Spring Semester 2020-2021

A Report for CSN-361 (Computer Network Lab)

Submitted by

Shubhang Tripathi (18114074)

stripathi1@cs.iitr.ac.in

Department of Computer Science and Engineering

Indian Institute of Technology (IIT) Roorkee

May 28, 2021

Contents

1 Problem Statement 1

1.1 Question

Write a C++ program to print the MAC address of your computer, the host name and the IP address of your computer.

1.2 Theory

1.2.1 MAC Address

Hosts and routers don't have link-layer addresses, instead the network interfaces (the adapters) have link-layer addresses. A host with multiple adapters has multiple such addresses. This link-layer address is known as a **LAN address** or a **physical address**, or more popularly a **MAC address**. For LANs including 802.11 WLANs and ethernet, this MAC address is of length 6 bytes, where each byte is represented as a hexadecimal value with ':' as a separator. This address is unique to the adapter i.e no other adapter has the same MAC address. This address is fixed, i.e does not change no matter which network one connects to.

1.2.2 Hostname

A hostname is an identifier (or label) for a device in a network, which is used to identify the device in some forms of network communication. This is used to distinguish devices on a local network.

1.2.3 IP Address

IP is the network layer protocol in the internet. An IP address is associated with an interface, i.e. if a host were to be on two different networks, it would have 2 IP addresses, but usually hosts are only on 1 network and thus have one IP address.

Each IP address is 32 bits (or 4 bytes) long, therefore it can on one of the 2^{32} possible values. These are usually written in the **dotted-decimal-notation**, in which each byte is written in decimal form, with each byte being separated by a '.'.

1.3 Implementation Details

1.3.1 Hostname

For getting hostname of current machine, the **gethostname** function is used, which is part of the C standard library. The function signature is:

```
int gethostname(char *name, size_t len)
```

This we just allocate a small buffer, and pass a reference to it along with it's length to this function which sets the hostname in our passed buffer.

1.3.2 MAC and IP address

As mentioned in the previous section, both the **MAC address** and the **IP address** belong to a particular network interface. Since these are managed by the operating system, hence the responsibility for providing APIs to access information related to these is also operating system specific. In my case, I have done the implementation using the linux API **getifaddrs**. The signature for this function is:

```c
int getifaddrs(struct ifaddrs **ifap);
```

According to the linux man pages, the desciption of the function is:

> The getifaddrs() function creates a linked list of structures describing the network interfaces of the local system, and stores the address of the first item of the list in *ifap.

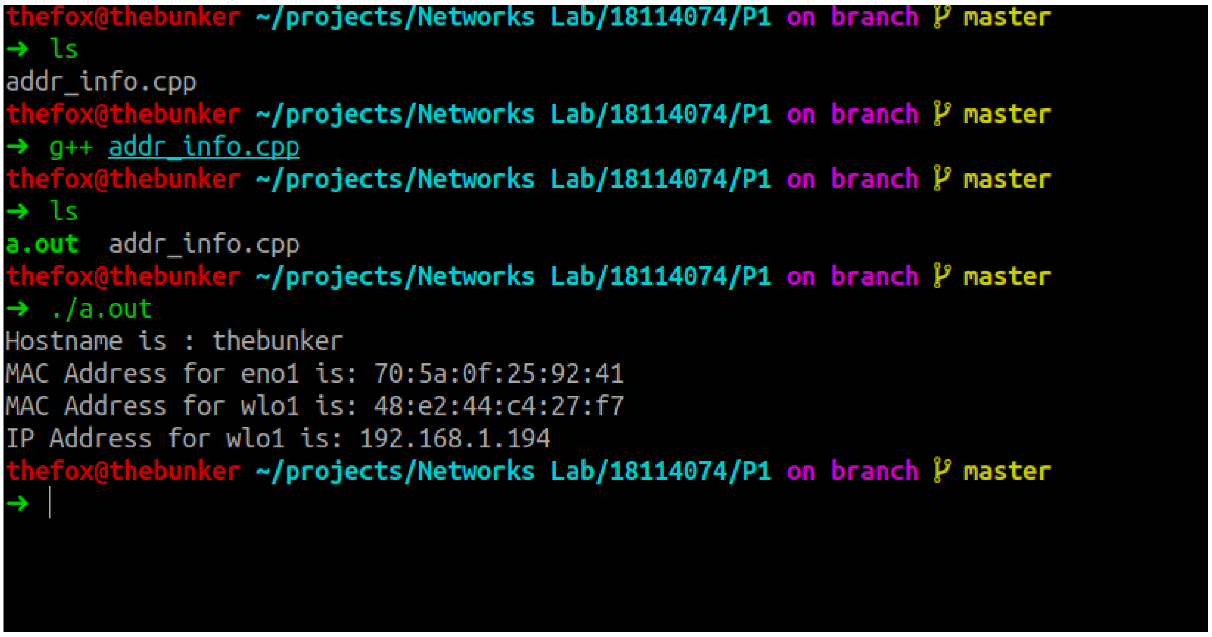
```c
struct ifaddrs {
    struct ifaddrs  *ifa_next;    /* Next item in list */
    char            *ifa_name;    /* Name of interface */
    unsigned int     ifa_flags;   /* Flags from SIOCGIFFLAGS */
    struct sockaddr *ifa_addr;    /* Address of interface */
    .
    .
    .
};
```

Thus, we can iterate over the list using the '**ifa_next**' field. We can find the name of the interface using '**ifa_name** ' field. For the actual address, we use the '**ida_addr**' field. That itself is a structure of type '**sockaddr**' which has a field '**sa_family**'. We can use this to determine if the current structure holds information for an IP address ('**AF_INET**') or a MAC address ('**AF_PACKET**'). Then, depending on the case, we can cast this addess into what we need (sockaddr struct has a data field which contains data specific to the struct being typecasted to):

- **sockaddr_in**: in case of an IP address. This can give access to a field '**sin_addr**', which when passed to the '**inet_ntoa**' function, will return the IP address in the dotted decimal notation.
- **sockadd_ll**: in case of a physical / MAC address. This gives access to '**sll_addr**' field, which holds the raw bytes corresponding to the MAC address. We can manually print these in the usual notation.

