DIRECTION_REAL_TO_SIM,		// direction
		sitl_translate_from_real_to_simulated_eh,	// translation function
		sitl_test_from_real_to_simulated_eh,		// test function
		"ethernet_v2",						// base packet
format
		150);								//
priority
	op_pk_sitl_register_translation_function (
		SITL_TRANSLATION_DIRECTION_SIM_TO_REAL,		// direction
		sitl_translate_from_simulated_to_real_eh,	// translation function
		sitl_test_to_real_to_simulated_eh,		// test function
		"ethernet_v2",						// base packet
format
```

```
                    100);                                                                                                    //
priority
          printf("I'm initializing! ");
          FRET (1);
          }

// The following two translation functions delegate the work
// to the standard SITL translation functions
// They are included here as an example. Alternatively, you may
// use the Opnet kp's directly.

//keeping this the same
DLLEXPORT int sitl_from_real_all_supported_eh (SitlT_SCDB* scdb_ptr)
          {
          int result, test;

          FIN (sitl_from_real_all_supported_eh (scdb_ptr));

          if (scdb_ptr == OPC_NIL)
                    FRET (0);

          result = op_pk_sitl_from_real_all_supported (scdb_ptr);
          printf("   to sim all supported - finished");
          FRET(result);
          }

DLLEXPORT int sitl_to_real_all_supported_eh (SitlT_SCDB* scdb_ptr)
          {
          int result;
          FIN (sitl_to_real_all_supported_eh(scdb_ptr));
          result = op_pk_sitl_to_real_all_supported (scdb_ptr);
```

```
        FRET (result);
        }

// Test function - if the next header is an extension header, pick me! - real to sim
int sitl_test_from_real_to_simulated_eh (SitlT_SCDB* scdb_ptr)
        {

        IPv6_Small_Header* v6small_ptr;
        IpT_Dgram_Fields* ip_fields = OPC_NIL;
        IPv6_40_Header* v640_ptr;
        Packet* ip_packet = OPC_NIL;
        InetT_Address test;
        int ipv6test, outtest, ret;
        OpT_uInt8*    icmp_type;
        char* print[256];
        OpT_Packet_Id id;

        FIN (sitl_test_from_real_to_simulated_eh (scdb_ptr));
        printf("\nIN:  ");
        //printf("  current pointer offset is %i", scdb_ptr->real_pk_offset);
        //printf("  real pk size is %i", scdb_ptr->real_pk_size);

        //op_pk_format(scdb_ptr->sim_pk_ptr, print);
        //printf("  format of the sim packet = %s ", print);
        ipv6test = op_pk_sitl_test_from_real_ipv6_ip(scdb_ptr);
        outtest = op_pk_sitl_test_to_real_ipv6_ip(scdb_ptr);
        //printf("     ipv6test = %i  and outtest = %i ", ipv6test,outtest);
        if(ipv6test)
                {
```

```
            v6small_ptr = (IPv6_Small_Header*)(scdb_ptr->real_pk_ptr + scdb_ptr->
            real_pk_offset);

            //op_pk_nfd_get_pkt(scdb_ptr->sim_pk_ptr, "data", &ip_packet); //this causes
            problems
            //ip_fields = (IpT_Dgram_Fields*)(ip_packet);

            id = op_pk_id(scdb_ptr->sim_pk_ptr);
            printf(" eth id: = %i ", id);

            v640_ptr = (IPv6_40_Header*)(scdb_ptr->real_pk_ptr + scdb_ptr-
            >real_pk_offset);
            printf("real next header =  %i", v6small_ptr->next_header);

            //printf("   sim next header = %i", ip_fields->protocol);
            //printf(" source address =  %x %x %x %x", v640_ptr-
            >source.addr32[0],v640_ptr->source.addr32[1],v640_ptr-
            >source.addr32[2],v640_ptr->source.addr32[3]);

            //v6small_ptr->next_header == DEST_OPTION
            FRET(v6small_ptr->next_header == DEST_OPTION); //v6small_ptr-
            >next_header == DEST_OPTION

            }
        else
          {
              FRET(0);
           }
        }

// Test function - sim to real (maybe just return true, get the 40 and leave it?)
```

```
int sitl_test_to_real_to_simulated_eh (SitlT_SCDB* scdb_ptr)
        {

        IPv6_Small_Header* v6small_ptr;
        IpT_Dgram_Fields* ip_fields = OPC_NIL;
        IPv6_40_Header* v640_ptr;
        EthT_Frame_Fields* eth_fields = OPC_NIL;
        Packet* eth_packet = OPC_NIL;
        Packet* ip_packet = OPC_NIL;
        Packet* ip_copy = OPC_NIL;
        Packet* data_copy = OPC_NIL;
        InetT_Address test;
        int ipv6test, outtest, next, ret;
        OpT_Packet_Id id;
        OpT_Packet_Size size;
        OpT_Packet_Size limit = 950;
        char* print[256];



        FIN (sitl_test_from_real_to_simulated_eh (scdb_ptr));
        printf("\nOUT:  ");
        //printf("  current pointer offset is %i", scdb_ptr->real_pk_offset);
        //printf("  real pk size is %i", scdb_ptr->real_pk_size);

        //op_pk_format(scdb_ptr->sim_pk_ptr, print);
        //printf("  format of the sim packet = %s ", print);
        ipv6test = op_pk_sitl_test_from_real_ipv6_ip(scdb_ptr);
        outtest = op_pk_sitl_test_to_real_ipv6_ip(scdb_ptr);
        printf(" ipv6test = %i ", ipv6test);
        if(ipv6test)
                {
```

```
        //v6small_ptr = (IPv6_Small_Header*)(scdb_ptr->real_pk_ptr + scdb_ptr-
        >real_pk_offset);
        printf(" out test start!");

        eth_packet = scdb_ptr->sim_pk_ptr;
        op_pk_nfd_get_pkt(eth_packet, "data", &ip_packet); //this strips out the
        packet..no good.
        //op_pk_format(ip_packet, print);
        size = op_pk_total_size_get(ip_packet);
        id = op_pk_id (ip_packet);
// printf("  format of the sim packet = %s ", print);
        printf("  id: = %i ", id);
        printf("  size: = %i ", size);

        if (ip_packet == OPC_NIL)
        {
        printf("bad packet");
        FRET (0);}

 //op_pk_nfd_get(ip_packet, "fields", &ip_fields);
        printf(" before copy!");
        //ip_copy = op_pk_copy(ip_packet);

        //op_pk_nfd_get_pkt(ip_copy, "data", &data_copy);
//size = op_pk_total_size_get(data_copy);
        //printf(" data size: = %i ", size);

        if(size > limit)
        {
        printf(" yes size is bigger that");
```

```
            ret = 1;
            }
            else
                  {
                  printf(" small packet");
                  ret = 0;
                  }

            //op_pk_destroy(ip_copy);

            /*printf(" copy!");
            op_pk_nfd_get(ip_copy, "fields", &ip_fields);

        printf("sim next header = %i", ip_fields->protocol);
            op_pk_destroy(ip_copy);
            next = ip_fields->protocol;
            //next=0;*/

            //op_pk_nfd_set(ip_packet, "fields", ip_fields);
            op_pk_nfd_set_pkt (eth_packet, "data", ip_packet); //reset
            printf("  finish out test");

            FRET(ret); //translate all ipv6!  next== DEST_OPTION

            }
      else
            {

        FRET(0);
        }
        }




// This function implements the translation of an EH packet from real to simulated (encapsulating
as one packet
int sitl_translate_from_real_to_simulated_eh (SitlT_SCDB* scdb_ptr)
        {
        IPv6_40_Header* eh_hdr_ptr = OPC_NIL;
        IpT_Dgram_Fields* ip_fields = OPC_NIL;
        Packet* eh_sim_ptr = OPC_NIL;
        Packet* eh_sim_data_ptr = OPC_NIL;
        unsigned char* data_ptr = OPC_NIL;

        int result1,traffic_class, data_size;
        InetT_Address source, dest;
        char* print[256];
        Ipv6T_Address* source_words;
        Ipv6T_Address* dest_words;
        int i =0;
        FIN (translate_from_real_to_simulated_eh (scdb_ptr));
        printf("  translation in ");
        // Get pointer to real packet

        //result1 = op_pk_sitl_translate_from_real_ipv6_ip(scdb_ptr);
        //op_pk_format(scdb_ptr->sim_pk_ptr, print);
        //printf("  format of the sim packet = %s ", print);
        //op_pk_nfd_get (scdb_ptr->sim_pk_ptr, "fields", &ip_fields);
        //printf("ident = %i, ttl= %i, protocol= %i  ", ip_fields->ident,ip_fields->ttl, ip_fields-
        >protocol);

        eh_hdr_ptr = (IPv6_40_Header*)(scdb_ptr->real_pk_ptr + scdb_ptr->real_pk_offset);


        ip_fields= ip_dgram_fdstruct_create(); //(IpT_Dgram_Fields*)(scdb_ptr->real_pk_ptr +
        scdb_ptr->real_pk_offset);

