Click Modular Router

Research Internship at PATS Group

Faculty of Sciences, University of Antwerp

Promoter : Chris Blondia
Supervisor : Johan Bergs

Student : Hoang Trung Hieu

TABLE OF FIGURES AND CHARTS

1. Introduction

Click Modular Router is a software architecture for building flexible and configurable router. It was introduced in PhD thesis of Eddie Kohler, Department of Electric Engineering and Computer Science, MIT, in 2000. Click Modular Router design and architecture based on the idea that a router consists of a set of connected packet processing modules (called elements). Each element in the router implements a simple packet processing operation like classification, scheduling or interfaces with network device. In order to build a router configuration, users choose a collection of proper elements and connect them to a directed graph. Packet received at input port will traverse through the edges of this graph before reaching output port. Router configuration script is written in a declarative language.

Latest Click distribution offers a wide range of elements allowing users to build any type of router which can match their requirements. On the other hand, it also provides an open framework to develop elements with specific purposes on their own.

Although Click design originally focused on forwarding packets and other active tasks, many other problems can be tackled by using Click as well, from experiment of new network operation, checking new protocol to network measurement. Click is well suited for academic projects with its strength and flexibility.

2. Research Internship Objectives

The objective of this research internship includes:

- Have understanding of Click Modular Router architecture and apply in building a software router
- Examine current Click distribution, identifying the functionality available in IPv4 stack but missing in IPv6 stack
- Implement some missing functionality in IPv6 stack according to their priority of importance
- Benchmark IPv4 and IPv6 elements, identifying performance bottleneck (if any) in current IPv6 implementation

3. Prerequisites

In order to run configuration script and compile new elements, it is required that Click are correctly installed in advance.

3.1. To install Click Modular Router in Ubuntu

Step 1: Download latest Click distribution at http://www.read.cs.ucla.edu/click/

Step 2: Go to Click source directory and run

```
./configure [--prefix=PREFIX]
```

If you want to enable all elements in Click distribution, run

```
./configure –enable-all-elements
```

Step 3: Build and install the click module

```
sudo make
```

Step 4: Make install

```
sudo make install
```

3.2. To run a router configuration script

```
click [script_file]
```

3.3. To compile and add new element class to Click

Step 1: Put element source code in 'elements/' directory where Click was installed in earlier section. Choose the sub-directory that is most appropriate for that element.

Step 2: Configure compile options. For example, if element is placed in 'local' directory, make sure to provide the '--enable-local' argument to 'configure'.

```
./configure –enable-local
```

Step 3: Check the source files in the 'elements/' subdirectories for EXPORT_ELEMENT directives, and compiles a list of elements that Click should compile.

```
make elemlist
```

Step 4: Check the 'userlevel/elements.conf' and 'linuxmodule/elements.conf' files to ensure .cc source file in this list.

Step 5: Run `'make install'`

4. Configuration Script

In this section, I chose to develop a router which is able to support forwarding incoming packets to 02 different subnets. The configuration script must be run on a computer with 02 (physical or logical) network interfaces.

Topology is based on IP Router model given in Eddie Kohler's PhD thesis and the script mainly re-use *fake-iprouter.click* example at http://www.read.cs.ucla.edu/click/examples/fake-iprouter.click. The purpose of this work is to demonstrate the functionality of critical elements in IPv4 stack and help to have understanding of how to build a router which can satisfy specific requirements.

Configuration Graph

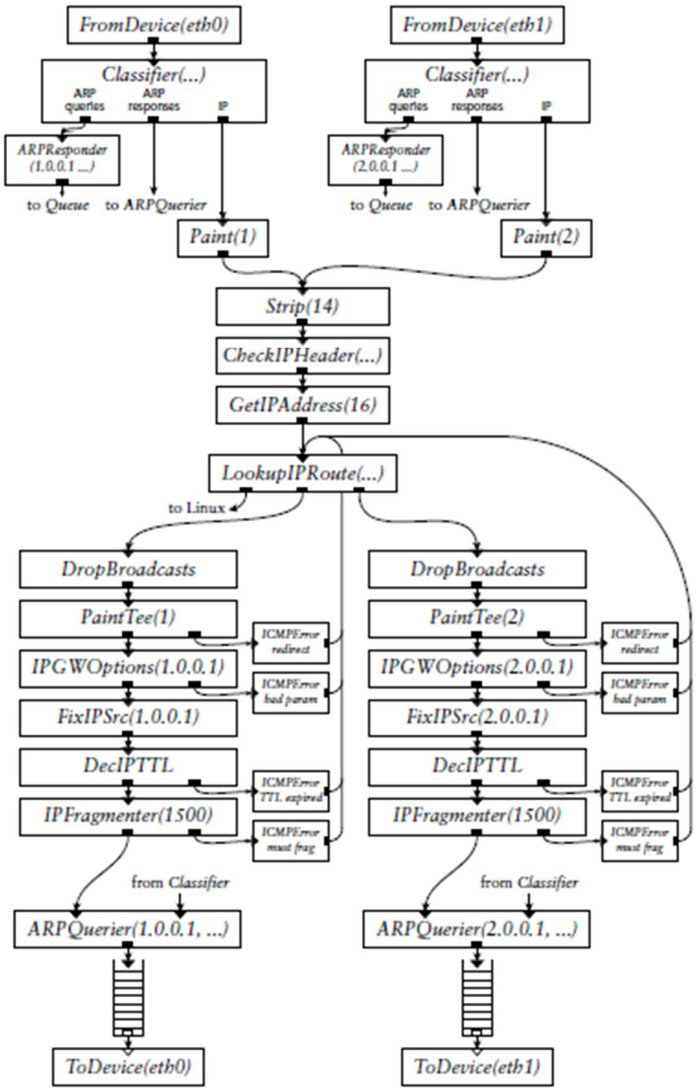

Figure 1. IP Router configuration [2]

Configuration Script

Please refer to *iprouter.click* for detail code.

Test Model

The following topology is used to check whether this router works correctly.