1.4 Results

The program is successfully able to print the hostname (as is also shown in the shell prompt after the @ symbols). It also prints the MAC addresses for the **eno1** interface (the ethernet interface) and **wlo1** interface (the WiFi interface). Since my computer was also connected to the WiFi, it also shows the IP address assigned to that interface.



```
thefox@thebunker ~/projects/Networks Lab/18114074/P1 on branch ᚠ master
➜ ls
addr_info.cpp
thefox@thebunker ~/projects/Networks Lab/18114074/P1 on branch ᚠ master
➜ g++ addr_info.cpp
thefox@thebunker ~/projects/Networks Lab/18114074/P1 on branch ᚠ master
➜ ls
a.out  addr_info.cpp
thefox@thebunker ~/projects/Networks Lab/18114074/P1 on branch ᚠ master
➜ ./a.out
Hostname is : thebunker
MAC Address for eno1 is: 70:5a:0f:25:92:41
MAC Address for wlo1 is: 48:e2:44:c4:27:f7
IP Address for wlo1 is: 192.168.1.194
thefox@thebunker ~/projects/Networks Lab/18114074/P1 on branch ᚠ master
➜ |
```

Figure 1: Results of the program for Problem Statement 1

2 Problem Statement 2

2.1 Question

Write a socket program in Java for PING command.

2.2 Theory

2.2.1 Internet Control Message Protocol(ICMP)

ICMP is used to communicate network-layer information between hosts and routers. Mostly this is used for error reporting. ICMP messages have a type and a code field. A particular combination of these is used for a particular type of message. Some of these are:

Table 1: ICMP message types

ICMP Type	Code	Description
0	0	echo reply (to ping)
3	0	destination network unreachable
3	2	destination protocol unreachable
.	.	.
.	.	.
.	.	.
8	0	echo request
.	.	.
.	.	.
.	.	.

The **PING** command mentioned in the question sends as ICMP type 8 code 0 message to a host. When the destination receives this request, it sends a type 0 code 0 message back. Usually, any TCP/IP implementation supports the **PING** server in the operating system itself. In order to send these ICMP messages, which are lower than TCP/UDP in the network stack, one needs to create a lower level packet, which can only be sent by a root / administrator user. The **ping** binary works for all users since it is a **setuid** binary, thus, it runs with root priviledges no matter the user who invoked the command.

2.3 Implementation Details

2.3.1 Socket Creation

The program takes input in the form of a command line arguement. The program checks if this is given or not, exiting with an error and displaying the help menu if needed. In order to send the low-level packets as mentioned above, the program creates a socket using the **SOCK_RAW** option, which creates a socket to provide raw access to the network protocol. The program also specifies the protocol to be ICMP. The program then sets the timeout value for the socket to 2 seconds using the **setsockopt** function.

2.3.2 Sending packets

The program uses the arguement passed in, as the hostname for the receiving machine. This is passed to the **gethostbyname** function. This function return a host entry corresponding to the hostname passed to it. The return value for this function is of type **struct hostent***.

The program uses the h_addr field in this structure to get the corresponding bytes for the address of the receiving machine, which is used in the **sockaddr_in** structure for the receiving machine. The program then goes into a loop where it creates a packet using the **icmp** structure, setting the required fields like:

- **icmp_type** to **ICMP_ECHO**.
- **icmp_id** to the pid of the current process.
- **icmp_seq** to the loop counter.
- **icmp_code** to 0, as mentioned in the table[1]
- **icmp_cksum** to the checksum of the structure. This is a standard 16 bit 1's complement arithmetic based checksum, whose algorithm is discussed in section 5.2.2

Then a time measurement is made using the **std::chrono** interface, which uses the system clock to give time as a floating point value for better precision (compared to the **time** function which gives number of seconds as an integer). This is used later to calculate the ping duration.

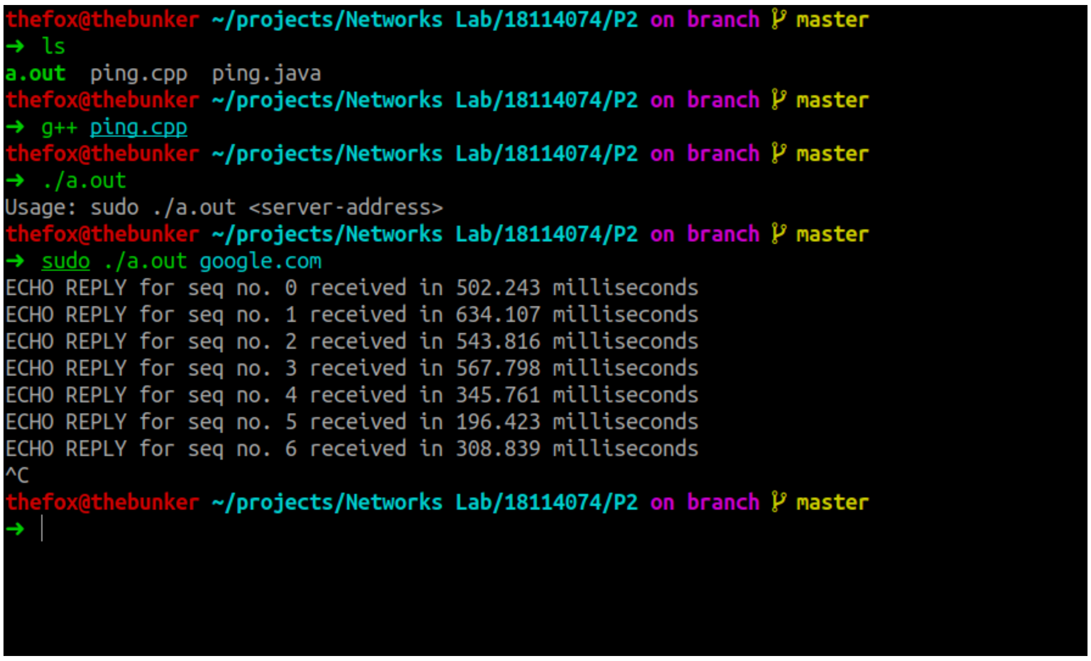
Then, to actually send the packet to the destination, the **sendto** api is used with the packet crafted and server address info found previously.

2.3.3 Receiving packets

The packets are received using the **recvfrom** api. If a receive was successful, a time measurement is made again. Each of these is first stripped off from it's ip header, then the remainder is assigned to an **icmp** structure, similar to the one used for sending the ICMP_ECHO packet. This packet's icmp_type field is checked with ICMP_ECHOREPLY, in which case, the ping was successful, and the program then prints the required statistic of sequence number and ping time.