//the bytes are swapped - correction follows
        eh_hdr_ptr->source.addr32[0] =
 (((eh_hdr_ptr->source.addr32[0] & 0xFF000000) >>24) |
((eh_hdr_ptr->source.addr32[0] & 0x00FF0000) >>8)  |
((eh_hdr_ptr->source.addr32[0] & 0x0000FF00) <<8)  |
((eh_hdr_ptr->source.addr32[0] & 0x000000FF) <<24) );

eh_hdr_ptr->source.addr32[1] =
(((eh_hdr_ptr->source.addr32[1] & 0xFF000000) >>24) |
((eh_hdr_ptr->source.addr32[1] & 0x00FF0000) >>8)  |
((eh_hdr_ptr->source.addr32[1] & 0x0000FF00) <<8)  |
((eh_hdr_ptr->source.addr32[1] & 0x000000FF) <<24) );

eh_hdr_ptr->source.addr32[2] =
(((eh_hdr_ptr->source.addr32[2] & 0xFF000000) >>24) |
((eh_hdr_ptr->source.addr32[2] & 0x00FF0000) >>8)  |
((eh_hdr_ptr->source.addr32[2] & 0x0000FF00) <<8)  |
((eh_hdr_ptr->source.addr32[2] & 0x000000FF) <<24) );

eh_hdr_ptr->source.addr32[3] =
(((eh_hdr_ptr->source.addr32[3] & 0xFF000000) >>24) |
((eh_hdr_ptr->source.addr32[3] & 0x00FF0000) >>8)  |
((eh_hdr_ptr->source.addr32[3] & 0x0000FF00) <<8)  |
((eh_hdr_ptr->source.addr32[3] & 0x000000FF) <<24) );

eh_hdr_ptr->dest.addr32[0] =
(((eh_hdr_ptr->dest.addr32[0] & 0xFF000000) >>24) |
((eh_hdr_ptr->dest.addr32[0] & 0x00FF0000) >>8)  |
((eh_hdr_ptr->dest.addr32[0] & 0x0000FF00) <<8)  |
((eh_hdr_ptr->dest.addr32[0] & 0x000000FF) <<24) );

eh_hdr_ptr->dest.addr32[1] =
(((eh_hdr_ptr->dest.addr32[1] & 0xFF000000) >>24) |
((eh_hdr_ptr->dest.addr32[1] & 0x00FF0000) >>8)  |
((eh_hdr_ptr->dest.addr32[1] & 0x0000FF00) <<8)  |
((eh_hdr_ptr->dest.addr32[1] & 0x000000FF) <<24) );

eh_hdr_ptr->dest.addr32[2] =
(((eh_hdr_ptr->dest.addr32[2] & 0xFF000000) >>24) |
((eh_hdr_ptr->dest.addr32[2] & 0x00FF0000) >>8)  |
((eh_hdr_ptr->dest.addr32[2] & 0x0000FF00) <<8)  |
((eh_hdr_ptr->dest.addr32[2] & 0x000000FF) <<24) );

eh_hdr_ptr->dest.addr32[3] =
(((eh_hdr_ptr->dest.addr32[3] & 0xFF000000) >>24) |
((eh_hdr_ptr->dest.addr32[3] & 0x00FF0000) >>8)  |
((eh_hdr_ptr->dest.addr32[3] & 0x0000FF00) <<8)  |
((eh_hdr_ptr->dest.addr32[3] & 0x000000FF) <<24) );

//printf ("address = %x %x %x %x ", eh_hdr_ptr->source.addr32[0], eh_hdr_ptr-
>source.addr32[1],eh_hdr_ptr->source.addr32[2],eh_hdr_ptr->source.addr32[3]);
```

```
source.addr_family = InetC_Addr_Family_v6;
source.address.ipv6_addr_ptr =&eh_hdr_ptr->source;//source_words;
dest.addr_family = InetC_Addr_Family_v6;
dest.address.ipv6_addr_ptr =&eh_hdr_ptr->dest; //dest_words;

// Create simulated EH packet
// Get fields from real packet and set them in the simulated packet

traffic_class = (eh_hdr_ptr->ver_traf_flow & 0x0FF00000) >> 5;

//printf("  about to set fields ");
//OpT_Packet_Size                orig_len;
ip_fields->ident = 0;

//printf(" ident done  ");

ip_fields->frag_len = eh_hdr_ptr->payload_length;
//printf(" payload set ");

ip_fields->ttl = eh_hdr_ptr->hop_limit;

ip_fields->src_addr = source;
ip_fields->dest_addr = dest;

//printf("family = %i", inet_address_family_get(&ip_fields.dest_addr));

ip_fields->protocol = eh_hdr_ptr->next_header;

ip_fields->tos = traffic_class;
//ip_fields->offset = 0;
```

```
//printf("   about to op pk fields ");

eh_sim_ptr = op_pk_create_fmt ("ip_dgram_v4");

ip_dgram_fields_set (eh_sim_ptr,ip_fields);
op_pk_nfd_set(eh_sim_ptr, "fields", ip_fields);

// Advance the real packet read pointer to get to the data (no data field in the ip packet...?
scdb_ptr->real_pk_offset += IPv6_LENGTH;

data_size = scdb_ptr->real_pk_size - scdb_ptr->real_pk_offset;
data_ptr = op_prg_mem_alloc (data_size);
op_prg_mem_copy (scdb_ptr->real_pk_ptr + scdb_ptr->real_pk_offset, data_ptr,
data_size);

eh_sim_data_ptr = op_pk_create (0);
op_pk_fd_set_ptr (eh_sim_data_ptr, 0, data_ptr, data_size*8, op_prg_mem_copy_create,
op_prg_mem_free, data_size);
op_pk_nfd_set (eh_sim_ptr, "data", eh_sim_data_ptr);

op_pk_bulk_size_set (eh_sim_ptr, data_size * 8); //bulk size
op_pk_total_size_set (eh_sim_ptr, (data_size+40)*8); //total size

scdb_ptr->real_pk_offset += data_size;
//*/
//printf("   move the pointer ");
// Save newly converted simulated packet
scdb_ptr->sim_pk_ptr = eh_sim_ptr;

printf("   and end ");
```

```
	/*(*****************************/
	/* testing out ethernet only **/
	/*****************************/
	/* doesnt work because the second half of the NDP doesnt go through - it does translate
in well though

	data_size = scdb_ptr->real_pk_size - scdb_ptr->real_pk_offset; //the rest of the packet
	data_ptr = op_prg_mem_alloc (data_size);
	op_prg_mem_copy (scdb_ptr->real_pk_ptr + scdb_ptr->real_pk_offset, data_ptr,
data_size);

	eh_sim_data_ptr = op_pk_create (0);
	op_pk_fd_set_ptr (eh_sim_data_ptr, 0, data_ptr, data_size*8, op_prg_mem_copy_create,
op_prg_mem_free, data_size);

		// Advance the real packet read pointer
	scdb_ptr->real_pk_offset += data_size;

	// Save newly converted simulated packet
	scdb_ptr->sim_pk_ptr = eh_sim_data_ptr;*/



	// Return success indication
	FRET (1);
	}

// This function implements the translation of an RTP packet from simulated to real
int sitl_translate_from_simulated_to_real_eh (SitlT_SCDB* scdb_ptr)
	{
```

```
	IPv6_40_Header* eh_hdr_ptr = OPC_NIL;
	Packet* ip_dgram_pkptr;
	Packet* ip_packet;
	Packet* eh_sim_ptr = OPC_NIL;
	Packet* eh_sim_data_ptr = OPC_NIL;
	IpT_Dgram_Fields* ip_fields = OPC_NIL;
	Packet* eth_packet = OPC_NIL;
	unsigned char* data_ptr;
	int next_header, header_length, data_size;

	FIN (translate_from_simulated_to_real_eh (scdb_ptr));
	printf("outbound");
	// Get pointer to begining of the real packet memory area
	eh_hdr_ptr = (IPv6_40_Header*)(scdb_ptr->real_pk_ptr + scdb_ptr->real_pk_offset);

	// Get simulated UDP, and RTP packet pointers
	//ip_dgram_pkptr = scdb_ptr->sim_pk_ptr;
	//op_pk_nfd_get_pkt (eth_packet, "data", &eh_sim_ptr);

	eth_packet = scdb_ptr->sim_pk_ptr; //ethernet packet?
	//op_pk_nfd_get(eth_packet, "fields", &eth_fields);
	op_pk_nfd_get_pkt(eth_packet, "data", &ip_packet); //this strips out the packet..no good.

	op_pk_nfd_get(ip_packet, "fields", &ip_fields);
	//ip_fields = (IpT_Dgram_Fields*)(ip_packet);
		//printf(" eth type = %i", eth_fields->type);
	printf(" ttl, sim next header = %i , %i", ip_fields->ttl, ip_fields->protocol);
	printf ("s address = %x ", ip_fields->src_addr.address.ipv6_addr_ptr->addr32[0]);//%x
%x %x , ip_fields->src_addr.addr32[1],ip_fields->src_addr.addr32[2],ip_fields-
>src_addr.addr32[3]);
```

```
          //op_pk_nfd_set(eth_packet, "fields", eth_fields);
          //op_pk_nfd_set_pkt (eth_packet, "data", ip_packet); //reset


// Get packet fields from simulated packet
//op_pk_nfd_get (eh_sim_ptr, "next_hdr", &next_header);
//op_pk_nfd_get (eh_sim_ptr, "hdr_length", &header_length);



/*********************************************/
/*        OpT_uInt32    ver_traf_flow; //contains the version, traffic class and flow
          OpT_uInt16    payload_length;
          OpT_uInt8     next_header;
          OpT_uInt8     hop_limit;
          Ipv6T_Address source;
          Ipv6T_Address dest;
/***********************************************/



// Set packet fields in real packet
eh_hdr_ptr->ver_traf_flow = htonl(0x60000000);
eh_hdr_ptr->payload_length = ip_fields->frag_len;
eh_hdr_ptr->next_header = ip_fields->protocol;
eh_hdr_ptr->hop_limit = ip_fields->ttl;

//get and swap source
eh_hdr_ptr->source.addr32[0] = ip_fields->src_addr.address.ipv6_addr_ptr->addr32[0];
eh_hdr_ptr->source.addr32[1] = ip_fields->src_addr.address.ipv6_addr_ptr->addr32[1];
eh_hdr_ptr->source.addr32[2] = ip_fields->src_addr.address.ipv6_addr_ptr->addr32[2];
```