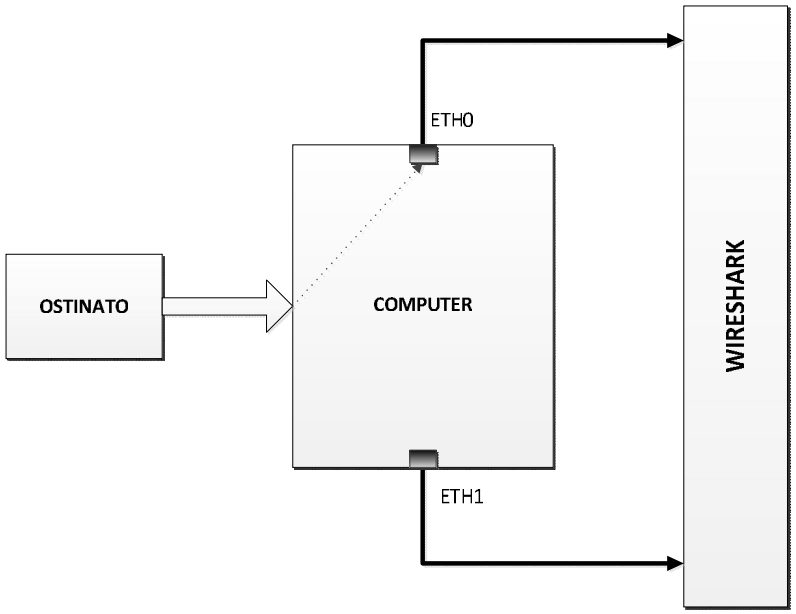

Figure 2. Test model for IP Router

Ostinato: is an open-source, cross-platform network packet crafter/traffic generator and analyzer with a friendly GUI [6]. It is used to generate packets of several streams with IP addresses corresponding to 02 subnets and feed to input port of the router.

Click Router: is a computer with 02 network interfaces ETH0 and ETH1 to simulate real router.

Wireshark: is a free and open source packet analyzer. It is used to capture the content of packets received at each network interface.

5. Issues with IPv6 support in current Click distribution

Click Modular Router has been distributed since 2000. Originally, Click was mainly developed for IPv4 stack as a matter of the fact that IPv6 was not widely used at that time. In recent years, with the rapid development of technology, every device can become connected. Each connected device must have a unique address in order to be identified in the network while the number of addresses offered by IPv4 stack is only limited. Although some techniques were proposed to re-use network addresses, IPv4 address is getting

depleted and the gradual transition to IPv6 is an unavoidable trend.

Fortunately, Click also provides framework for users to develop new elements themselves. A few elements supporting IPv6 stack have been added to Click by Eddie Kohler and developers around the world. In the latest Click release (version 2.0.1 on September 24, 2011), there are 18 elements supporting IPv6 packet processing. However, as compared to IPv4 stack implementation, the functionality supported is still limited and some existing elements got very poor implementation.

Missing functionalities are listed in the Table 1. It is noted that this list does not cover all but the most critical ones.

No	Functionality	Corresponding element in IPv4 stack	Current status
I	**Basic functionalities**		
1	Classify IPv6 packet by input pattern string	IPClassifier	Not implemented
2	Process IPv6 extension header such as Hop-by-Hop, Destination, Routing and so forth	Not applicable	Not implemented
3	Filter IPv6 packet by content	IPFilter	Not implemented
4	Set IPv6 Traffic Class	Not applicable	Not implemented
5	Set IPv6 Flow Label	Not applicable	Not implemented
6	Fragment packet with size exceeding link MTU	IPFragmenter	Poor implementation
7	Erase IPv6 packet payload	EraseIPPayload	Not implemented
8	Resemble fragmented IPv6 packet	IPRessembler	Not implemented
9	Set destination IPv6 address annotations randomly	SetRandIPAddress	Not implemented
10	Set destination IPv6 address annotations	SetIPAddress	Not implemented
II	**ICMPv6**		
1	Check ICMPv6 header	CheckICMPHeader	Not implemented
2	ICMPv6 ping request and reply	ICMPPingSource and ICMPPingResponder	Not implemented
3	Encapsulate packet into ICMPv6 packet	ICMPPingEncap	Not implemented
III	**DHCPv6**		Current Click distribution does not have any element to support DHCPv6

Table 1. Missing functionalities in IPv6 stack

6. IPv6 elements implementation

As stated earlier in section 2, one objective of this internship is to implement missing

functionalities in IPv6 stack. However, because the number of missing functionalities is really big, it is impossible to implement all of them in this project. Only a few elements were selected according to their priority of importance.

After discussing with my supervisor, I decided to implement IPv6 classifier and some IPv6 header extension processing elements including Hop-by-Hop, Fragment and Routing header. There are several reasons for this selection:

- Firstly, it can be seen from many configuration script that selected elements are used more frequently than the others.
- Secondly, by developing these elements, it would help me to understand deeply IPv6 stack.
- Finally, the code of these elements can be easily re-used in case I want to develop other functionalities in future.

It is noted that before re-compiling Click, all source and header files of new elements should be placed in *[Click installed directory]/elements/ip6* directory.

6.1. IP6Classifier

6.1.1 Usage

Configuration: *IP6Classifier(PATTERN_1, PATTERN_2..., PATTERN_N)*

Input port: *01*

Output port: *any*

Processing: *Push*

Description

IP6Classifier classifies IPv6 packet stream by patterns regulated by user. Each element's output port is associated with corresponding pattern. The pattern syntax is as follow

ip proto PROTO

> PROTO is an IP protocol name including tcp/udp/icmp. Match packets of the given protocol.

ip vers VERSION

> VERSION is a value between 0 and 15. Match IP packets with the given version.

ip hll HLL

HLL is hop limit value. Match IP packets with the given HLL value.

ip cos COS

COS is a 8-bit traffic class value. Match IP packets with given COS value.

ip flow FL

FL is a 20-bit flow label value. Match IP packets with the given FL value.

icmp type TYPE

TYPE is an ICMP packet type. Match ICMP packets with the given TYPE

udp port PORT

PORT is a port number. Match UDP packets with source or destination port as given in PORT

tcp port PORT

PORT is a port number. Match TCP packets with source or destination port as given in PORT

[SRCORDST] host IPADDR

IPADDR is list of IPv6 addresses and SRCORDST is 'src', 'dst', 'src or dst', or 'src and dst'. Matches packets sent to (src) and/or from (dst) the given addresses.

[SRCORDST] net NETADDR

NETADDR is an IPv6 network address (i.e only 64 high order bits are take into consideration) and SRCORDST is 'src', 'dst', 'src or dst', or 'src and dst. Matches packets sent to (src) and/or from (dst) the given network.

[SRCORDST] [tcp | udp] port PORT

PORT is a list of TCP or UDP port numbers and SRCORDST is as above. Matches packets sent to (src) and/or from (dst) the given TCP or UDP port.

true: Match every packet.

false: Match no packet.

6.1.2. Implementation

Challenges encountered

Click distribution supports packet classification for IPv4 packets in IPClassifier element. IPClassifier is inherited from IPFilter element which filters IP packets by content. At the beginning, I intended to re-use the code from IPClassifier element for IPv6 classifier. However, after carefully examining the code of IPFilter and IPClassifier elements, it turned out that the existing code is too complicated to re-use. The author built up a complete parser to analyze pattern string and used many code libraries that make it difficult to trace. Eventually, I decided to develop this classifier element from the scratch by myself.

Data structure for IPv6 extension header

In IPv6 packet, there are 02 distinct types of header, main header and extension header. As compared to IPv4 packet, the functionalities of option is removed from main header in order to decrease the impact of overhead processing on performance of network devices. Main header remains fixed in size (40 bytes) while extension header can be added as needed. Header extension of different types appears in different structures.