2.4 Results

Just like the **ping** program, this program is successfully able to send the required ICMP_ECHO packets and receive ICMP_ECHOREPLY packets. As mentioned earlier, the program requires superuser priviledges to send these packets at a low level, thus it needed to be run using the **sudo** command.



```
thefox@thebunker ~/projects/Networks Lab/18114074/P2 on branch ᚠ master
➜ ls
a.out  ping.cpp  ping.java
thefox@thebunker ~/projects/Networks Lab/18114074/P2 on branch ᚠ master
➜ g++ ping.cpp
thefox@thebunker ~/projects/Networks Lab/18114074/P2 on branch ᚠ master
➜ ./a.out
Usage: sudo ./a.out <server-address>
thefox@thebunker ~/projects/Networks Lab/18114074/P2 on branch ᚠ master
➜ sudo ./a.out google.com
ECHO REPLY for seq no. 0 received in 502.243 milliseconds
ECHO REPLY for seq no. 1 received in 634.107 milliseconds
ECHO REPLY for seq no. 2 received in 543.816 milliseconds
ECHO REPLY for seq no. 3 received in 567.798 milliseconds
ECHO REPLY for seq no. 4 received in 345.761 milliseconds
ECHO REPLY for seq no. 5 received in 196.423 milliseconds
ECHO REPLY for seq no. 6 received in 308.839 milliseconds
^C
thefox@thebunker ~/projects/Networks Lab/18114074/P2 on branch ᚠ master
➜ |
```

Figure 2: Results of the program for Problem Statement 2

3 Problem Statement 3

3.1 Question

Implement an error detection mechanism using the standard CRC algorithm. Write two programs: generator and verifier. The generator program reads from standard input an n-bit message as a string of 0's and 1's as a line of ASCII text. The second line is the k-bit polynomial, also in ASCII. It first checks that the polynomial is not divisible by x and x+1. If it is divisible by x or x+1, it outputs error else it outputs to standard output a line of ASCII text with n+k-1 0's and 1's representing the message to be transmitted. Then it outputs the polynomial, just as it read it in. The verifier program reads in the output of the generator program (if output is not error) and outputs a message indicating whether it is correct or not. Finally write a program, alter, that inverts one bit in the first line of the output of the generator depending on its argument, but copies rest of the first line and second line correctly. Now type the following and report the outcome.

(i) generator < file | verifier
(ii) generator < file | alter arg | verifier

3.2 Theory

3.2.1 Block Coding

In block coding, a message is broken up into blocks, each of k bits, called datawords,then r parity bits are added to it. Then the new block length $n = k + r$. These new n-bit blocks are called codewords. The process of block coding is an injective mapping i.e. a particular dataword is always encoded as the same codeword.

3.2.2 Linear Block Codes

Linear Block Code are a subset of block codes with the property that XOR of any two valid codewords is another valid codeword.

3.2.3 Cyclic Codes

Cyclic codes are linear block with a property that if a valid codeword is cyclically shifted, then the result is also a valid codeword.

3.2.4 Cyclic Redundancy Check (CRC)

Cyclic Redundancy check (CRC) is a category of cyclic codes used in networks for error detection.

Generator: At the encoder site, we have a k bit dataword. The sent codeword has n bits. The dataword is modified by adding $n - k$ 0s to the right side of the word. Then this modifed word is passed to the **generator**. This uses a divisor of size $n - k + 1$, which is agreed upon by both the sender and the receiver. The generator then divides (modulo-2) the modified dataword with the divisor. The remainder of this division is appended to the original dataword and sent to the receiver.

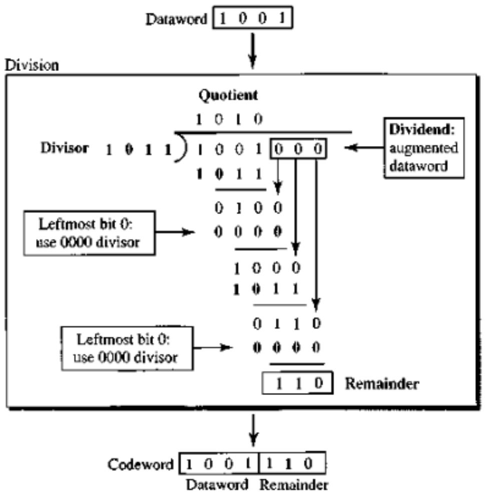
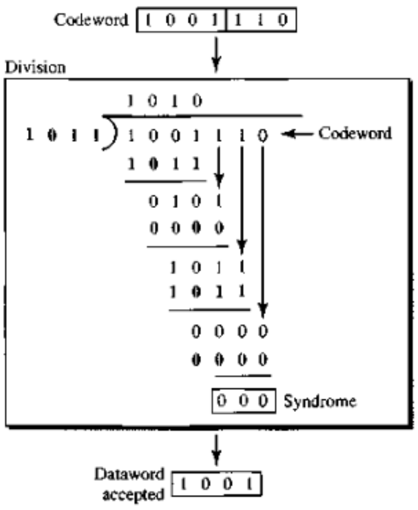
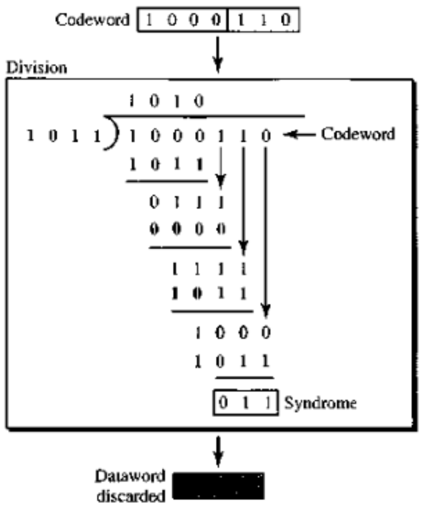


Figure 3: Working of the the generator

Verifier: At the receiver end, the codeword (possibly corrupted) is received by the decoder. A copy of the codeword is given to the checker. This checker is similar to the generator in that it uses the agreed upon divisor along with the codeword for modulo-2 division. The remainder is then fed to a logic analyzer, which checks if the remainder is all 0s. If that is the case, the last $n - k$ bits from the codeword are removed, and the rest is accepted. If that does not happen, then this codeword is discarded.



(a) Working of decoder with correct codeword



(b) Working of decoder with incorrect codeword

3.2.5 Polynomials

Cyclic Codes can also be represented as polynomials. A bit pattern can be represented using a polynomial where the power of each term shows the position of the bit, the coefficient shows the value of that bit. The polynomial can also be written in a shorter notation, by ignoring terms with 0 as the coefficient.