```
eh_hdr_ptr->source.addr32[3] = ip_fields->src_addr.address.ipv6_addr_ptr->addr32[3];

eh_hdr_ptr->source.addr32[0] =
(((eh_hdr_ptr->source.addr32[0] & 0xFF000000) >>24) |
((eh_hdr_ptr->source.addr32[0] & 0x00FF0000) >>8)  |
((eh_hdr_ptr->source.addr32[0] & 0x0000FF00) <<8)  |
((eh_hdr_ptr->source.addr32[0] & 0x000000FF) <<24) );

eh_hdr_ptr->source.addr32[1] =
(((eh_hdr_ptr->source.addr32[1] & 0xFF000000) >>24) |
((eh_hdr_ptr->source.addr32[1] & 0x00FF0000) >>8)  |
((eh_hdr_ptr->source.addr32[1] & 0x0000FF00) <<8)  |
((eh_hdr_ptr->source.addr32[1] & 0x000000FF) <<24) );

eh_hdr_ptr->source.addr32[2] =
(((eh_hdr_ptr->source.addr32[2] & 0xFF000000) >>24) |
((eh_hdr_ptr->source.addr32[2] & 0x00FF0000) >>8)  |
((eh_hdr_ptr->source.addr32[2] & 0x0000FF00) <<8)  |
((eh_hdr_ptr->source.addr32[2] & 0x000000FF) <<24) );

eh_hdr_ptr->source.addr32[3] =
(((eh_hdr_ptr->source.addr32[3] & 0xFF000000) >>24) |
((eh_hdr_ptr->source.addr32[3] & 0x00FF0000) >>8)  |
((eh_hdr_ptr->source.addr32[3] & 0x0000FF00) <<8)  |
((eh_hdr_ptr->source.addr32[3] & 0x000000FF) <<24) );
        //get and swap dest
eh_hdr_ptr->dest.addr32[0] = ip_fields->dest_addr.address.ipv6_addr_ptr->addr32[0];
eh_hdr_ptr->dest.addr32[1] = ip_fields->dest_addr.address.ipv6_addr_ptr->addr32[1];
eh_hdr_ptr->dest.addr32[2] = ip_fields->dest_addr.address.ipv6_addr_ptr->addr32[2];
eh_hdr_ptr->dest.addr32[3] = ip_fields->dest_addr.address.ipv6_addr_ptr->addr32[3];
```

```
eh_hdr_ptr->dest.addr32[0] =
(((eh_hdr_ptr->dest.addr32[0] & 0xFF000000) >>24) |
((eh_hdr_ptr->dest.addr32[0] & 0x00FF0000) >>8)  |
((eh_hdr_ptr->dest.addr32[0] & 0x0000FF00) <<8)  |
((eh_hdr_ptr->dest.addr32[0] & 0x000000FF) <<24) );

eh_hdr_ptr->dest.addr32[1] =
(((eh_hdr_ptr->dest.addr32[1] & 0xFF000000) >>24) |
((eh_hdr_ptr->dest.addr32[1] & 0x00FF0000) >>8)  |
((eh_hdr_ptr->dest.addr32[1] & 0x0000FF00) <<8)  |
((eh_hdr_ptr->dest.addr32[1] & 0x000000FF) <<24) );

eh_hdr_ptr->dest.addr32[2] =
(((eh_hdr_ptr->dest.addr32[2] & 0xFF000000) >>24) |
((eh_hdr_ptr->dest.addr32[2] & 0x00FF0000) >>8)  |
((eh_hdr_ptr->dest.addr32[2] & 0x0000FF00) <<8)  |
((eh_hdr_ptr->dest.addr32[2] & 0x000000FF) <<24) );

eh_hdr_ptr->dest.addr32[3] =
(((eh_hdr_ptr->dest.addr32[3] & 0xFF000000) >>24) |
((eh_hdr_ptr->dest.addr32[3] & 0x00FF0000) >>8)  |
((eh_hdr_ptr->dest.addr32[3] & 0x0000FF00) <<8)  |
((eh_hdr_ptr->dest.addr32[3] & 0x000000FF) <<24) );

// Advance the real packet write pointer
scdb_ptr->real_pk_offset += IPv6_LENGTH;

// Get the packet payload from the simulated packet and copy
// it to the real packet
printf(" after setting");
```

```
        op_pk_nfd_get_pkt(ip_packet, "data", &eh_sim_data_ptr);
        printf(" 1");
        data_size = op_pk_total_size_get (eh_sim_data_ptr)/8;
        printf(" 2");
        op_pk_fd_get_ptr (eh_sim_data_ptr, 0, (void**)&data_ptr);
        printf(" 3");
        op_prg_mem_copy (data_ptr, scdb_ptr->real_pk_ptr + scdb_ptr->real_pk_offset,
data_size);
        printf(" 4");

        // Advance the real packet write pointer
        scdb_ptr->real_pk_offset += data_size;

        // Free simulated packet memory
        op_prg_mem_free (data_ptr);
        op_pk_destroy (ip_packet);
        scdb_ptr->sim_pk_ptr = OPC_NIL;

        // Return success indication
        FRET (1);
        }
```

APPENDIX C: VOIP METRICS PYTHON SCRIPT

```
#!/usr/bin/env python
import csv
import sys

CHKSUM = 6
TIME = 1

client_pkts = csv.reader(open(sys.argv[1], 'r'))
server_pkts = csv.reader(open(sys.argv[2], 'r'))

c_row = next(client_pkts)
c_row = next(client_pkts)
c_checksum = c_row[CHKSUM]

s_row = next(server_pkts)
s_checksum = s_row[CHKSUM]

while (c_checksum != s_checksum):
        s_row = next(server_pkts)
        s_checksum = s_row[CHKSUM]

# Now, we assume that the two are aligned, and that we will be fine doing a
# straight comparison until the client runs out.
delta_sum = 0
jitter_delta_sum = 0
count = 0
while (1 == 1):
        if ((s_row[CHKSUM][0:2] == "0x") and (c_row[CHKSUM][0:2] == "0x")):
                if (int(s_row[CHKSUM], 16) == int(c_row[CHKSUM], 16)):
```

```
            current_time = (float(s_row[TIME]) - float(c_row[TIME]) - 105.63158)
            delta_sum += current_time
            if (count > 0):
                jitter_delta_sum += abs(current_time - last_time)
            count += 1
            last_time = current_time
    try:
        c_row = next(client_pkts)
        s_row = next(server_pkts)
    except StopIteration:
        break

