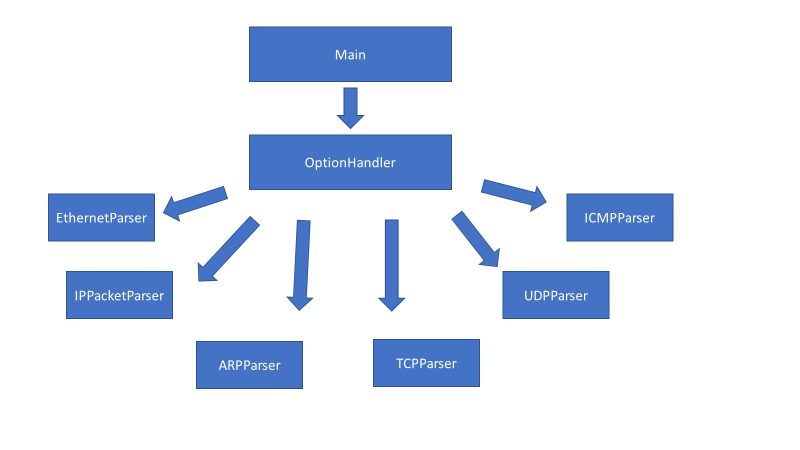
**Intro:**

In this project both a packet parser and a packet generator were created using Java. The packet parser was built using a modular approach, the top part being Main. Main instantiates the real top class which is called OptionHandler which runs the user’s options. The OptionHandler instantiates a class for each different kind of header parsing. In all 6 classes were instantiated ARParser, UDPParser, TCPParser, ICMPParser, IPPacketParser, and EthernetParser. For packet generation a class was created that is called PacketGenerator.

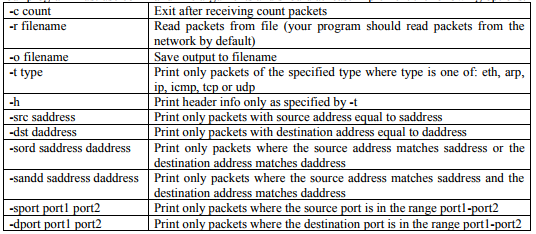
Packet Parser Hierarchy



**OptionHandler:**

OptionHandler receives the user’s options from main and using the library org.apache.commons.cli the options are parsed. All the viable options consist of the following:

Packet Parsing Options



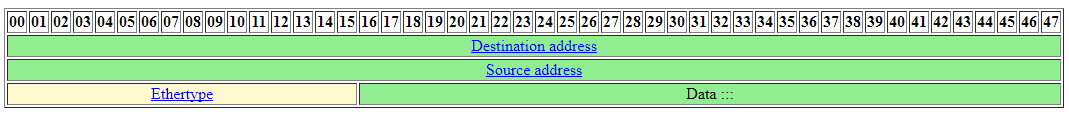
In addition, the options can be viewed from the program by passing the -help option. The function that parses the options is called parseOptions. When parsing options the function getArgument is used to retrieve the arguments passed with the options. Once the options have been parsed and the arguments retrieved the runOptions function is ran. If the user does not pass in a file to read packets from, the computer is checked for network interfaces, it will then prompt the user to choose from one. If a read file is provided the class will open it up and begin looking for bytes to parse. When no output file is provided the default will be set which is to print output to the console. If an output file is specified, the output will be sent to the specified file.

A switch statement is used to parse for diverse types of packets. When the user does not specify what kind of packet to parse for, the defaults is set to TCP packet parsing. At the beginning of all the packet parsing switch statements, the counter option is checked to see if it has been specified. If it has then a count is kept of every packet that has been parsed with the specified header, if it hasn’t then packets will continue to be parsed indefinitely until the user types ctrl + C to close the program. Based on whether the header only option is set, the output of the parsing will be the header of the specified type only, or the header plus the payload of the type. The only exception is arp packets since they do not have a payload only their headers will be printed. When parsing ip tcp, udp, and icmp packets the address is checked to see if the packet has the right address. If it does then it is printed, if not then it is ignored. The function that does this checking is called checkIPAddressFilter. When parsing tcp, and udp packets, the ports are also checked for filtering. The function that does this is called checkPortRange.