For example, the divisor

$$1010101$$

can be written as

$$1x^6 + 0x^5 + 1x^4 + 0x^3 + 1x^2 + 0x^1 + 1x^0$$

which can further be shortened to

$$x^6 + x^4 + x^2 + 1$$

3.3 Implementation Details

3.3.1 Generator and Verifier

Both the generator and verfier programs first check if the inputs received by them are binary strings or not, it not, they report the error and exit. Also, as mentioned in the problem statement, there is also a check to see whether the polynomial (divisor) is divisible by x or $x + 1$. Since this means that x or $x + 1$ should not be a root of the polynomial, therefore, we just evaluate the polynomial for these values i.e. $x = 0$ and $x = -1$ respectively. For $x = 0$, we just see if the coefficient of x^0 i.e. the constant term is 0 or not. For $x = -1$, we sum the coefficients, but every alternate coefficient is multiplied by -1.

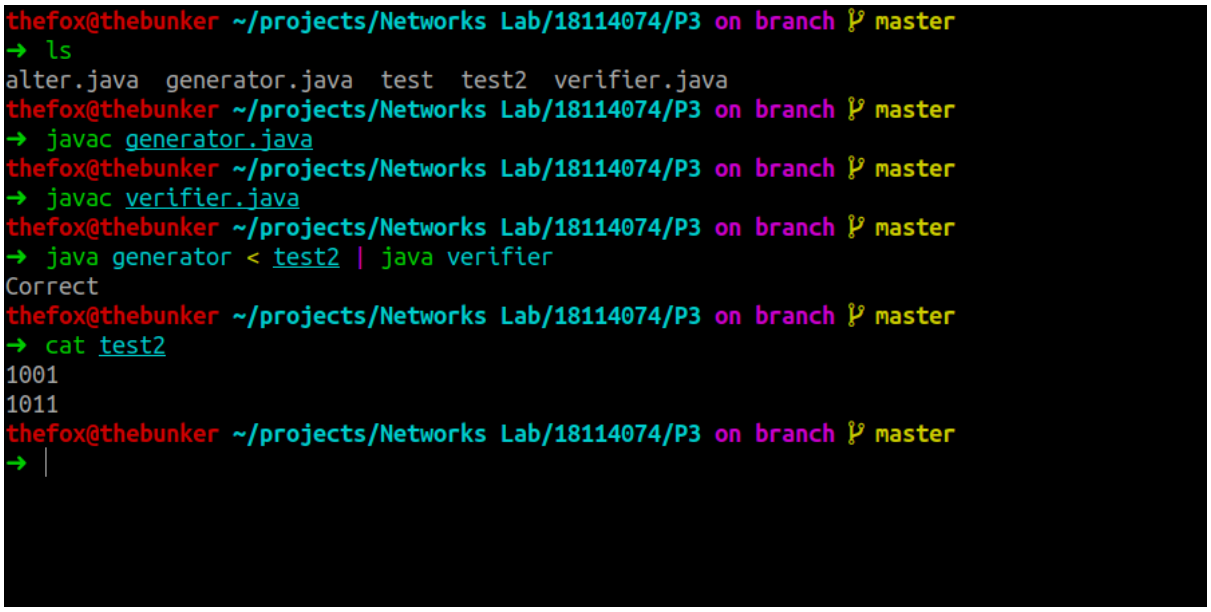
After this initial checkup is done, the generator program appends $n-k+1$ 0s to the dataword (input message). After this the process of division is common for the generator and verifier. A for loop is used to iterate over the first n bits of the dividend(codeword/augmented dataword), each time it checks if the i'th bit is 1. It that is the case, k bits of the dividend starting from i are XORed with the polynomial, else they remain unchanged. Now, for the generator, the remainder from this operation is appended to the original message and printed to the output along with the input polynomial. In case of the verifier, this remainder is just checked for equality with 0, if that is the case, "Correct" is printed to the output, else the "Incorrect" message is displayed along with the $k - 1$ bits of the remainder.

3.3.2 Alter

The alter program, is very simple. It just read 2 lines from the input, then based on the integer offset passed as a command line arguement, it flips one of the bits in the first line. Then it outputs the possibly changed first line, along with the unmodified second line.

3.4 Results

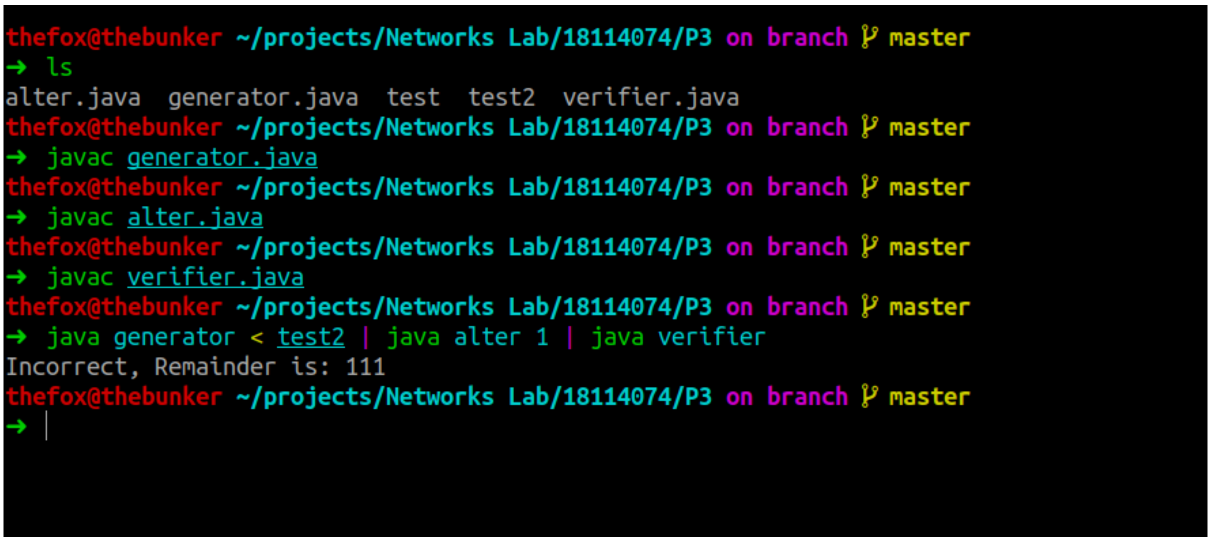
The verfier program prints "correct", when the input from the generator is passed to it without any alteration.



```
thefox@thebunker ~/projects/Networks Lab/18114074/P3 on branch ᚴ master
➜ ls
alter.java  generator.java  test  test2  verifier.java
thefox@thebunker ~/projects/Networks Lab/18114074/P3 on branch ᚴ master
➜ javac generator.java
thefox@thebunker ~/projects/Networks Lab/18114074/P3 on branch ᚴ master
➜ javac verifier.java
thefox@thebunker ~/projects/Networks Lab/18114074/P3 on branch ᚴ master
➜ java generator < test2 | java verifier
Correct
thefox@thebunker ~/projects/Networks Lab/18114074/P3 on branch ᚴ master
➜ cat test2
1001
1011
thefox@thebunker ~/projects/Networks Lab/18114074/P3 on branch ᚴ master
➜ |
```