Main header is already declared as *click_ip6* structure in *'/include/clicknet/ip6.hh'* header file. We should have a common data type representing extension header of different types which can be used not only in this element but the others as well. Keep this requirement in mind, a union data type is selected to describe extension header of IPv6 packet. It is declared in *'/include/clicknet/ip6.hh'* header file as well.

```
/*IP6 Header Extension*/
union click_ip6_header_ext {

	/*For Hop-by-Hop, Destination and Routing Header*/
	struct {
		uint8_t _nxt_header;		/* Next header*/
		uint8_t _header_length;		/*Header lenght*/
	} ip6_header_extension;

	/*Routing Header*/
	struct {
		uint8_t _nxt_header;		/* Next header*/
		uint8_t _header_length;		/*Header lenght*/
		uint8_t _routing_type;
		uint8_t _segment_left;
		uint8_t _reserved;
		uint8_t _strict_loose[3];
	} ip6_routing_extension;

	/*For Fragment Header */
	struct {
		uint8_t _frag_nxt_header;	/* Next header*/
		uint8_t _frag_reserved;		/* Reserved 8 bits*/
```

```
            uint16_t _frag_offset_flag;
            uint32_t _frag_id;            /*Identification 32 bits*/
      } ip6_frag;

      /*For Authentication Header*/
      struct {
            uint8_t _ip6_nxt_header;      /* Next header*/
            uint8_t _ip6_payload_length;  /* Payload length*/
            uint16_t _ip6_reserved;       /* Reserved*/
            uint32_t _ip6_spi;            /* Security parameter index*/
            uint32_t _ip6_sn;             /* Sequence number*/
      } ip6_auth_header;

      /*Encapsulating Security Payload header is not supported*/

      /*TCP header*/
      struct {
            uint16_t _src_port;
            uint16_t _dst_port;
            uint32_t _seq_no;
            uint32_t _ack_no;
            uint16_t _data_offset;
            uint16_t _window_size;
            uint16_t _checksum;
            uint16_t _urgent_pointer;
      } ip6_tcp_header;

      /*UDP header*/
      struct {
            uint16_t _src_port;
            uint16_t _dst_port;
            uint16_t _length;
            uint16_t _checksum;
      } ip6_udp_header;

      /*ICMP header*/
      struct{
            uint8_t _type;
            uint8_t _code;
            uint16_t _checksum;
      } ip6_icmp_header;

};
```

Pattern string parser module

This module used a simple parser to check and ensure that input pattern string is correct and conform to syntax. Pattern string firstly is decomposed into a stream of tokens (words) and then passed to the parser. If there is any error in syntax, the parser will notify user and exit program.

The operations on string data type is usually expensive and consumes a lot of system resource. To improve the performance, instead of using string, classification criteria and

parameters from pattern strings should be described by appropriate data types, e.g bit pattern or unsigned integer. It is noted that parameters to classification criteria are in two types only, numeric data or IPv6 addresses. Moreover, one classification criteria may have more than one parameter. For example, a pattern string “tcp port 80 8080” matches TCP packets with source/destination port of 80 or 8080. So while parsing pattern string, this module also converts classification criteria in pattern string to following data types.

Data structure represents a list of patterns (*ip6classifier.hh*)

```
/*List of patterns*/
struct filter_types{
      uint16_t output_port;   //output port of packets matching this pattern
      uint16_t type;                //main category of classification
      uint16_t sub_type;            //sub-catergory of classification
      uint16_t sub_sub_type;  //sub-sub catergory of classification
      arguments *list;
      filter_types *next_pattern;
      //Constructor
      filter_types(){
            list = new arguments;
            next_pattern = NULL;
      }

};
```

Data structure represents a list of parameters to classification criteria (*ip6classifier.hh*)

```
/*List of parameters given to classification criteria*/
struct arguments {
      union {
            uint32_t numeric_data;  //numeric parameters
            IP6Address *ip6address; //ip address parameters
      } current_argument;
      arguments *next_argument;

      //Constructor
      arguments(){
            current_argument.ip6address = NULL;
            next_argument = NULL;
      };

};
```

Packet processing module

In this module, incoming packets will be processed according to classification criteria. Switch-case statement is used to emit packets to appropriate output port.

All code and script

Please refer to *ip6classifier.hh* and *ip6classifier.cc* for detail code.

6.1.3. Limitation

Extension header data structure does not support Encapsulating Security Payload header. If a packet has this type of header, it will be ignored.

Even though performance improves significantly by using unsigned integer to represent classification criteria, it would be better if bit pattern were used. Bit operations are always more efficient than unsigned integer operations but it has to trade off by the complexity of the algorithm.

As compared to IPv4 classifier element, IP6Classifier does not support the combination of primitive patterns with the connectives 'and', 'or' and 'not'. It is also not able to understand directives '>', '>=', '<', '<=' for numeric parameters such as port number of ICMP type.

6.2. IP6Fragmenter

Configuration: *IP6Fragmenter(MTU)*

Input port: *01*

Output port: *01*

Processing: *Push*

Description

Expect IPv6 packets as input. If the IP packet length is <= MTU, simply emits the packet on output 0. If the packet size is greater than MTU, IP6Fragmenter splits packet into fragments with size <=MTU and emits to output 0. All IPv6 annotations are copied to the fragments. If there is any error in header, the packet will be discarded.

6.2.1. Implementation

IPv6 Fragment Header

The management of fragmentation in IPv6 is different from that in IPv4. In IPv4, fragmentation is router tasks. For example, when an IPv4 router receive an packet with payload greater than MTU of the link and the don't fragment is not set, it will divide that packet into several fragmented packets and re-transmit to the network.

In IPv6, routers operate in different way. When they receive a packet whose size exceeds MTU of the link, that packet is simply discarded and the source will be informed about this fact (through ICMP packet) and the maximum acceptable packet length. The fragmentation can be implemented only by the source node. Fragment header is only processed by the

destination node and ignored by all intermediate nodes along the path. Fragment header is identified by the value of 44 in the Next Header field.