print("Delta Sum: ", delta_sum)
print("Jitter Sum: ", jitter_delta_sum)
print("Count: ", count)
print("Average (ms): ", (delta_sum / count) * 1000)
print("Jitter (ms): ", (jitter_delta_sum / count) * 1000)
```

APPENDIX D: FRAMEWORK USER MANUAL

The Framework User Manual explains how to apply the IPv6 Simulation Framework developed in this thesis to other IPv6 applications. Details on the specific topologies included in the framework can be found in Chapter 5. It is highly recommended that users are familiar with OPNET Modeler and have taken the tutorials provided by the company.

The User Manual will cover the components of the simulation interface, configuration of SITL nodes within simulation and the static address configuration for the live device. One example will then be given to explain the steps of running the simulation and navigating between scenarios. A second example will show the steps needed to create a personalized traffic profile.

APPENDIX D.1 Components of the Simulation Interface

The simulation interface is the standard OPNET Modeler GUI, presented in Figure D.1, which shows the network topology at the highest level of abstraction. In this view, the Add Module pane is open on the left hand side. This pane is used to browse through available network devices to drag and drop into the topology. In the framework, these topologies are pre-built; therefore, the Add Module pane only will be used in the event the user wants to use a device more suited towards their live network.

This view is where attributes are accessed for each node by right clicking. For example, to set the IP on the router statically, right click on the router and click Edit Attributes (Figure D.2). Browse to IP, IPv6 Parameters, Interface Information, the Interface in question (in this case IF1) and finally to Global Addresses (Figure D.3). An IPv6 address can be entered directly into this form. Other attributes that can be set pertain to routing, Quality of Service, and other items that are not related to the framework. The other way to assign addresses in Modeler is through Rapid Deployment. Browse to Protocols, IP, Addressing, and Auto-Assign IP Addresses (Figure D.4). This menu will let you chose IPv4 or IPv6 to be deployed on all nodes in the topology. This option does not give the user any control over which addresses and subnets are assigned.

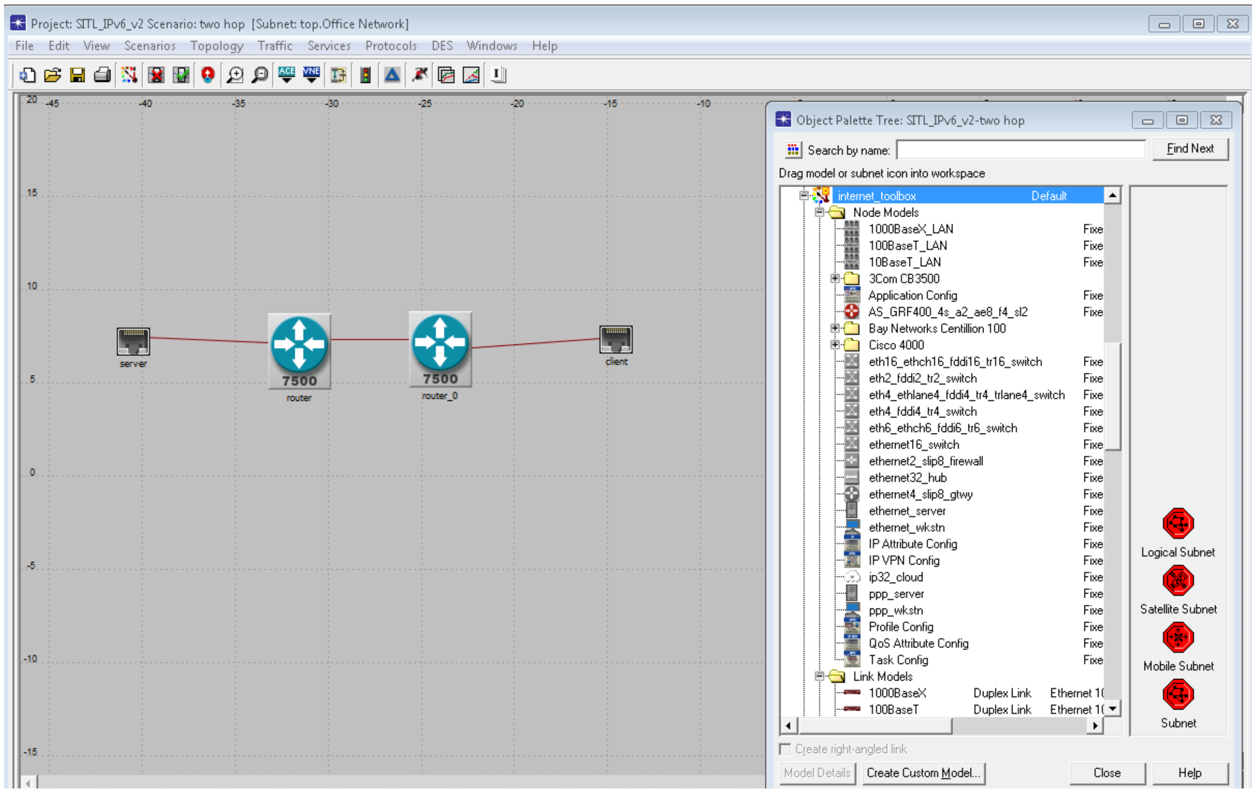


Figure D.1 OPNET GUI

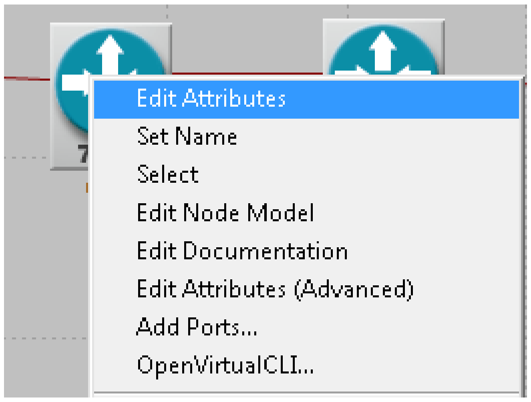
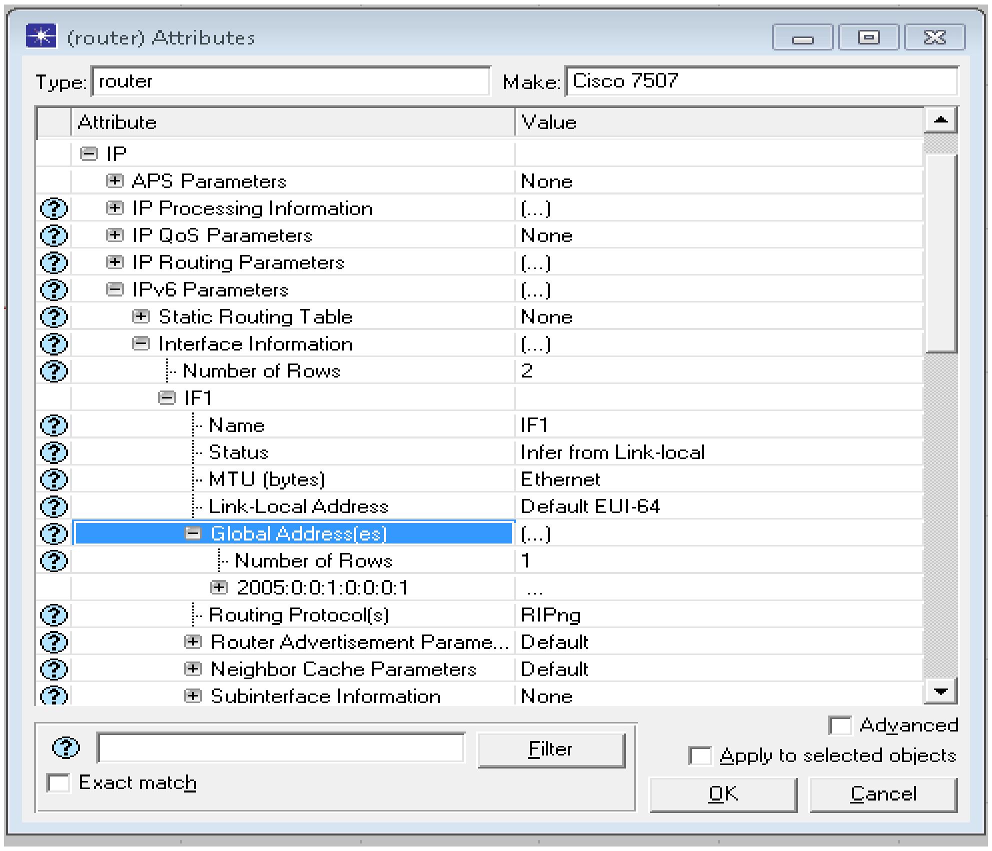


Figure D.2 Edit Attributes

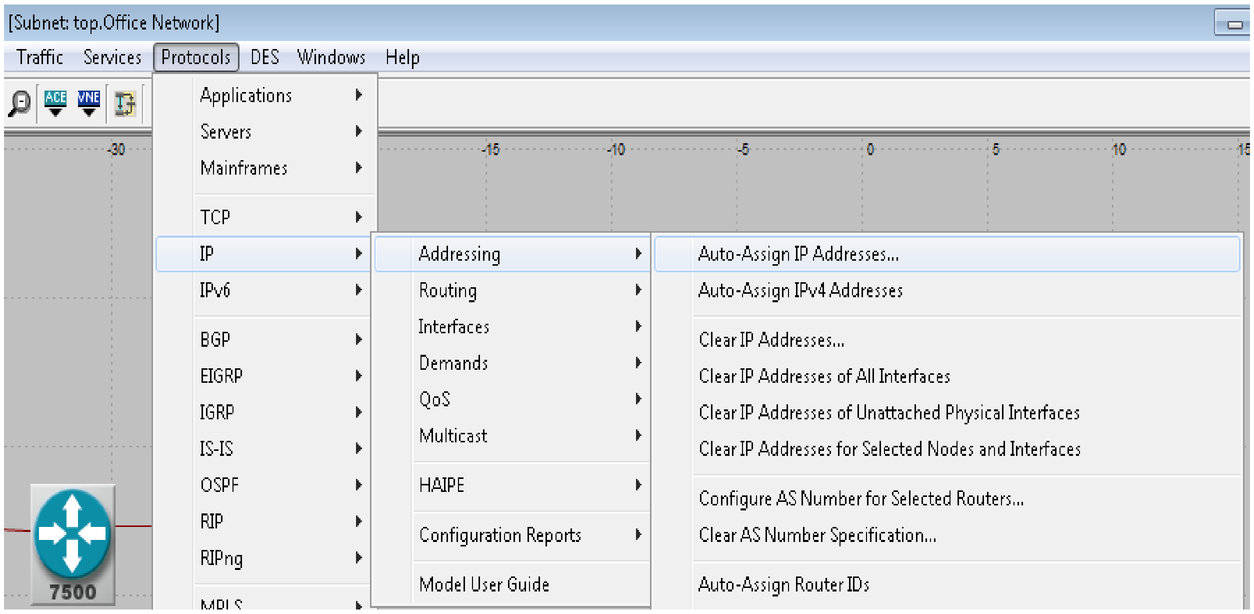


(router) Attributes

Type: router   Make: Cisco 7507

Attribute	Value
⊟ IP	
⊞ APS Parameters	None
⊞ IP Processing Information	(...)