Figure 5: Results of the program for Problem Statement 3(a)

The verfier program prints ”incorrect”, with the produced remainder if the output from the generator is altered using the alter program before being passed to the verifier program.



```
thefox@thebunker ~/projects/Networks Lab/18114074/P3 on branch ᚴ master
➜ ls
alter.java  generator.java  test  test2  verifier.java
thefox@thebunker ~/projects/Networks Lab/18114074/P3 on branch ᚴ master
➜ javac generator.java
thefox@thebunker ~/projects/Networks Lab/18114074/P3 on branch ᚴ master
➜ javac alter.java
thefox@thebunker ~/projects/Networks Lab/18114074/P3 on branch ᚴ master
➜ javac verifier.java
thefox@thebunker ~/projects/Networks Lab/18114074/P3 on branch ᚴ master
➜ java generator < test2 | java alter 1 | java verifier
Incorrect, Remainder is: 111
thefox@thebunker ~/projects/Networks Lab/18114074/P3 on branch ᚴ master
➜ |
```

Figure 6: Results of the program for Problem Statement 3(b)

4 Problem Statement 4

4.1 Question

Write a C++/Java program that accepts an IP address and subnet mask in CIDR notation, and print the following information about the sub-network:

1) Subnet Mask in dotted decimal notation (Example 255.0.0.0)
2) Network Address in dotted decimal notation: (Example 103.0.0.0)
3) Usable Host IP Range: Starting IP 103.0.0.1 — Ending IP 103.255.255.254

4.2 Theory

4.2.1 Classless Inter Domain Routing

The internet's uses the **Classless Interdomain Routing (CIDR)** method to assign addresses. The basic idea behind CIDR is to allocate IP addresses in variable size blocks. For e.g., if a site needs 4000 addresses, it is given a block of 4096 addresses. This block is aligned to a 4096-byte boundary.

IP Range Out of the range of IPs, the first IP is the **network address** and the last IP is the broadcast address. Between those values, are the usable IP addresses for devices in the network.

Subnet Mask A subnet mask is an IP address which contains the n most significant bits of the address set to 1 and the rest to 0.

4.2.2 CIDR Notation

CIDR uses a notation of the form $d_1.d_2.d_3.d_4/n$, where $d_1.d_2.d_3.d_4$ is an IP address in dotted decimal notation and n is a number.

The n most significant bits of the address correspond to the prefix (or network prefix) of the address. All IP addresses in the subnet share this same common prefix. The other *32 - n* bits of an address are used to distinguish among the devices within the subnet.

4.3 Implementation Details

4.3.1 Input Parsing

The program takes input as a command line arguement. It is first checked if an arguement is given. If not, the help is printed. Then the arguement is split at the ':'. Then it is checked whether both parts are present or not. Then the IP address is converted from the dotted decimal notation to a 32 bit integer value.

4.3.2 Subnet Mask

The subnet mask is calculated by creating a number with all bits as 1, then shifting the number by (32 - n). This value is the subnet mask in 32 bit number form. This is then converted to a dotted decimal notation form for displaying.

4.3.3 Network Address

This is calculated by **AND**ing (& operator) the IP address with the subnet mask obtained (in integer form). This is then converted to a dotted decimal notation for displaying.

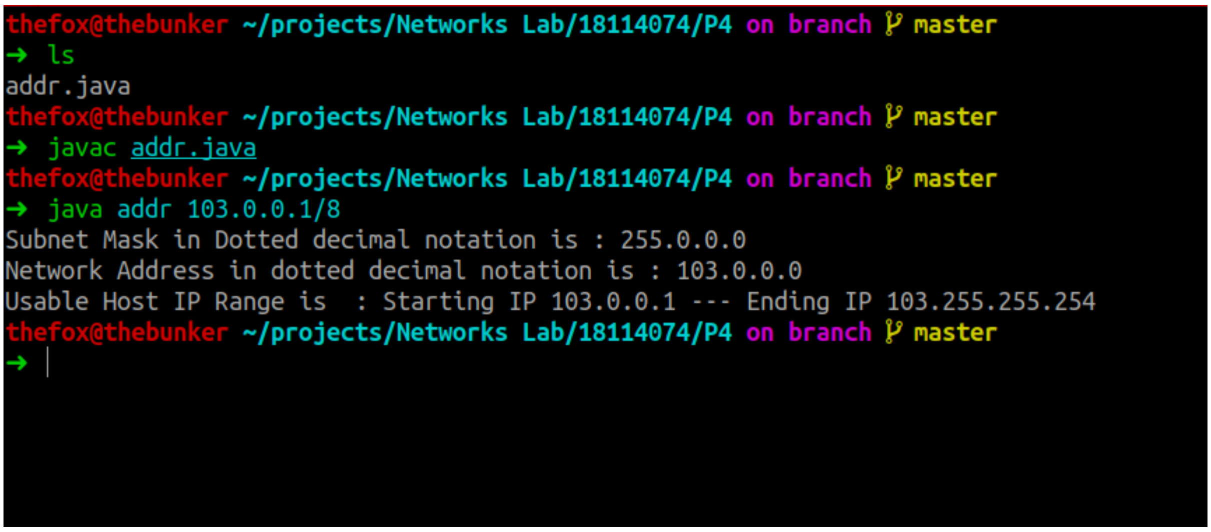
4.3.4 Usable IP Range

This consists of 3 cases:

- If mask is 32, which means that the subnet has only 1 address. Thus this is the only usable IP value.
- If mask is 31, which means that the subnet has 2 addresses, one for network address and one for broadcast address. Thus, there are no usable IP addresses.
- In other cases, the IP just after the network address is the starting address for the usable IP range and the IP address just before the broadcast address (obtained by **OR**ing (| operator) the network address with a number which has the bottom (32 - x) bits set to 1) is the last usable IP address.