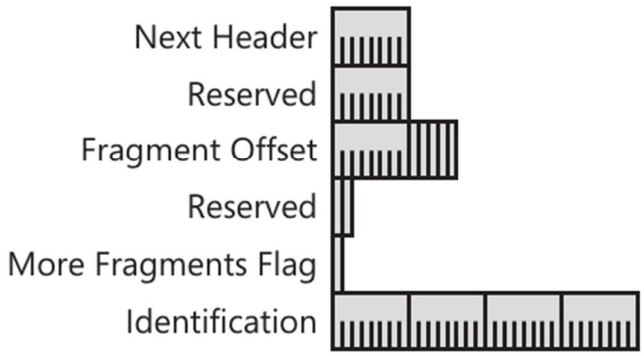

Figure 3. The structure of Fragment Extension Header [5]

- The 8-bit Next Header field: identify the next header extension
- The 8-bit Reserved field: reserved for future uses. It is ignored on reception
- The 13-bit Fragment Offset field: contains the offset of the data following this header relative to the start of the fragmentable part of the original packet (before fragmentation), in 8-octet (64-bit) units.
- The 2-bit Res field: reserved for future uses.
- The 1-bit M field: indicate whether this is the last fragment in packet (value of 1) or not (value of 0)
- The 32-bit Identification field: used to identify all fragments belonging to one packet

Fragment Process

IPv6 packet consists of fragmentable part and unfragmentable part. Unfragmentable part includes IPv6 header plus Hop-by-Hop Options, Destinations and Routing headers. Fragmentable part consits of the rest of the packet. If the packet length excesses MTU of the link, it will be splitted in fragments that are multiple of 8 octets (with exception for the last one). Fragment header is inserted into every fragment and each fragment will be transmitted as separate IPv6 packet. It is the task of destination node to resemble fragments into original packet.

Regarding unfragmentable part, all fields are the same as in original packet except:

- Payload Length field must reflect the new length of fragment
- Next Header field of the last header in unfragmentable part has to be set to 44

All code and script

Please refer to *ip6fragmenter.hh* and *ip6fragmenter.cc* for detail code.

6.2.2 Limitation

Extension header data structure does not support Encapsulating Security Payload header. If a packet has this type of header, it will be ignored.

The resemble element has not been implemented yet.

6.3. IP6HopByHop

Configuration: *IP6HopByHop*

Input port: *01*

Output port: *04*

Processing: *Push*

Description

Expect IPv6 packets as input. If the packet is jumbo packet then push to output port 1, if it is Router Alert packet then emits on output port 2, error jumbo packet emits on output port 3. Default output port is 0 (for the rest)

6.3.1. Hop-by-Hop Header

Figure 4. The structure of Hop-by-Hop Extension Header [5]

The Hop-by-Hop Options header is used to carry optional information that must be examined by every node along a packet delivery path. This type of header must immediately follow the IPv6 header, and its presence is identified by a value zero in the Next Header field of the IPv6 header.

- The 8-bit Next Header field: identify the next header field
- The 8-bit Header Extension Length: the length of Hop-by-Hop Option header in 8 octets (64 bits) unit not including the first 8 octets.
- Option field: variable length and contains one or more options

In Hop-by-Hop, one option that must take into consideration is Jumbo Payload option. Jumbo Payload is identified by Option Type value of 194. It is used to indicate a packet

with payload size that is more than 65, 535 bytes (which is beyond the capacity of 16 bit Payload Length in IPv6 main header). The Jumbo Payload option has the alignment requirement of 4n + 2 (that means the Jumbo Payload option must begin from the byte boundary of 6, 10, 14 and so forth).

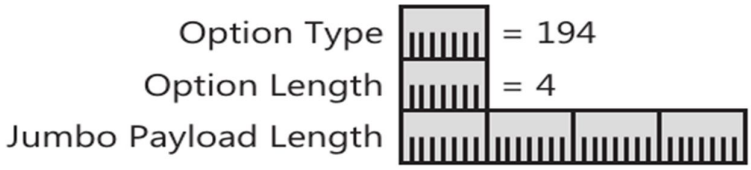

Figure 5. The structure of Jumbo Payload option [5]

Another important type of option in Hop-by-Hop header is Router Alert (with Option Type value 5). It is used to indicate to a router that the contents of the packet require additional processing. The Router Alert option has the alignment requirement of 2n + 0.

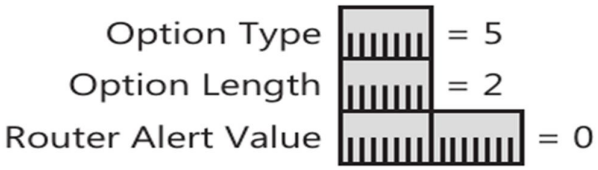

Figure 6. The structure of Router Alert option [5]

6.3.2. Implementation

Please refer to *ip6hopbyhop.hh* and *ip6hopbyhop.cc* for detail code.

6.4. IP6Routing

Configuration: *IP6Routing*

Input Port: *01*

Output Port: *01*

Processing: *Push*

Description

Expect IPv6 packets as input. The element firstly checks whether a packet has Routing header extension or not. If not, it will be left intact and transmitted immediately to output port 0. If the packet has Routing header extension type 0, it will be processed accordingly before being emitted to output. At present, this element only support Routing header type 0.

6.4.1. Routing Header Extension

The Routing Header is identified by a Next Header value of 43. This header supports a function similar to the IPv4 packet Source Route option. The Routing header consists of a Next Header field, a Header Extension Length field (defined in the same way as the Hop-by-Hop Options extension header), a Routing Type field, a Segments Left field that indicates the number of intermediate destinations that are still to be visited, and routing type-specific data.

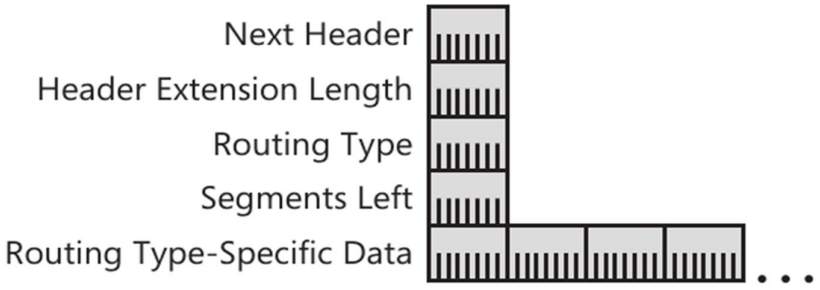

Figure 7. The structure of Routing extension header [5]

There are several types of Routing Header but in this internship only Routing Type 0 for loose

source routing, defined in RFC 2460, is supported.

Routing Type 0 structure:

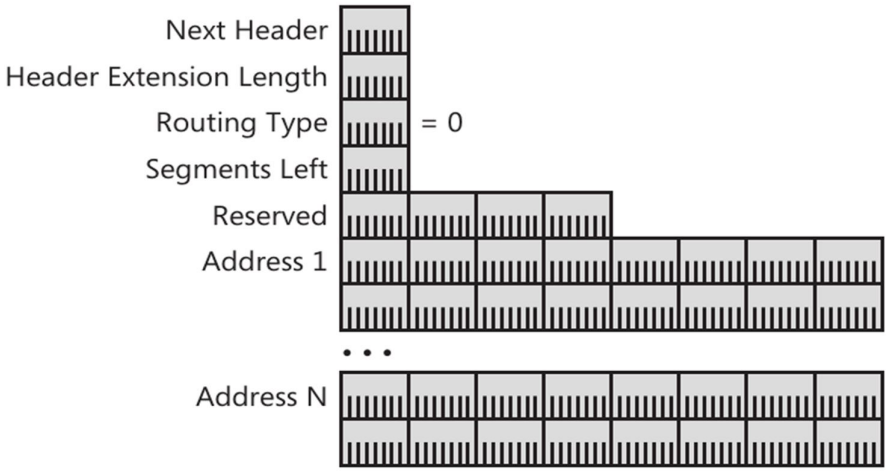

Figure 8. The structure of Routing Type 0 header [5]

- The 8-bit Routing Type field: contains zero value
- The 8-bit Segment Left field: contains the list of intermediate node that this packet has to traverse before reaching destination node. It is noted that the maximum value

of this field is 23.