The packets are retrieved though a function that keeps track of whether they are coming from a file or a network interface card named getPacket. If the type is set to ethernet, the length of the packet is checked for a minimum length of 14, then parsed with the EthernetParser instance. If the type is set to arp, the length is checked for a minimum length of 41. The ethernet is parsed and the type of packet is retrieved, and compared to 0x0806 which means that it is arp. The packet is then passed through the ARPParser instantiation. If the type is set to ip, the length is checked for a minimum of 33. The ethernet is parsed, and the type is then compared to 0x0800 which means that the packet type is ip. The packet is then passed through the IPPacketParser instantiation. If the type is set to icmp, the length of the packet is checked for a minimum length of 41. The ethernet is parsed and the type checked for ip. An ip packet would then be parsed and the protocol compared to the decimal value 1, which means that it is an icmp packet. An icmp packet would then be passed through the ICMPParser instantiation. If the type is set to tcp, the length is checked for a minimum of 53. The ethernet is parsed and the type checked for ip. An ip packet would then be parsed and the protocol compared to the decimal value 6, which means that it is a tcp packet. A tcp packet would then be passed through the TCPParser instantiation. If the type is set to udp, the length is checked for a minimum length of 41. The ethernet is parsed and the type is checked for ip. An ip packet would then be parsed and the protocol compared to the decimal value 17, which means that it is a udp packet. A udp packet would then be passed through the UDPParser instantiation.

**EthernetParser:**

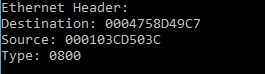
Ethernet Packet



<http://www.networksorcery.com/enp/protocol/IEEE8023.htm>

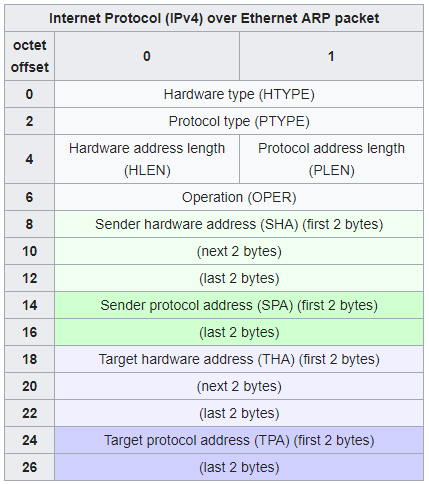
Ethernet packets are parsed according to the above image. The EthernetParser class contains a pair of variables for each different part of the header, except for payload which only has one variable representing it. The first variable stores the value as a byte array, while the second converts it to a string. The conversion to string maintains the value in hex. The payload is stored as a byte array copying the packet that was parsed from index 14 to the end of the array. The class also contains getters to retrieve every part of the header as either bytes, or a string. Furthermore, two functions named printAll, and printHeaderOnly are defined for printing. The payload is converted into a UTF-8 encoding.

Parsed Ethernet Header Example



**ARPParser:**

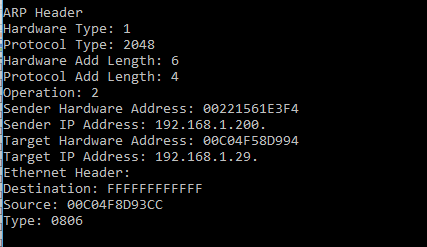
ARP Packet



<https://en.wikipedia.org/wiki/Address_Resolution_Protocol>

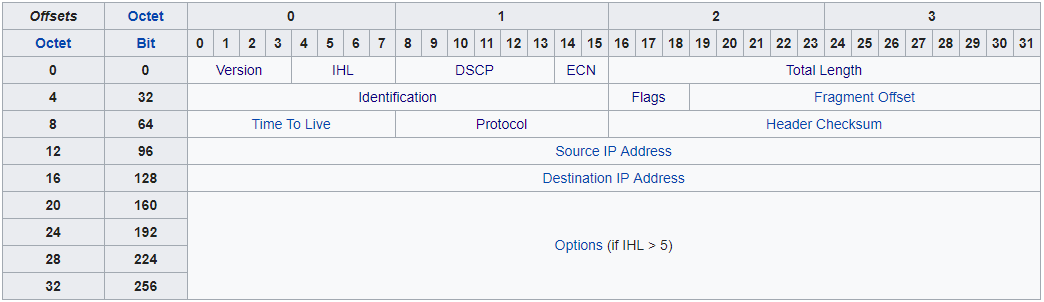
ARP packets are parsed according to the above image. The ARPParser class contains a pair of variables for each different part of the header. The first variable stores the value as either a byte array or integer, while the second converts it to a string. All the values except for the MAC addresses are converted into decimal. When converting values from bytes to integer, a bit wise and operation with 0xFF was performed with the retrieved byte. This would get rid of the sign extension that Java does by default. When retrieving multi byte values, a bit wise shift of 8 is done, and the next retrieved converted byte is added. The class also contains getters to retrieve every part of the header as a string. Furthermore, the functions named printAll is defined for printing.