⊞ IP QoS Parameters	None
⊞ IP Routing Parameters	(...)
⊟ IPv6 Parameters	(...)
⊞ Static Routing Table	None
⊟ Interface Information	(...)
Number of Rows	2
⊟ IF1	
Name	IF1
Status	Infer from Link-local
MTU (bytes)	Ethernet
Link-Local Address	Default EUI-64
⊟ Global Address(es)	(...)
Number of Rows	1
⊞ 2005:0:0:1:0:0:0:1	...
Routing Protocol(s)	RIPng
⊞ Router Advertisement Parame...	Default
⊞ Neighbor Cache Parameters	Default
⊞ Subinterface Information	None

Figure D.3 Statically Setting an IP



[Subnet: top.Office Network]
Traffic
Services
Protocols
DES
Windows
Help
Applications
Servers
Mainframes
TCP
IP
IPv6
BGP
EIGRP
IGRP
IS-IS
OSPF
RIP
RIPng
MPLS
Addressing
Routing
Interfaces
Demands
QoS
Multicast
HAIPE
Configuration Reports
Model User Guide
Auto-Assign IP Addresses...
Auto-Assign IPv4 Addresses
Clear IP Addresses...
Clear IP Addresses of All Interfaces
Clear IP Addresses of Unattached Physical Interfaces
Clear IP Addresses for Selected Nodes and Interfaces
Configure AS Number for Selected Routers...
Clear AS Number Specification...
Auto-Assign Router IDs
7500

Figure D.4 Rapid Deployment

Another component of this interface is the ability to switch between scenarios. This ability is used in the framework to switch between topologies. Note: all nodes must have their attributes reconfigured when switching to a new scenario. Browse to Scenarios, Switch To Scenario, and choose the topology to use (Figure D.5).

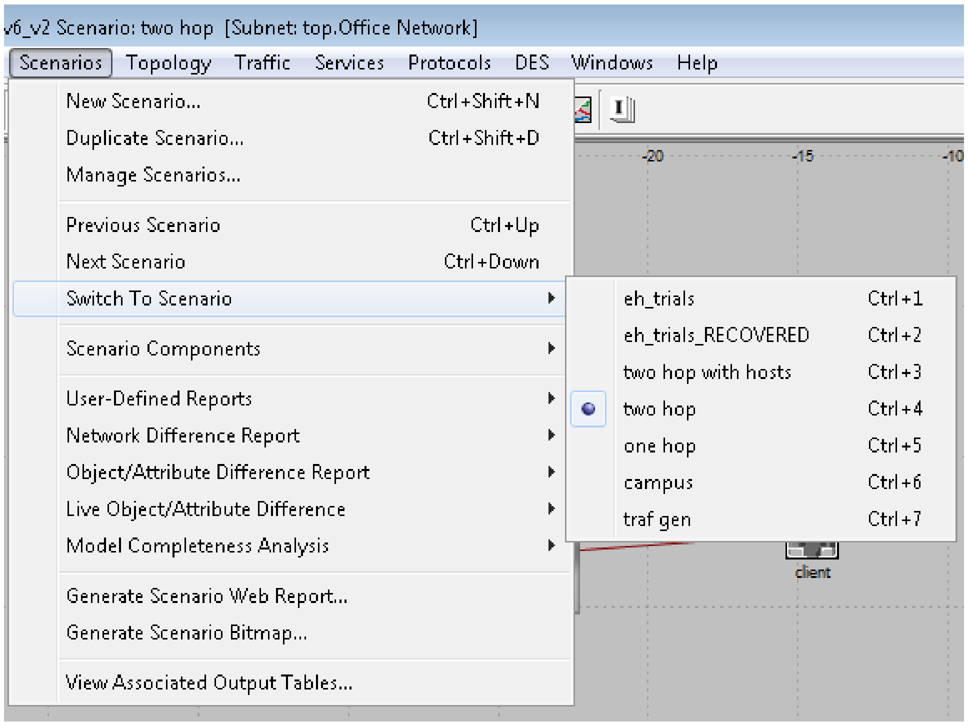

v6_v2 Scenario: two hop [Subnet: top.Office Network]
Scenarios
Topology
Traffic
Services
Protocols
DES
Windows
Help
New Scenario... Ctrl+Shift+N
Duplicate Scenario... Ctrl+Shift+D
Manage Scenarios...
Previous Scenario Ctrl+Up
Next Scenario Ctrl+Down
Switch To Scenario
Scenario Components
User-Defined Reports
Network Difference Report
Object/Attribute Difference Report
Live Object/Attribute Difference
Model Completeness Analysis
Generate Scenario Web Report...
Generate Scenario Bitmap...
View Associated Output Tables...
eh_trials Ctrl+1
eh_trials_RECOVERED Ctrl+2
two hop with hosts Ctrl+3
two hop Ctrl+4
one hop Ctrl+5
campus Ctrl+6
traf gen Ctrl+7
-20
-15
-10
client

Figure D.5 Switching Scenarios

Another component of the framework is picking statistics to capture. Although statistics are pre-chosen for the user in the framework, the user can modify what he wants to capture by right clicking on a node or the background of the interface and choosing Choose Individual DES Statistics (Figure D.6). This leads to a menu with several statistics to choose from; for this example, IPv6 and SITL statistics are shown in Figure D.7.

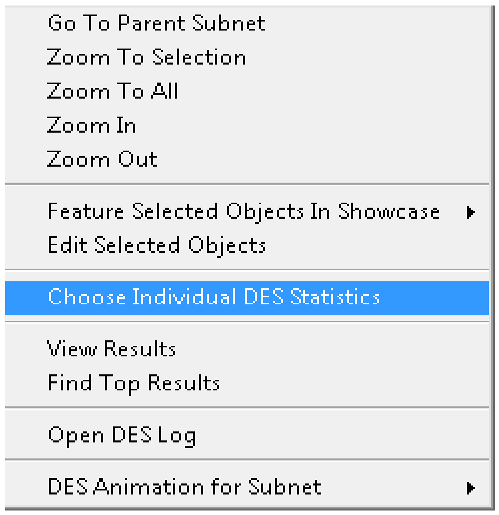
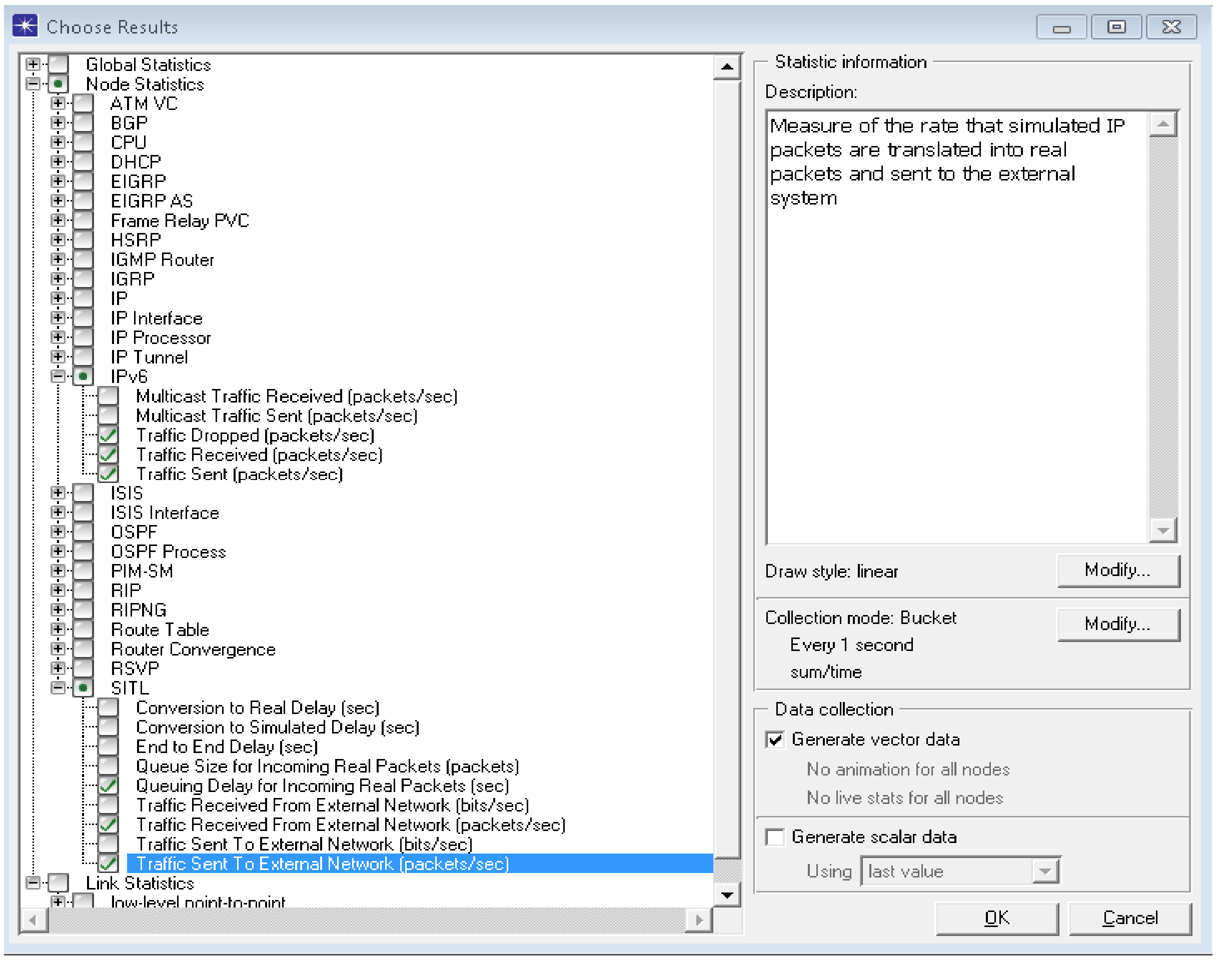


Figure D.6 Defining Statistics