- The 8-bit Reserved field: is reserved for future uses and ignored on reception.
- The 24-bit Strict/Loose Bit Map field is a mask containing a Strict/Loose bit for each address. This field will be ignored in IP6Routing element.

Address field contains a list of IPv6 addresses of nodes to be traversed along the path to the destination. Intermediate nodes must be visited in exactly the same order.

6.4.2. Implementation

When IP6Routing receives a packet at its input port, it will traverse through all header extensions to check whether Routing extension exists or not. If yes, the Routing header will be processed and the following actions are taken [5]:

1. The current destination address and the address in the (N – Segments Left + 1) position in the list of addresses are swapped, where N is the total number of addresses in the Routing header.

2. The Segments Left field is decremented.

3. The packet is emitted to output port

By the time the packet arrives at the final destination, the Segments Left field has been set to 0 and the list of intermediate addresses visited in the path to the destination is recorded in the Routing header.

All code and script

Please refer to *ip6routing.hh* and *ip6routing.cc* for detail code.

6.4.3. Limitation

The support of Routing Type 0 header is deprecated by Internet Engineering Task Force (IETF) because it exposes some issues in network security. In some OS, incoming packet with Routing Type 0 header will be silently discarded.

Routing Type 2 header used in Mobile IPv6 is not supported.

7. Benchmark IPv4 and IPv6 implementation performance

In this section, performance of elements is measured only by CPU consumption while running a configuration script with different bit rates. The configuration scripts used should be as simple as possible in order to have accurate evaluation.

In some configurations which must use CheckIPHeader/CheckIP6Header to check the validity of incoming packets, CPU resource will not be solely consumed by benchmarked elements. However, test results still provide sufficient information to have conclusion.

7.1. Experiment Model

Although Click is distributed with a few elements capable of generating arbitrary types of packets, these elements themselves consume a lot of system resource such as CPU and internal memory. Therefore, if we use them to benchmark performance of other elements, the result will not be accurate. In such case, it is impossible to specify how much the resource consumption of traffic generator or tested elements are.

To avoid this issue, I set up following topology to separate traffic generation and evaluation processes.

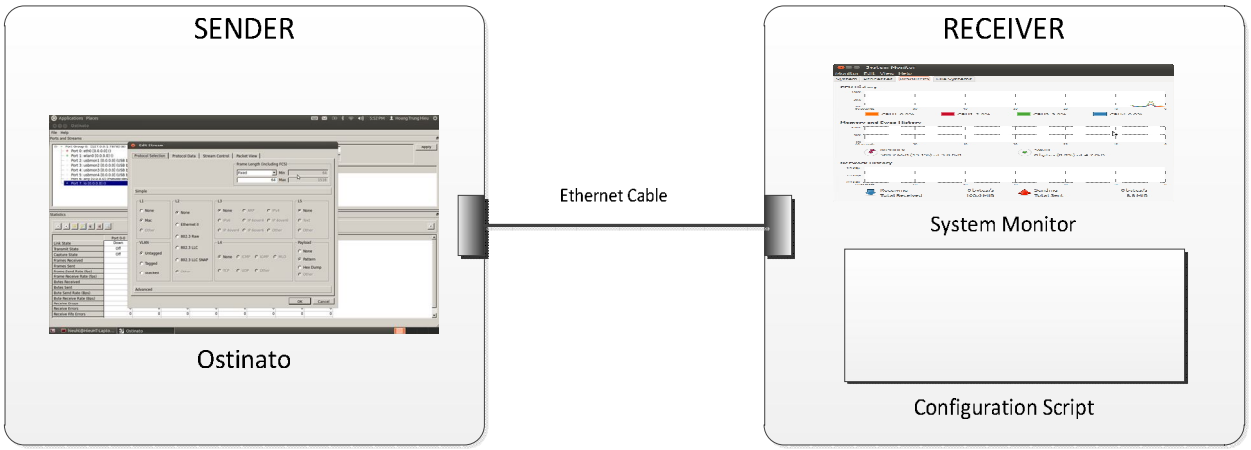

Figure 9. Experiment Model

Two computers are connected directly by network cable. One is responsible for transmitting traffic at different rates by using a third-party software tool, while Click configuration script running in the other receives that packet flow, pushing it to measured element. Performance will be evaluated at receiving computer and the result is not affected by unwanted processes.

Regarding third-party traffic generator, there are several tools available both as open-source or commercial software but in this experiment I decided to use Ostinato [6]. Ostinato is an open-source, cross-platform network packet traffic generator and analyzer. It comes with friendly GUI that makes it easy to build up any type of packet at different bit rates.

To monitor CPU consumption, I used System Monitor package in Ubuntu 12.04 LTS.

7.2. Testbed

As stated earlier, two computers are used for benchmarking IPv4 and IPv6 supporting elements in Click, one for sending data and one for receiving.

The sender, in which traffic generator is installed, has following configuration

- Processor: Intel(R) Core(TM) i5-2410M CPU @ 2.30GHz 2.30GHz
- Installed memory (RAM): 4.00GB
- System type: 64-bit Operating System
- Operating System: Ubuntu 12.04 LTS 64-bit
- Interface card: 1000Mbps LAN

The receiver, in which Click configuration script runs, has following configuration

- Processor: Intel(R) Atom(TM) CPU N455 @1.66Ghz 1.67Ghz
- Installed memory (RAM): 2.00GB
- System type: 32-bit Operating System
- Operating System: Ubuntu 12.04 LTS 32-bit
- Interface card: 100Mbps LAN

As documented in Ostinato user guide, since traffic is generated by software, the maximum bit rate depends on the host computer configuration (CPU, memory…). In order to measure Click elements with sufficient high bit rate, I decided to use the computer with better configuration as sender. In practice, the maximum data rate it can generate is approximately 60Mbps.

7.3. Experiment

Each test is conducted with different theoretical bit rate ranging from 1Mbps to 50Mbps. This is the bit rate generated from Ostinato tool at sender, but while transmitting to receiver, some packets are dropped and it results in the lower bit rate arriving at the receiver. The performance evaluation will be based on this receiving bit rate.