4.4 Results

The program successfully prints the 3 required outputs, i.e. the subnet mask in dotted decimal notation, the network address in dotted decimal notation and the usable host IP range.



```
thefox@thebunker ~/projects/Networks Lab/18114074/P4 on branch ᚠ master
➜ ls
addr.java
thefox@thebunker ~/projects/Networks Lab/18114074/P4 on branch ᚠ master
➜ javac addr.java
thefox@thebunker ~/projects/Networks Lab/18114074/P4 on branch ᚠ master
➜ java addr 103.0.0.1/8
Subnet Mask in Dotted decimal notation is : 255.0.0.0
Network Address in dotted decimal notation is : 103.0.0.0
Usable Host IP Range is  : Starting IP 103.0.0.1 --- Ending IP 103.255.255.254
thefox@thebunker ~/projects/Networks Lab/18114074/P4 on branch ᚠ master
➜ |
```

Figure 7: Results of the program for Problem Statement 4

5 Problem Statement 5

5.1 Question

Write a program that an instructor can use to demonstrate the method of calculating IPv4 checksum. Your program should ask the user to enter the values of different fields of an IPv4 header. It should then calculate IPv4 checksum. Your program should not only show the final result but should also demonstrate the method (each step) to calculate the checksum.

5.2 Theory

5.2.1 IP datagrams

An IP datagram has 2 parts a header part and the data. The header contains 20-bytes of required fields and a variable length options field. The structure of the header is shown in figure below[8].

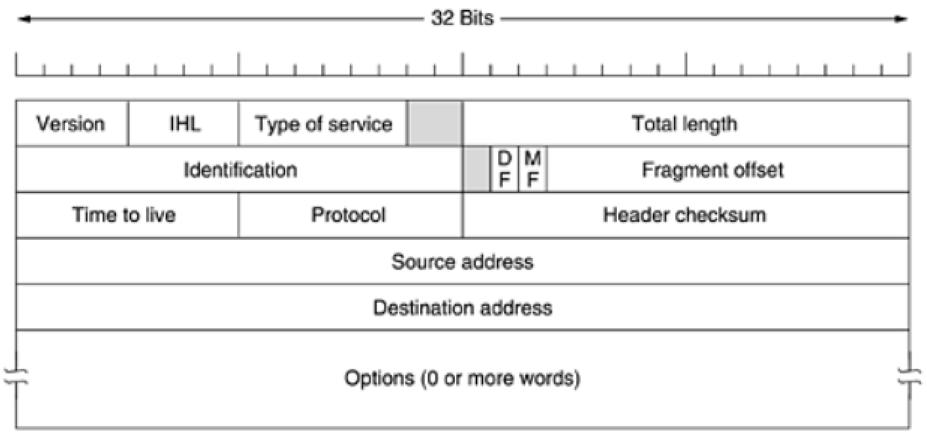


Figure 8: IPv4 Header

The various fields in the header are:

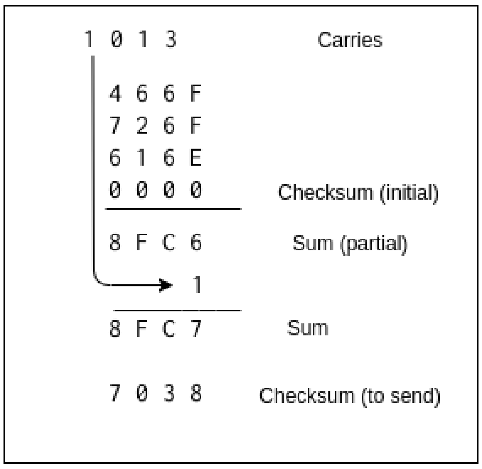
- **Version**: This is a 4 bit field. It shows which version of the IP protocol is being used by the current datagram. This is done in order to have the transition between versions.
- **IHL**: This is a 4 bit field. It tells how long the header is, in 4-byte words. The smallest possible value is 5 (when no options are present). Since this is a 4-bit field, the maximum value is 15, thus limiting the size of the header to 60 bytes.
- **Type of Service**: This is 6-bit field. It is used to indicate which service class each packet belongs to. These classes include the 4 queueing priorities, 3 discard probabilities and some other historical classes.
- **Total Length**: It is a 16 bit field. This is the total length of the datagram (i.e. header + data), in bytes. Since it is a 16 bit field, the largest possible value is 65535 bytes.
- **Identification**: It is a 16 bit field. This identifies the datagram a particular fragment belongs to. This value is same for all the packets in a datagram.

- **DF**: It is a 1 bit field. It sends for Don't Fragment. It is like an order to a router to not fragment the datagram. This might be because the receiver is incapable of reassembling the fragments back.
- **MF**: It is a 1 bit field. It stands for More Fragments. This is used to signify whether a fragment is the last fragment for a datagram or not. This bit is set to 1 for all the fragments in a datagram except for the last one.
- **Fragment offset**: It is a 13 bit field. It tells us where the current fragment starts in the datagram. Since it is a 13 bit field, there can be a maximum of 8192 fragments for a datagram.
- **Time To Live**: It is an 8 bit field. In practice, this value is decremented on each hop, thus allowing for a maximum of 255 hops. When this value becomes 0, the machine receiving this packet discards the datagram, and sends a warning packet to the sender.
- **Protocol**: It is an 8 bit field. It specifies which transport process will handle the assembled datagram. The possibilities are TCP, UDP and some others.
- **Header Checksum**: It is a 16 bit field. This is used to verify the header and detect any error which might have occured. The algorithm for the computation of this field is discussed in the next section$^{5.2.2}$.
- **Source Address**: It is a 32 bit field. It is the identifier for the sender. More about IP addresses is discussed in section 1.2.3
- **Destination Address**: It is a 32 bit field. It is the identifier for the receiver.

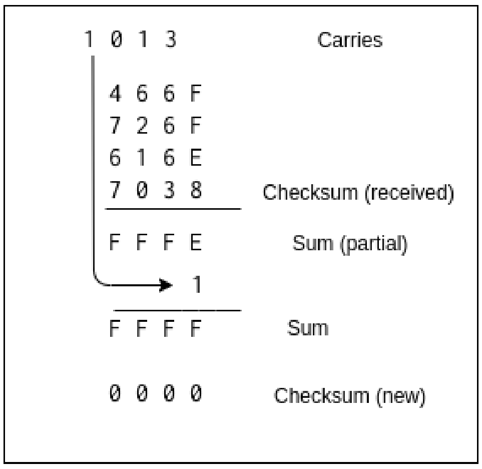
5.2.2 Checksum Algorithm

In short, the checksum algorithm takes all 16-bit halfwords and then adds then using 1's complement arithmetic. If the sum happens to be greater than 2^{16}, then the extra leftmost bits are added to the 16 rightmost bits (wrapping the sum) . Then, finally a 1's complement is done on the result.