Parsed ARP Header Example



**IPPacketParsing:**

IP Packet

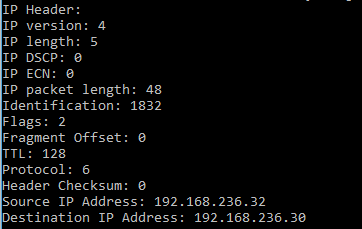


<https://en.wikipedia.org/wiki/IPv4>

IP packets are parsed according to the above image. The IPPacketParser class contains a pair of variables for most of the different parts of the header. The exception is version, ihl, dscp, ecn, and payload. The first four are converted to strings, while payload is kept as a byte array. The first variable stores the value as an integer, while the second converts it to a string. When converting values from bytes to integer, a bit wise and operation with 0xFF was performed with the retrieved byte. This would get rid of the sign extension that Java does by default.

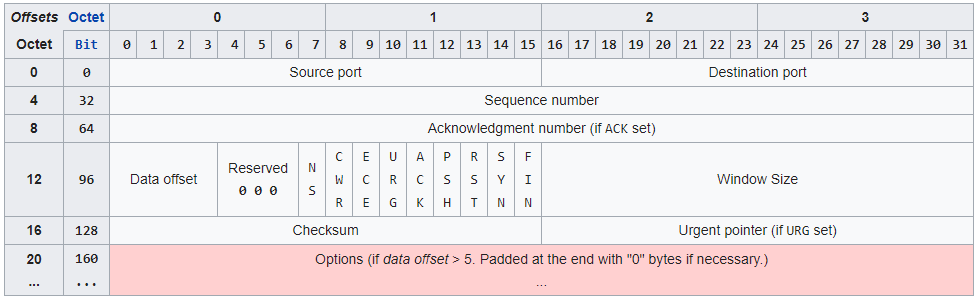
When retrieving multi byte values, a bit wise shift of 8 is done, and the next retrieved converted byte is added. When retrieving values that were less than a byte long, a masking technique was used. The class also contains getters to retrieve every part of the header as a string. The payload is stored as a byte array copying the packet that was parsed from index 34 to the end of the array. The payload is converted into a UTF-8 encoding. Furthermore, two functions named printAll, and printHeaderOnly are defined for printing.

Parsed IP Header Example



**TCPParser:**

TCP Packet

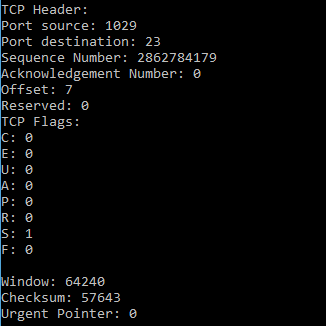


<https://en.wikipedia.org/wiki/Transmission_Control_Protocol>

TCP packets are parsed according to the above image. The TCPParser class contains a pair of variables for most of the different parts of the header, except for the payload, and the flags which are all stored in one variable. The first variable stores the value as either an integer or a long, while the second converts it to a string. When converting values from bytes to integer, a bit wise and operation with 0xFF was performed with the retrieved byte. This would get rid of the sign extension that Java does by default.

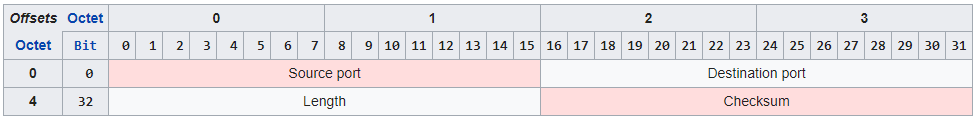
When retrieving multi byte values, a bit wise shift of 8 is done, and the next retrieved converted byte is added. When retrieving values that were less than a byte long, a masking technique was used. When parsing flags, the aggregation of all the flags is stored in one variable, the string representation has the different types of flags parsed. The class also contains getters to retrieve every part of the header as a string. The payload is stored as a byte array copying the packet that was parsed from index 54 to the end of the array. The payload is converted into a UTF-8 encoding. Furthermore, two functions named printAll, and printHeaderOnly are defined for printing.