Most of Click elements involve in processing packet header, not the payload so to determine elements' performance, in order to change the bit rate, it is preferable to increase number of packets with small size than simply increase packet size. If not being mentioned explicitly, all test in this section use UDP packet with default size of 64 byte.

7.4.1. CheckIPHeader and CheckIP6Header

Configuration script

The configurations in Figure 10 are used to test CheckIPHeader and CheckIP6Header

elements.

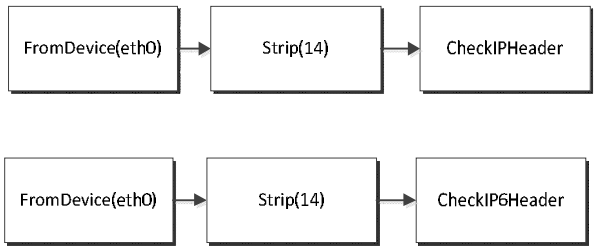

Figure 10. Check IP header configuration script

Test result

Sending Rate (bps)	Receiving Rate (bps)	CPU Consumption (%)	
		IPv4	IPv6
20,000,000	13,600,000	96	100
15,000,000	9,600,000	82	88
12,000,000	8,000,000	82	84
10,000,000	7,000,000	74	80
7,000,000	4,848,000	68	74
5,000,000	3,480,000	48	52
3,000,000	2,088,000	28	28
2,000,000	1,400,000	18	18
1,000,000	696,000	8	8
512,000	357,600	4	4

Table 2. Check IP header performance

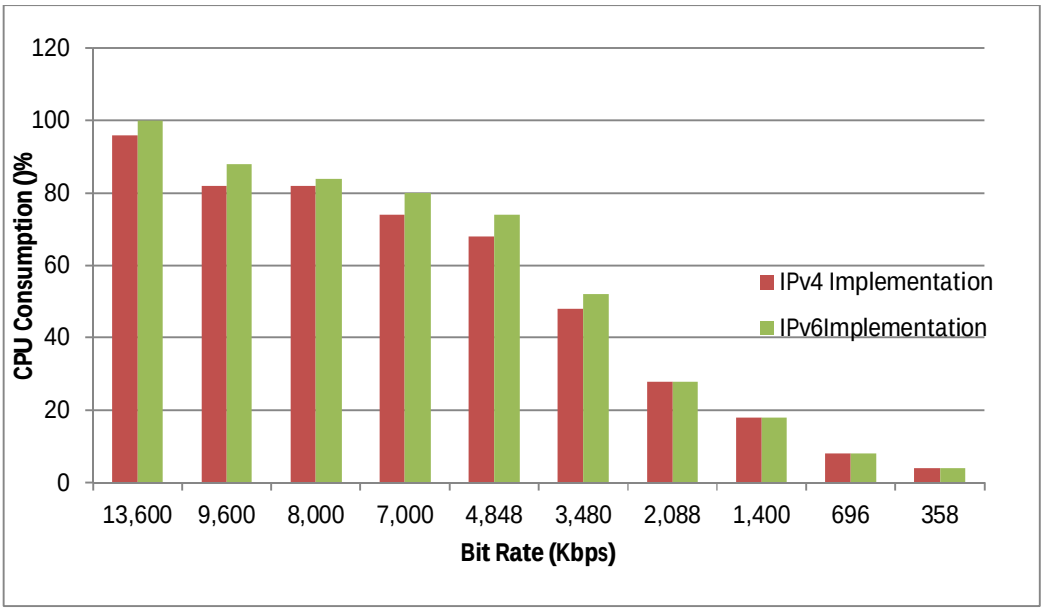

Chart 1. Check IP header performance

Test result shows that at low bit rates, performance of both implementation is the same. However, when the bit rate increases over 3.5Mbps, IPv4 implementation is a bit more efficient than that of IPv6.

7.4.2. IPClassifier and IP6Classifier

Configuration script

The configurations in Figure 11 are used to test the performance of CheckIPHeader and CheckIP6Header elements.

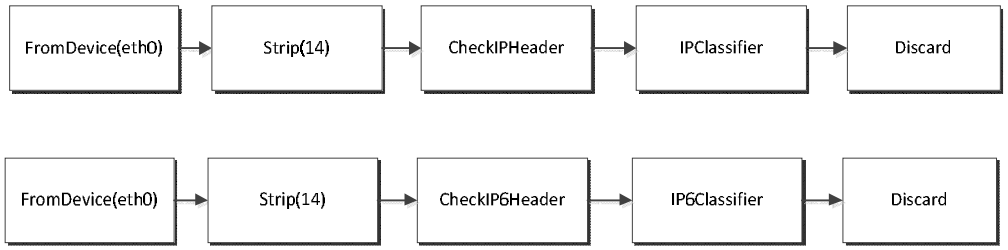

Figure 11. IP classifier configuration script

Test Result

Sending Rate (bps)	Receiving Rate (bps)	CPU Consumption (%)	
		IPv4	IPv6
20,000,000	13,600,000	100	100
15,000,000	9,600,000	98	100
12,000,000	8,000,000	86	98
10,000,000	7,000,000	78	90
7,000,000	4,848,000	72	82
5,000,000	3,480,000	64	70
3,000,000	2,088,000	36	42
2,000,000	1,400,000	24	28
1,000,000	696,000	12	14
512,000	357,600	6	6

Table 3. Classifier performance

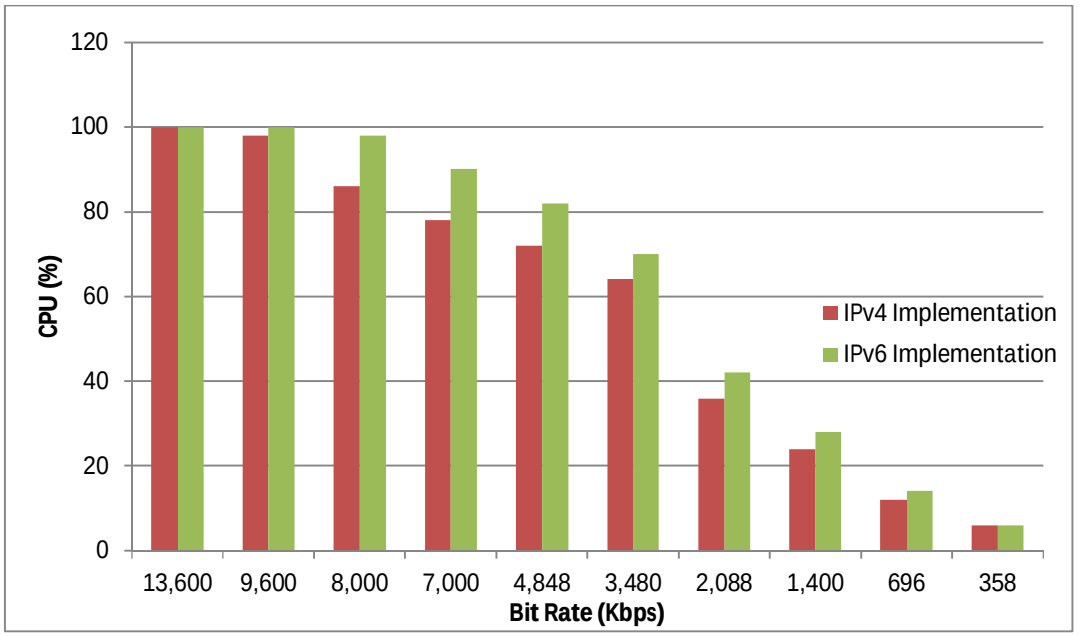

Chart 2. IP classifier performance

The test result shows that IPv4 implementation is more efficient than that of IPv6, especially with bit rate around 7Mbps.