On the sender side, the checksum part is initially set to 0, then the checksum is calculated and set to the field. On the receiver site, the result is calculated and checked against 0, if it happens to be zero, no errors were found and the datagram is accepted, else the datagram is rejected.



(a) Checksum calculation by sender



(b) Checksum calculation by receiver

5.3 Implementation Details

The program prompts the user for the value of the different fields, checks if those values are valid or not, then prompts the user again, if needed. For getting the 16 bit halfwords, the program does bit shifts and appends wherever needed, for example, the first halfword is found by:

```
vals[0] = (short)((version << 12) | (IHL << 8) | (typeOfService << 2));
```

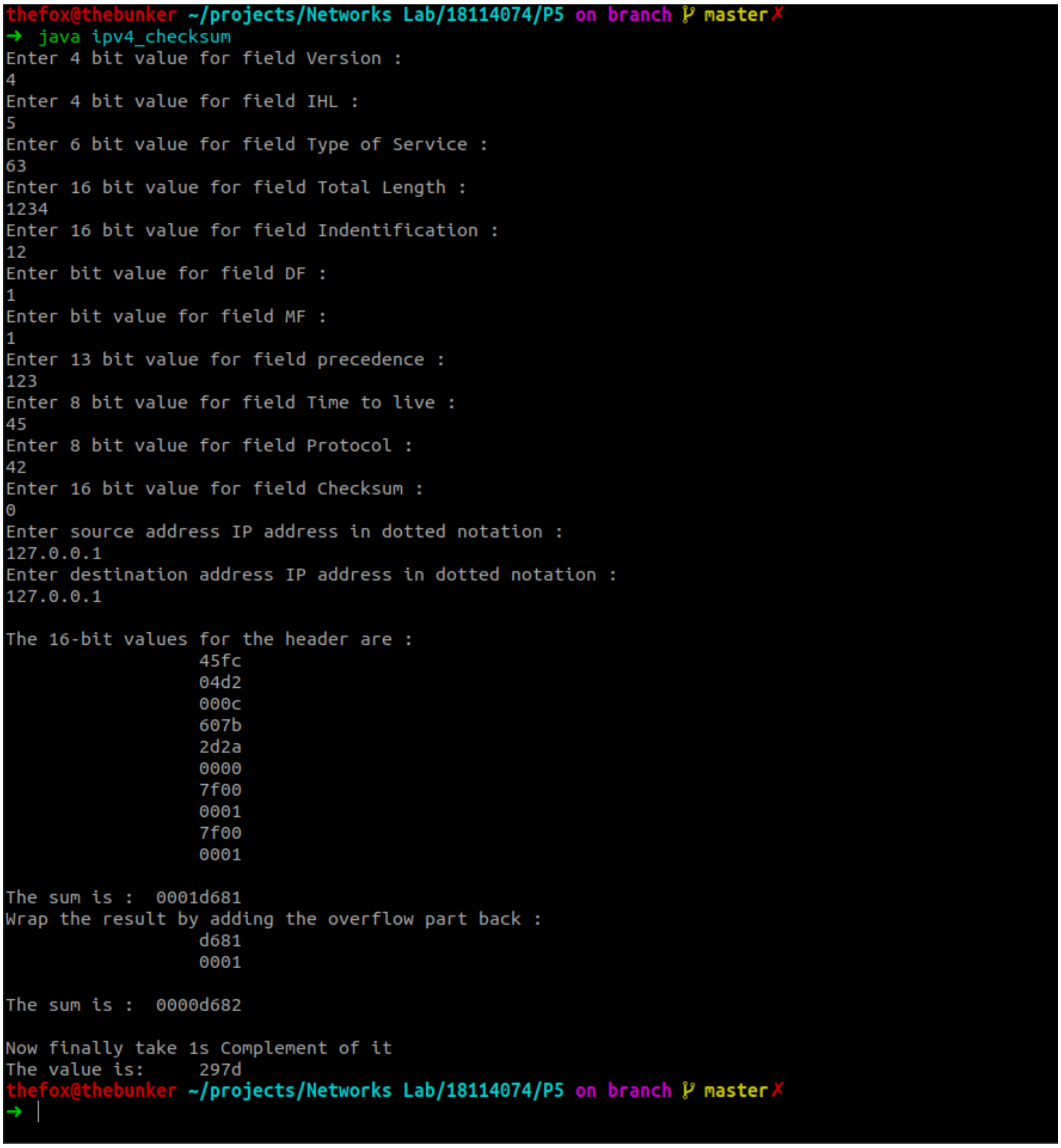
This is for shorter fields. For larger fields like IP addresses, the program uses the individual bytes along with required bit shifts:

```
vals[7] = (short)((src[2] << 8) | (src[3]));
```

The checksum algorithm just sums all these shorts and then adds the leftmost bits outside the 16 bit boundary to the rightmost 16 bits. Then finally a 1's complement of the result is performed. All these steps are shown by the program as well: The numbers to be added are displayed (in hexadecimal notation) with matching indent levels. The partial sum result is displayed such that the rightmost 16 bits are aligned with the individual halfwords. Then the final sum is displayed. Finally, the result of the 1's complement is shown. All these look like the figures[9b] in the previous section. All the instruction like "performing an addition" or "take a 1's complement" is also displayed.