Parsed TCP Header Example



**UDPParser:**

UDP Packet

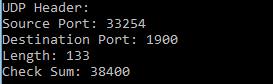


<https://en.wikipedia.org/wiki/User_Datagram_Protocol>

UDP packets are parsed according to the above image. The UDPParser class contains a pair of variables for most of the different parts of the header, except for the payload. The first variable stores the value as an integer, while the second converts it to a string. When converting values from bytes to integer, a bit wise and operation with 0xFF was performed with the retrieved byte. This would get rid of the sign extension that Java does by default.

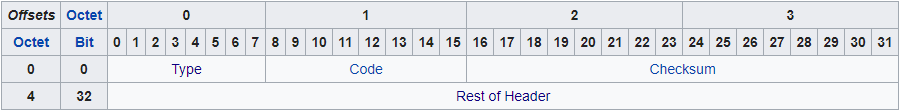
When retrieving multi byte values, a bit wise shift of 8 is done, and the next retrieved converted byte is added. The class also contains getters to retrieve every part of the header as a string. The payload is stored as a byte array copying the packet that was parsed from index 42 to the end of the array. The payload is converted into a UTF-8 encoding. Furthermore, two functions named printAll, and printHeaderOnly are defined for printing.

Parsed UDP Header Example



**ICMPParser:**

ICMP Packet

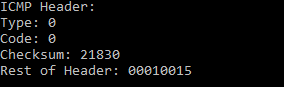


<https://en.wikipedia.org/wiki/Internet_Control_Message_Protocol>

ICMP packets are parsed according to the above image. The ICMPParser class contains a pair of variables for most of the different parts of the header, except for the payload. The first variable stores the value as either an integer or a byte array, while the second converts it to a string. When converting values from bytes to integer, a bit wise and operation with 0xFF was performed with the retrieved byte. This would get rid of the sign extension that Java does by default. When retrieving multi byte values, a bit wise shift of 8 is done, and the next retrieved converted byte is added.

The “Rest of Header” part of the icmp packet is maintained as byte array, and a getter to retrieve it as such is provided. This is to facilitate further processing that this project doesn’t do but might be needed in a later project. When retrieving multi byte values, a bit wise shift of 8 is done, and the next retrieved converted byte is added. The class also contains getters to retrieve every part of the header as a string. The payload is stored as a byte array copying the packet that was parsed from index 42 to the end of the array. The payload is converted into a UTF-8 encoding. Furthermore, two functions named printAll, and printHeaderOnly are defined for printing.

Parsed ICMP Header Example



**PacketGenerator:**

The PacketGenerator class starts off by scanning for network interface cards, and once found it will prompt the user to choose one. After a NIC has been chosen the user will be prompted to enter the name of a file that has the packets to be sent. The class will then retrieve all the packets and put them on the network.

**Lab Packet Retrieval:**

1. Client IP, username and password of the telnet session to 192.168.1.22

**IP: 192.168.1.66, Username: group15, Password: 192.168.1.66**

1. Client IP of the failed ftp logon session to 192.168.1.66

**IP: 192.168.1.62**

1. Client IP, username, password and name of the file transferred to 192.168.1.42

**IP: 192.168.1.42, Username: group14, Password: 192.168.1.62**

1. Client IP, name and content of the html file transferred by 192.168.1.10

**IP: 192.168.1.22, Name: cs7493,**

**Content: <html><body>hey, Congrats, you've found our web page!</body></html>**

1. IP address of hosts iodine and hydrogen as returned by DNS servers at 192.168.1.14 and 192.168.1.46.

**Iodine IP: 192.168.1.62, Hydrogen IP: 192.168.1.10**

1. Send an ARP request to 192.168.1.200, capture the ARP reply with your parser and report the MAC address in your report

**MAC address: 00221561E3F4**

ARP Request in Bytes:

**ff ff ff ff ff ff 50 48 49 4f 4e 43 08 06 00 01**

**08 00 06 04 00 01 00 C0 4F 58 D9 94 c0 a8 01 1d**

**00 00 00 00 00 00 c0 a8 01 c8**

**Compiling and Running:**

**Windows:**

**Compiling:**

javac -cp ".;commons-cli-1.4.jar" -d . Main.java

**Running:**

java -cp ".;commons-cli-1.4.jar" Main

**Linux:**

**Compiling:**

javac -cp ".:commons-cli-1.4.jar" -d . Main.java

**Running:**

java -cp ".:commons-cli-1.4.jar" Main