7.4.3. IPFragmenter and IP6Fragmenter

Configuration script

The configurations in Figure 12 are used to test the performance of IPFragmenter and IP6Fragmenter elements.

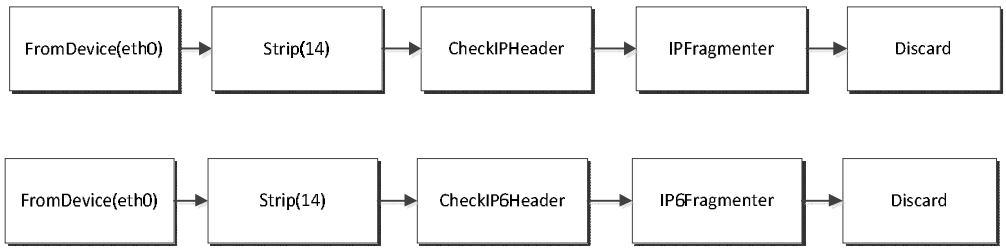

Figure 12. IP fragmenter configuration script

Test result

In this test, 256-byte UDP packet is used and MTU of the link is 100.

Sending Rate (bps)	Receiving Rate (bps)	CPU Consumption (%)	
		IPv4	IPv6
50,000,000	39,200,000	88	100
30,000,000	24,800,000	72	92
25,000,000	22,400,000	60	84
20,000,000	13,600,000	48	74
15,000,000	9,600,000	38	48
12,000,000	8,000,000	30	38
10,000,000	7,000,000	26	32
7,000,000	4,848,000	18	22
5,000,000	3,480,000	12	16
3,000,000	2,088,000	8	12
2,000,000	1,400,000	4	6
1,000,000	696,000	2	4
512,000	357,600	88	100

Table 4. IP fragmenter performance

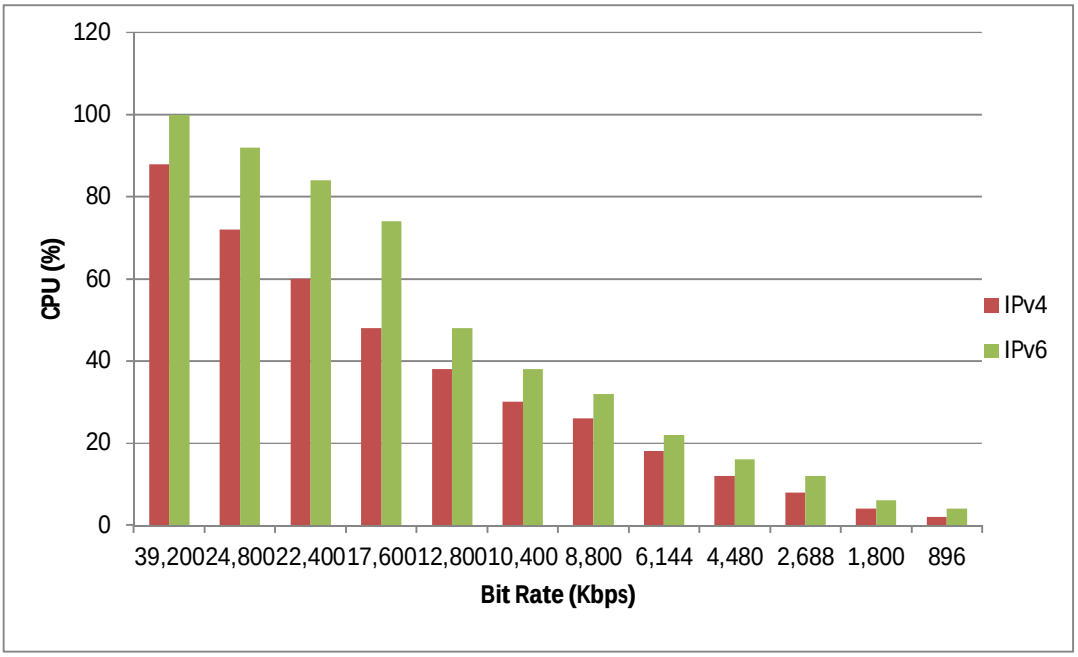

Chart 3. IP fragmenter performance

With all bit rates, IPv4 implementation is significantly more efficient. CPU consumption of IPv4 element is about 20 ~ 30% lower than that of IPv6 element.

7.4.3. SetIPAddress and SetIP6Address

Configuration script

The configurations in Figure 13 are used to test the performance of SetIPAddress and SetIP6Address elements.

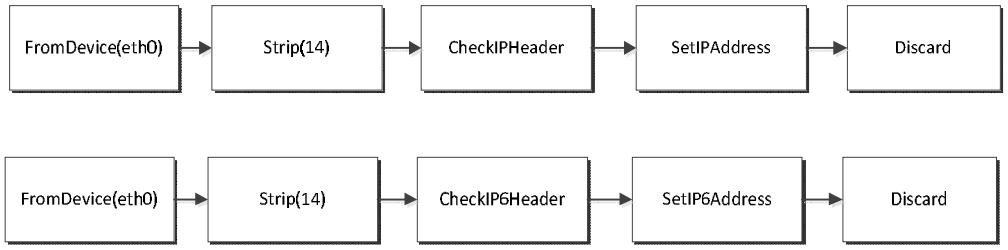

Figure 13. Set IP address configuration script

Test result

Theorical Rate (bps)	Receiving Rate (bps)	CPU Consumption (%)	
		IPv4	IPv6
20,000,000	13,600,000	92	100
15,000,000	9,600,000	88	92
12,000,000	8,000,000	80	86
10,000,000	7,000,000	70	80
7,000,000	4,848,000	64	74
5,000,000	3,480,000	42	46
3,000,000	2,088,000	26	28
2,000,000	1,400,000	18	18
1,000,000	696,000	8	8
512,000	357,600	92	100