5.4 Results

The program successfully calculates the checksum of the IPv4 header, while showing all the necessary computation steps.



```
thefox@thebunker ~/projects/Networks Lab/18114074/P5 on branch ᚠ master✗
➜ java ipv4_checksum
Enter 4 bit value for field Version :
4
Enter 4 bit value for field IHL :
5
Enter 6 bit value for field Type of Service :
63
Enter 16 bit value for field Total Length :
1234
Enter 16 bit value for field Indentification :
12
Enter bit value for field DF :
1
Enter bit value for field MF :
1
Enter 13 bit value for field precedence :
123
Enter 8 bit value for field Time to live :
45
Enter 8 bit value for field Protocol :
42
Enter 16 bit value for field Checksum :
0
Enter source address IP address in dotted notation :
127.0.0.1
Enter destination address IP address in dotted notation :
127.0.0.1

The 16-bit values for the header are :
                  45fc
                  04d2
                  000c
                  607b
                  2d2a
                  0000
                  7f00
                  0001
                  7f00
                  0001

The sum is :  0001d681
Wrap the result by adding the overflow part back :
                  d681
                  0001

The sum is :  0000d682

Now finally take 1s Complement of it
The value is:     297d
thefox@thebunker ~/projects/Networks Lab/18114074/P5 on branch ᚠ master✗
➜ |
```

Figure 10: Results of the program for Problem Statement 5

6 Problem Statement 6

6.1 Question

Write a C++/Java program for the Decibel (dB) calculator. You program should perform the following operations:

1) If transmit power of a network device is given in watts (Example: '20 W') then the program should print the transmit power in Decibel Watts (dBW) and Decibel milliwatts (dBm).
2) Else if transmit power of a network device is given in Decibel Watts (dBW) or Decibel Milliwatts (Example: '20 dBW' or '20 dBm') then the program should print the transmit power in Watts.

6.2 Theory

Decibel Watt (dBW) is a unit for the measurement of strength of a signal. It is used because very large and very small values can be represented in this unit using a small number of values.

To convert power in Watts to decibel watts:

$$Power\ in\ dBW \ = \ 10\ \log_{10}\frac{Power}{1W}$$

and to convert back:

$$Power\ in\ W \ = \ 10^{\frac{Power\ in\ dBW}{10}}$$

Similarly decibel milliwatts (dBm) is in reference to a milliwatt and 1 $watt \ = \ 1000\ milliwatts$. Thus,

$$\begin{aligned} Power\ in\ dBm \ &= \ 10\log_{10}\frac{Power}{1mW} \\ &= \ 10\log_{10}\frac{1000 * Power}{1W} \\ &= \ 30 \ + \ 10\log_{10}\frac{Power}{1W} \\ &= \ 30 \ + \ Power\ in\ dBm \end{aligned}$$

6.3 Implementation Details

6.3.1 Input Parsing

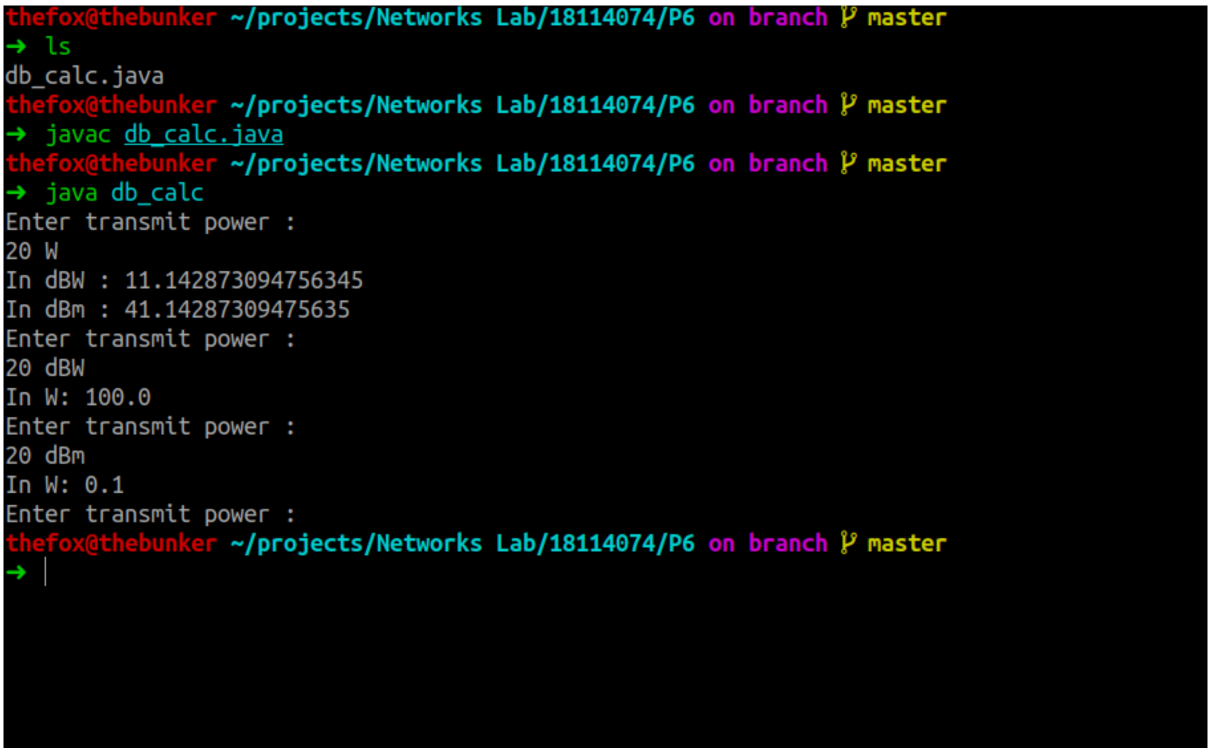
In order to parse the input, it is split upon a space. Then the first part is used as a '**double**' type value. After that a switch case statement is executed on the basis of the second part of the obtained split. This has cases for **W**, **dBW** and **dBm**, along with a default case for error handling.

6.3.2 Conversion

The implementation leverages the conversion between dBm to dBW, and thus just contains functions to convert between W to dBW and vice versa, which just implement the formulas mentioned above. To convert to dBm from Watts, 30 is added to the dBW value. To convert from dBm to W, we just reuse the conversion from dBW to Watts, whose result would be milliWatts in such a case, then divide it by 1000 to convert it to Watts.

6.4 Results

The program is successfully able to convert the transmit power from Watts to decibel watts and decibel milliwatts and vice versa.



```
thefox@thebunker ~/projects/Networks Lab/18114074/P6 on branch ᚠ master
➜ ls
db_calc.java
thefox@thebunker ~/projects/Networks Lab/18114074/P6 on branch ᚠ master
➜ javac db_calc.java
thefox@thebunker ~/projects/Networks Lab/18114074/P6 on branch ᚠ master
➜ java db_calc
Enter transmit power :
20 W
In dBW : 11.142873094756345
In dBm : 41.14287309475635
Enter transmit power :
20 dBW
In W: 100.0
Enter transmit power :
20 dBm
In W: 0.1
Enter transmit power :
thefox@thebunker ~/projects/Networks Lab/18114074/P6 on branch ᚠ master
➜ |
```

Figure 11: Results of the program for Problem Statement 6