```
Global Statistics
Node Statistics
    ATM VC
    BGP
    CPU
    DHCP
    EIGRP
    EIGRP AS
    Frame Relay PVC
    HSRP
    IGMP Router
    IGRP
    IP
    IP Interface
    IP Processor
    IP Tunnel
    IPv6
        Multicast Traffic Received (packets/sec)
        Multicast Traffic Sent (packets/sec)
      ✓ Traffic Dropped (packets/sec)
      ✓ Traffic Received (packets/sec)
      ✓ Traffic Sent (packets/sec)
    ISIS
    ISIS Interface
    OSPF
    OSPF Process
    PIM-SM
    RIP
    RIPNG
    Route Table
    Router Convergence
    RSVP
    SITL
        Conversion to Real Delay (sec)
        Conversion to Simulated Delay (sec)
        End to End Delay (sec)
        Queue Size for Incoming Real Packets (packets)
      ✓ Queuing Delay for Incoming Real Packets (sec)
        Traffic Received From External Network (bits/sec)
      ✓ Traffic Received From External Network (packets/sec)
        Traffic Sent To External Network (bits/sec)
      ✓ Traffic Sent To External Network (packets/sec)
Link Statistics
    low-level point-to-point
```

Statistic information

Description:

Measure of the rate that simulated IP packets are translated into real packets and sent to the external system

Draw style: linear

Collection mode: Bucket
Every 1 second
sum/time

Data collection

[✓] Generate vector data
No animation for all nodes
No live stats for all nodes

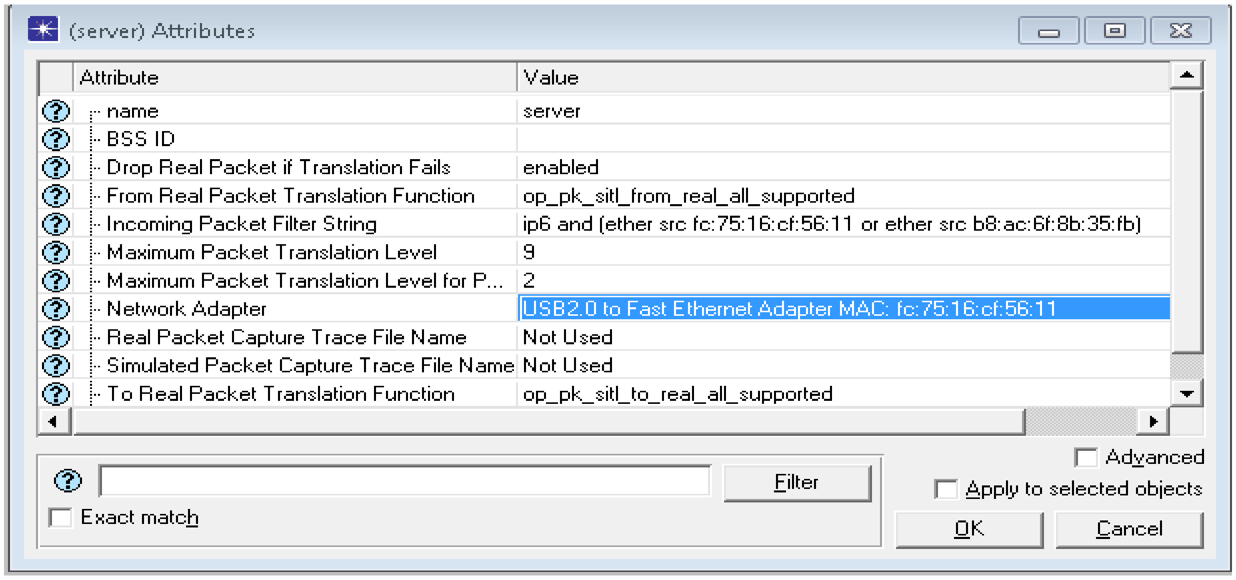
[ ] Generate scalar data
Using last value

Figure D.7 Choosing Statistics

The GUI’s abstraction level can be narrowed down by doubling clicking on any component to obtain the corresponding process diagram. The GUI has several more menus and options but they are not directly used in applying the framework. Those users that wish to learn more about the interface can access the OPNET Documentation for more information [34].

APPENDIX D.2 SITL Configuration

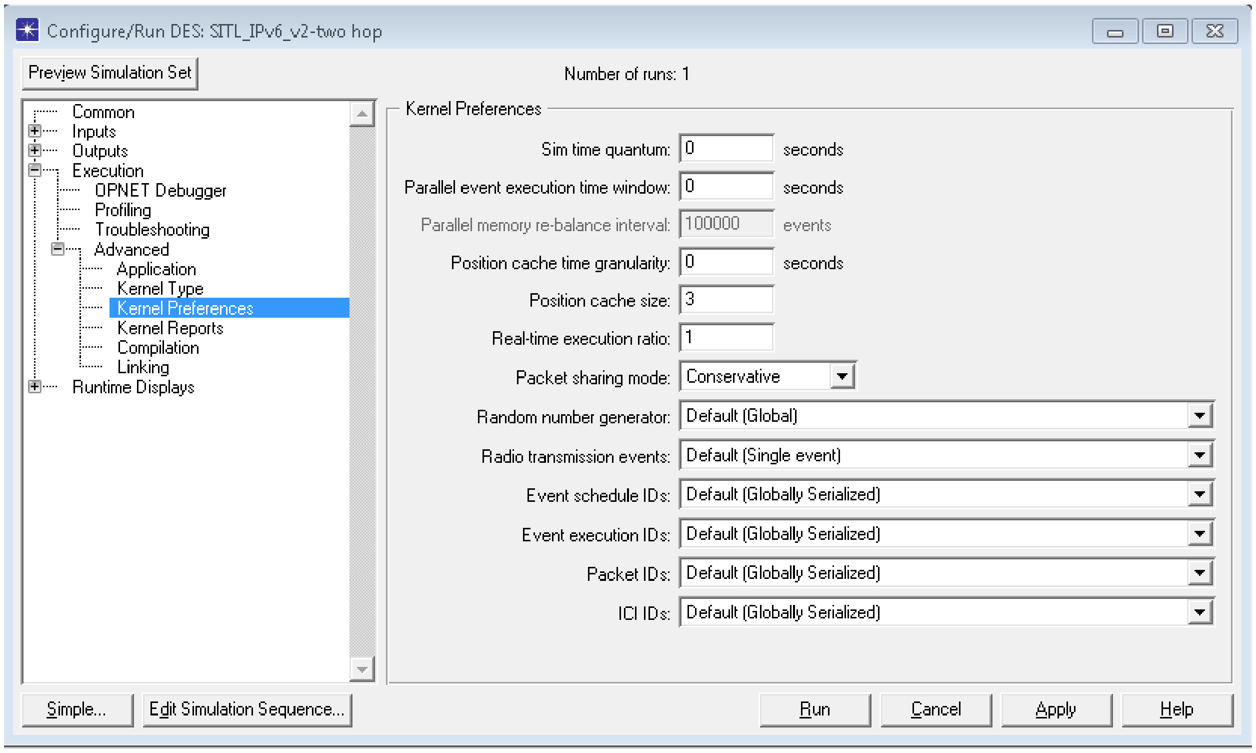
To run the SITL module in the framework, the simulation SITL node must be configured with a packet filter and interface. The packet filter format in Chapter 5 is the generic format used in the framework. Users can add conditions to the filter should they desire in the SITL configuration menu in the Incoming Packet Filter String form (Figure D.8). The Network Adapter, also in the SITL configuration menu, should be set to the interface that the user wants to capture packets from. Modeler must be run with Administrator privileges to be able to set this attribute. If a custom translation function is used the following fields must be changed to use the custom translation methods: From Real Packet Translation Function, To Real Packet Translation Function, and Translation Initialization Function. More information about custom SITL translations can be found in the OPNET Documentation [34].



Attribute	Value
name	server
BSS ID	
Drop Real Packet if Translation Fails	enabled
From Real Packet Translation Function	op_pk_sitl_from_real_all_supported
Incoming Packet Filter String	ip6 and (ether src fc:75:16:cf:56:11 or ether src b8:ac:6f:8b:35:fb)
Maximum Packet Translation Level	9
Maximum Packet Translation Level for P...	2
Network Adapter	USB2.0 to Fast Ethernet Adapter MAC: fc:75:16:cf:56:11
Real Packet Capture Trace File Name	Not Used
Simulated Packet Capture Trace File Name	Not Used
To Real Packet Translation Function	op_pk_sitl_to_real_all_supported

Figure D.8 SITL Attribute Menu

For a simulation to run with SITL, the simulation kernel must run with a real-time execution ratio of 1. This is set in the simulation pane (Browse to DES, Configure/Run Discrete Event Simulation in the GUI) under Execution, Advanced, Kernel Preferences, Real-time execution ratio shown in Figure D.9.


Configure/Run DES: SITL_IPv6_v2-two hop

Number of runs: 1

Kernel Preferences

Sim time quantum:	0	seconds
Parallel event execution time window:	0	seconds
Parallel memory re-balance interval:	100000	events
Position cache time granularity:	0	seconds
Position cache size:	3	
Real-time execution ratio:	1	
Packet sharing mode:	Conservative	
Random number generator:	Default (Global)	
Radio transmission events:	Default (Single event)	
Event schedule IDs:	Default (Globally Serialized)	
Event execution IDs:	Default (Globally Serialized)	
Packet IDs:	Default (Globally Serialized)	
ICI IDs:	Default (Globally Serialized)	

Figure D.9 SITL Real-time Ratio

In addition to configuring SITL in simulation, the live device/application must be statically configured with an IP address and gateway as the evaluation results in Chapter 6 presented. For a Linux based system the commands are:

```
ifconfig <interface> add inet6 <IP Address>
route –A inet6 add default gw <IP Address of First Hop in Simulation> dev <interface>
```

Other commands can also be used to achieve the same configuration in Linux.

For a Windows based system, the IP Address and gateway can be set using the command line or within the Windows Network Connections GUI.

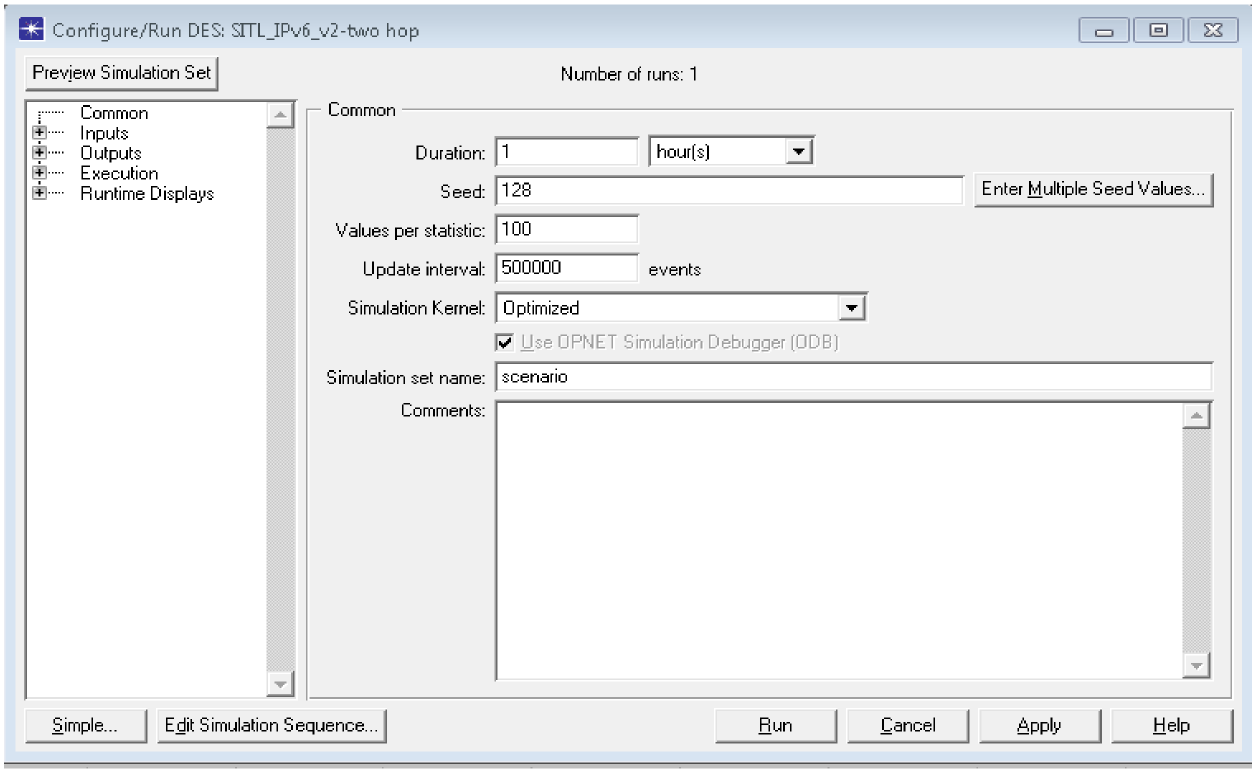
Please read Chapter 5 on SITL configuration between live devices and simulation machines to make an informed decision on how best to connect an application to simulation.

APPENDIX D.3 Example 1: Running a Simulation

Simulations are simple to run after configuring SITL and the IP attributes for each node. These are the items that need to be done before running the simulation:

1. Check that IP Addresses are on all simulated nodes (Provided by Framework)
2. SITL nodes have Packet Filter (MUST be modified to User MAC addresses)
3. SITL nodes have interfaces chosen (MUST be modified to User Interface)
4. Statistics have been chosen (Provided by Framework)
5. Application device has static IP and gateway (MUST be configured by User)
6. Real-time Execution Ratio is set to 1 (Provided by Framework)
7. Ensure Background Traffic is configured (Optional based on Scenario)

After configuring the simulation and the application in testing, browse to DES, Configure/Run Discrete Event Simulator. The default simulation menu will appear and look like Figure D.10.


Configure/Run DES: SITL_IPv6_v2-two hop

Number of runs: 1

Common

Duration: 1 hour(s)

Seed: 128

Values per statistic: 100

Update interval: 500000 events

Simulation Kernel: Optimized

☑ Use OPNET Simulation Debugger (ODB)

Simulation set name: scenario

Comments:

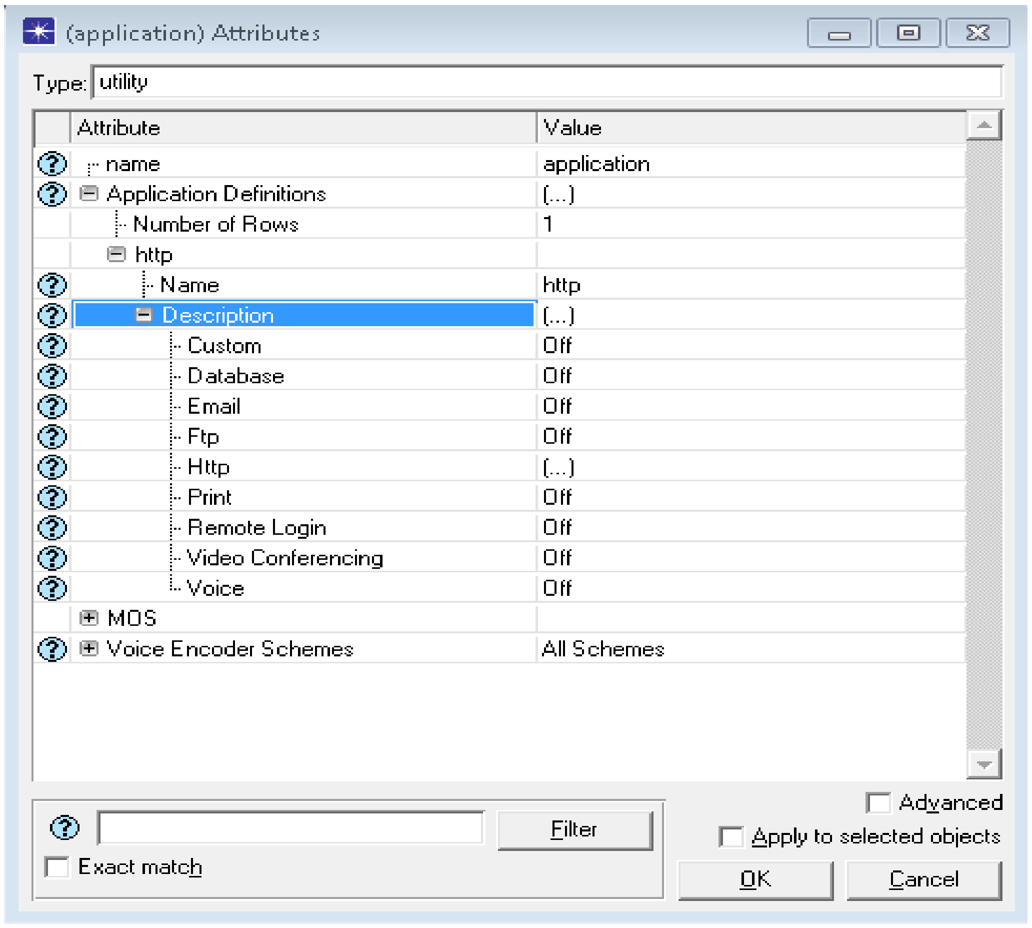
Figure D.10 Default Simulation Menu

In this menu, the user can set the duration of the simulation, the values collected per statistic, the seed used for the pseudo-random generator for events, and the type of kernel. To run the debugger, the Development Kernel must be used. Click Run to start the simulation. From the application machine, run any test cases to evaluate the application.

APPENDIX D.4 Example 2: Configuring a Traffic Profile

The framework provides traffic profiles that are configured to match Virginia Tech live network values. Users not affiliated with Virginia Tech can create a personalized traffic profile that matches their network statistics. To create a traffic profile, users need to have collected traffic load values for nodes or routers on their networks. These values are used in the profile creation.

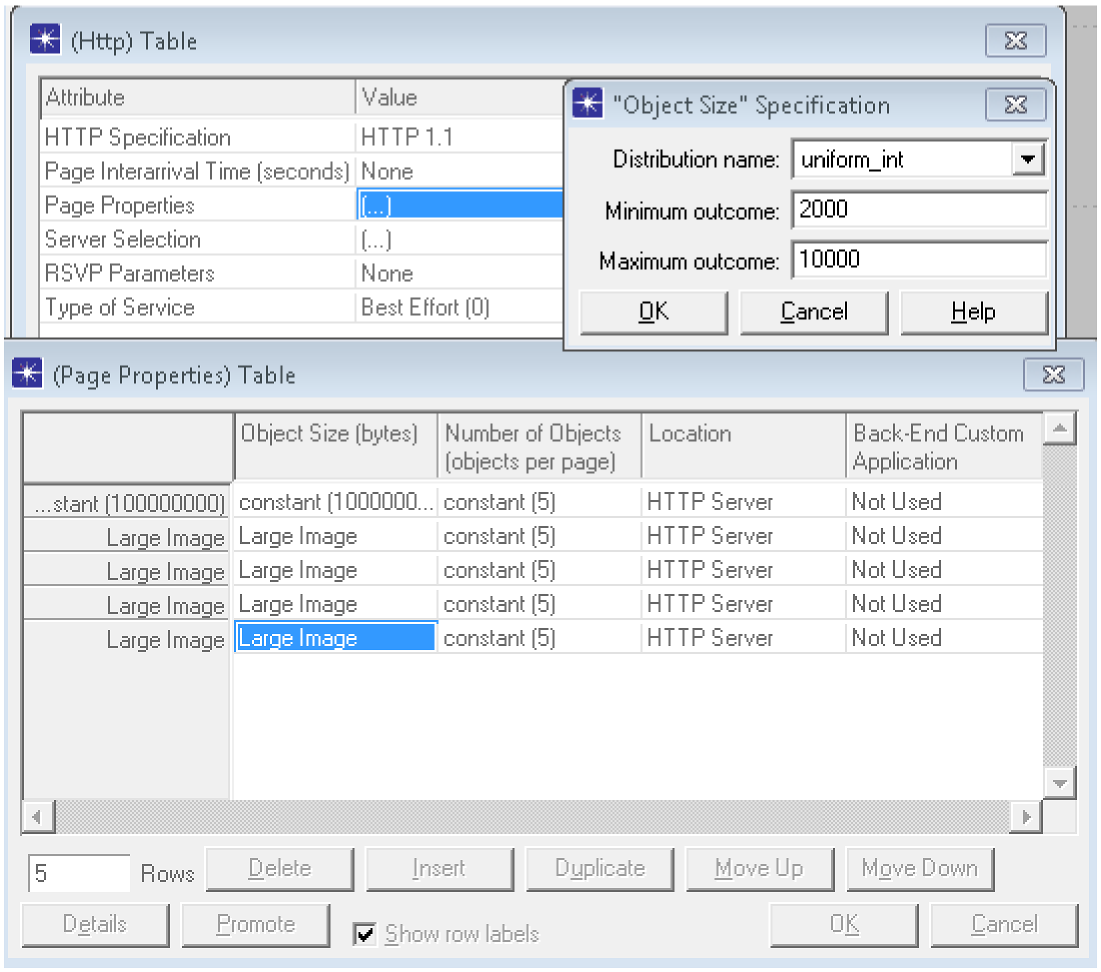
First, choose one of the scenarios of the framework that have integrated Background Traffic (Two Hop with Background Traffic or one of the Campus layouts). First, right click on the Application Config module and click Edit Attributes. The Application Configuration menu, shown in Figure D.11, is used to determine the traffic demand for a single application. In this example, HTTP traffic was chosen as the application and a custom value is being set for the demand. The customization menu, which displays the number of items sent per request for an HTTP page, is accessed by double clicking the HTTP application field. The items can also be modified by double clicking. The three configuration menus are shown in Figure D.12. The "large image" item is being modified in the top right menu. The size units for these forms are bytes. Adjust these figures to match the values obtained from the live network.