Table 5. Set IP address performance

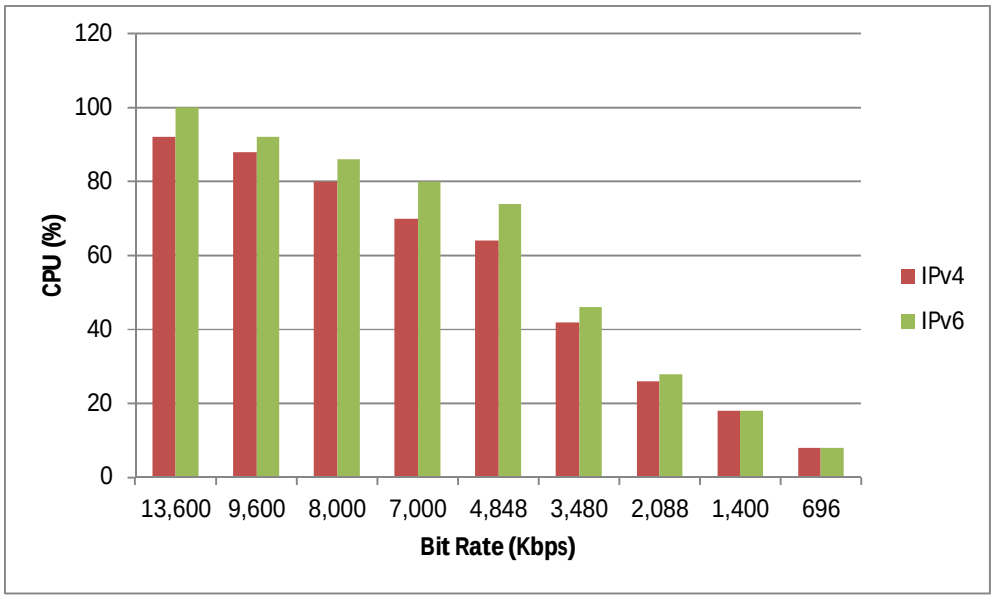

Chart 4. Set IP address performance

7.4.3. MarkIPHeader and MarkIP6Header

Configuration script

The configurations in Figure 14 are used to test the performance of MarkIPHeader and MarkIP6Header elements.

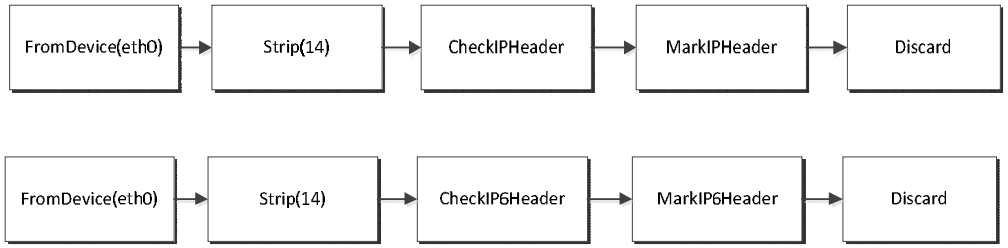

Figure 14. Mark IP header configuration script

Test result

Sending Rate (bps)	Receiving Rate (bps)	CPU Consumption (%)	
		IPv4	IPv6
20,000,000	13,600,000	100	100
15,000,000	9,600,000	90	90
12,000,000	8,000,000	82	86
10,000,000	7,000,000	76	82
7,000,000	4,848,000	70	74
5,000,000	3,480,000	46	50
3,000,000	2,088,000	24	28
2,000,000	1,400,000	18	20
1,000,000	696,000	6	8

Table 6. Mark IP header performance

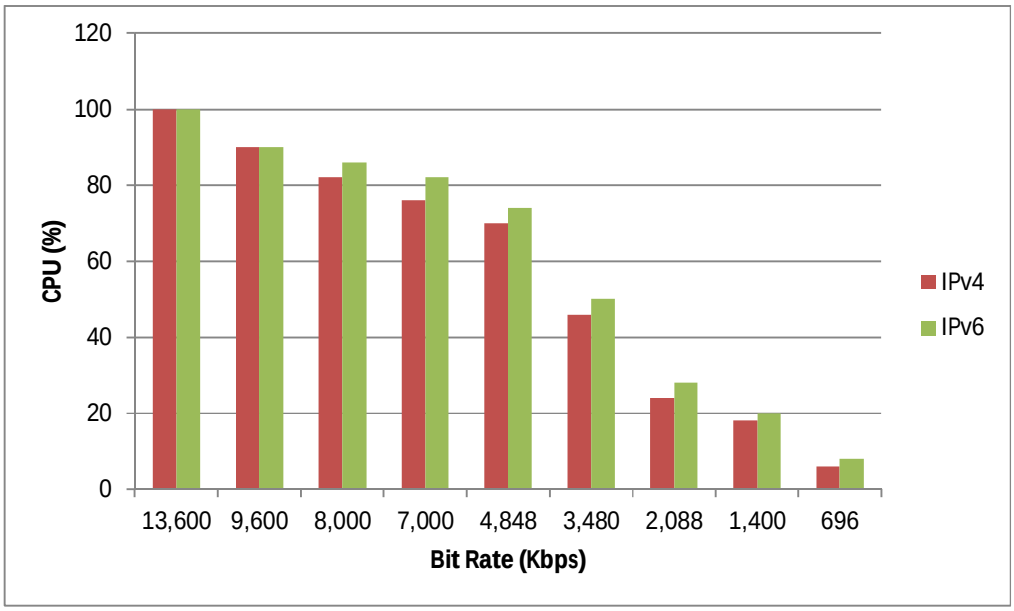

Chart 5. Mark IP header performance

7.4.4. Conclusion

It can be concluded from the test results that with low bit rates (around 1 Mbps) there is no big difference in performance between IPv4 and IPv6 implementation but in higher bit rates, IPv4 elements consume less system resource than corresponding IPv6 elements.

In addition, as compared to the other built-in IPv6 elements in Click (version 2.0.1), new elements written in this internship are less efficient probably because the algorithm and memory allocation currently used are not the most optimal solution.

8. Future works

In future work, I expect to:

- Implement elements supporting ICMPv6 and DHCPv6 functionalities.
- Optimize code in current implementation in order to improve performance
- Benchmark IPv6 elements when they co-work together in more complex configuration and compare to IPv4 implementation.

References

[1] Click Modular Router Homepage. http://www.read.cs.ucla.edu/click/click.

[2] Eddie Kohler. Click Modular Router. PhD thesis at MIT, February 2001.

[3] Felipe Huici. Measuring Click's Forwarding Performance. Department of Computer Science University College London, 2005.

[4] Eddie Kohler. Click for Measurement. UCLA Computer Science Department Technical Report, 2006.

[5] Joseph Davies. Understanding IPv6, 2nd Edition. Microsoft Press, 2008.

[6] Ostinato Project Homepage. http://code.google.com/p/ostinato/

[7] Sample tutorials by the PATS group. http://www.pats.ua.ac.be/software/click/