(application) Attributes

Type: utility

Attribute	Value
name	application
Application Definitions	(...)
Number of Rows	1
http	
Name	http
Description	(...)
Custom	Off
Database	Off
Email	Off
Ftp	Off
Http	(...)
Print	Off
Remote Login	Off
Video Conferencing	Off
Voice	Off
MOS	
Voice Encoder Schemes	All Schemes

Figure D.11 Application Configuration Menu


(Http) Table

Attribute	Value
HTTP Specification	HTTP 1.1
Page Interarrival Time (seconds)	None
Page Properties	(...)
Server Selection	(...)
RSVP Parameters	None
Type of Service	Best Effort (0)

"Object Size" Specification

Distribution name:	uniform_int
Minimum outcome:	2000
Maximum outcome:	10000

(Page Properties) Table

	Object Size (bytes)	Number of Objects (objects per page)	Location	Back-End Custom Application
...stant (100000000)	constant (1000000...	constant (5)	HTTP Server	Not Used
Large Image	Large Image	constant (5)	HTTP Server	Not Used
Large Image	Large Image	constant (5)	HTTP Server	Not Used
Large Image	Large Image	constant (5)	HTTP Server	Not Used
Large Image	Large Image	constant (5)	HTTP Server	Not Used

5 Rows

Show row labels ✓

Figure D.12 Configuration Menus

After configuring all desired applications, the Profile Config module must be used to create a profile containing the applications. A profile must have one application defined but can contain multiple definitions. After configuring the applications in the Profile Config module using drop down menus, the profile must be deployed to the simulated nodes.

To deploy a profile, browse to Protocols, Applications, Deploy Defined Applications (Figure D.13). This brings up the Deployment menu, shown in Figure D.14, in which all defined profiles are listed in the right pane and all nodes in the topology are listed in the left pane. For each profile, define at least one client and server. Apply changes and exit the menu.

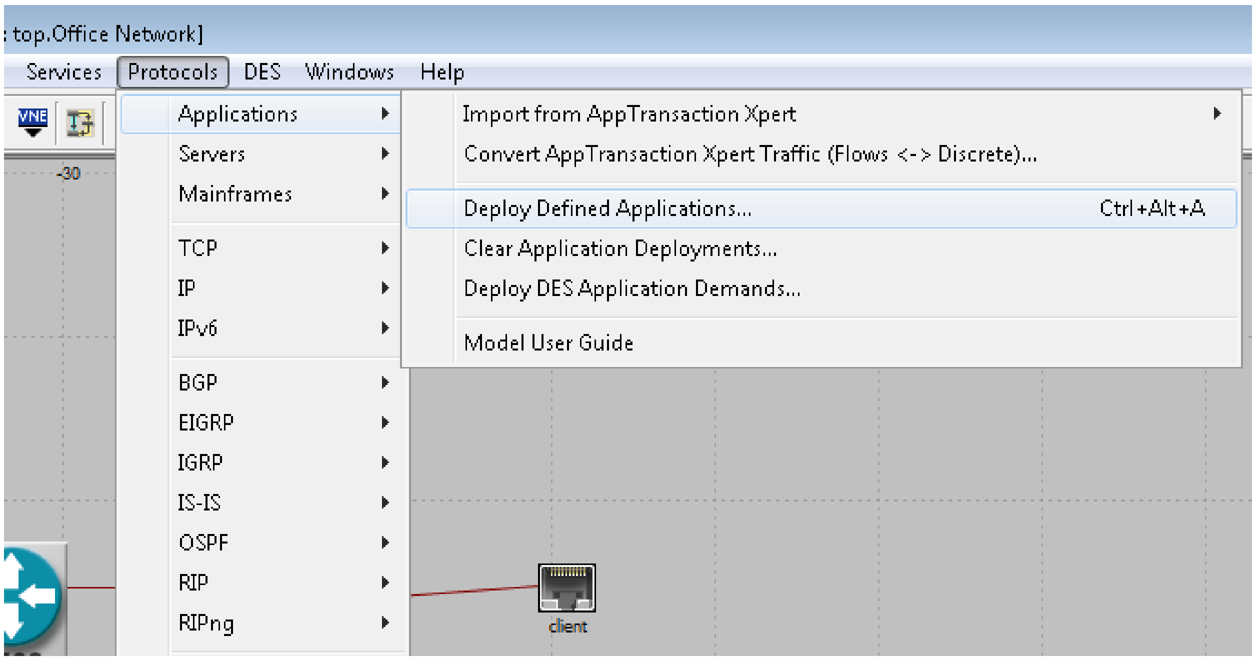
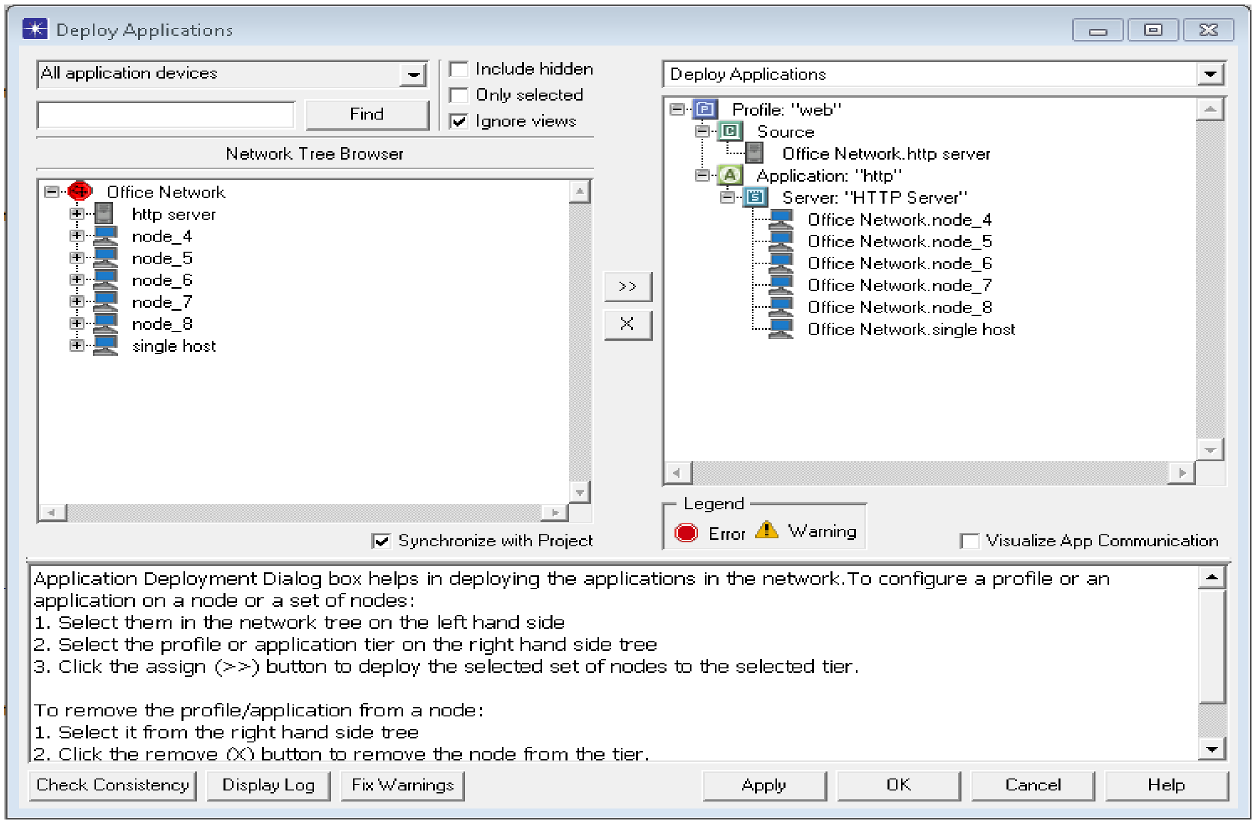

: top.Office Network]
Services
Protocols
DES
Windows
Help
Applications
Servers
Mainframes
TCP
IP
IPv6
BGP
EIGRP
IGRP
IS-IS
OSPF
RIP
RIPng
Import from AppTransaction Xpert
Convert AppTransaction Xpert Traffic (Flows <-> Discrete)...
Deploy Defined Applications...
Ctrl+Alt+A
Clear Application Deployments...
Deploy DES Application Demands...
Model User Guide
client

Figure D.13 Deploy Defined Applications



Application Deployment Dialog box helps in deploying the applications in the network.To configure a profile or an application on a node or a set of nodes:
1. Select them in the network tree on the left hand side
2. Select the profile or application tier on the right hand side tree
3. Click the assign (>>) button to deploy the selected set of nodes to the selected tier.

To remove the profile/application from a node:
1. Select it from the right hand side tree
2. Click the remove (X) button to remove the node from the tier.

Figure D.14 Deployment Menu

Use statistics gathering to determine if the traffic demands are set for the right values and adjust accordingly.