**Name: Tanmoy Sarkar**
**Roll No: 002010501020**
**Class: BCSE III**
**Assignment No: 8**
**Subject: Computer Network**
**Group: A1**

---

# DNS Protocol

**Overview :** On the internet for communication between client and server , client requires server/host's IP address, but in this crowded network it's not possible to remember Ip of every host to connect to it. So there comes the domain name [example.com] . It's much easier to remember domain names rather than IP addresses. But the client in the lower lever requires an IP address to communicate. Here the DNS come. Clients can request DNS to send the domain's IP address, so that client can use the IP to connect.

**How it works :**
DNS protocol works over UDP protocol.

- Client send and dns request consists of domain name, type of query [A, NS, CNAME, MX, TXT etc.], query class type [usually it is IN (internet)].
- The DNS server will check for the domain in the zone records .
- If DNS has zone records of that domain, will send the Answers of the query.
- If there is no zone record for that domain, it will send the dns request to top level TLD root servers and from there the request will be send to the latest nameservers of that domain . The nameserver will forward the result to that client.

**Implementation :**
DNS protocol and its packet encoding and decoding have been implemented as per specified in RFC 1035.

*DNS request has this structure*

```
+--------------------+
|       Header       |
+--------------------+
|      Question      | the question for the name server
+--------------------+
|       Answer       | RRs answering the question
+--------------------+
|      Authority     | RRs pointing toward an authority
+--------------------+
|     Additional     | RRs holding additional information
+--------------------+
```

For a basic DNS server, the implementation of the Header, Question and Answer section is mandatory.

*Header section consists of*

- 16 byte of transaction ID (to track the requests between client , dns server and multiple root servers)
- A one bit field that specifies whether this message is a query (0), or a response (1).
- OPCODE : A four bit field that specifies a kind of query in this message.
    - 0 - Standard Query
    - 1 - an inverse query
    - 2 - a server status request

- AA : Authoritative Answer - this bit is valid in responses, and specifies that the responding name server is an  authority for the domain name in question section.
- TC : TrunCation - specifies that this message was truncated  due to length greater than that permitted on the transmission channel.
- QDCOUNT :  an unsigned 16 bit integer specifying the number of entries in the question section.
- ANCOUNT :  an unsigned 16 bit integer specifying the number of  resource records in the answer section.
- NSCOUNT :  an unsigned 16 bit integer specifying the number of name  server resource records in the authority records section.
- ARCOUNT :  an unsigned 16 bit integer specifying the number of  resource records in the additional records section.
- Other parameters also there Recursion Available [RA], Response Code [RCODE]

*Question section format*

```
   0  1  2  3  4  5  6  7  8  9  0  1  2  3  4  5
 +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
 |                                               |
 /                     QNAME                     /
 /                                               /
 +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
 |                     QTYPE                     |
 +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
 |                     QCLASS                    |
 +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

It's consists of :

- QNAME   : Domain name of query [ex. Google.com]. It's variable length
- QTYPE    : An numeric value which represents Record Type. [A, CNAME, MX, TXT, AAAA]
- QCLASS  : A two octet code that specifies the class of the query.  For example, the QCLASS field is IN for the Internet.

*Answer Section Format :*

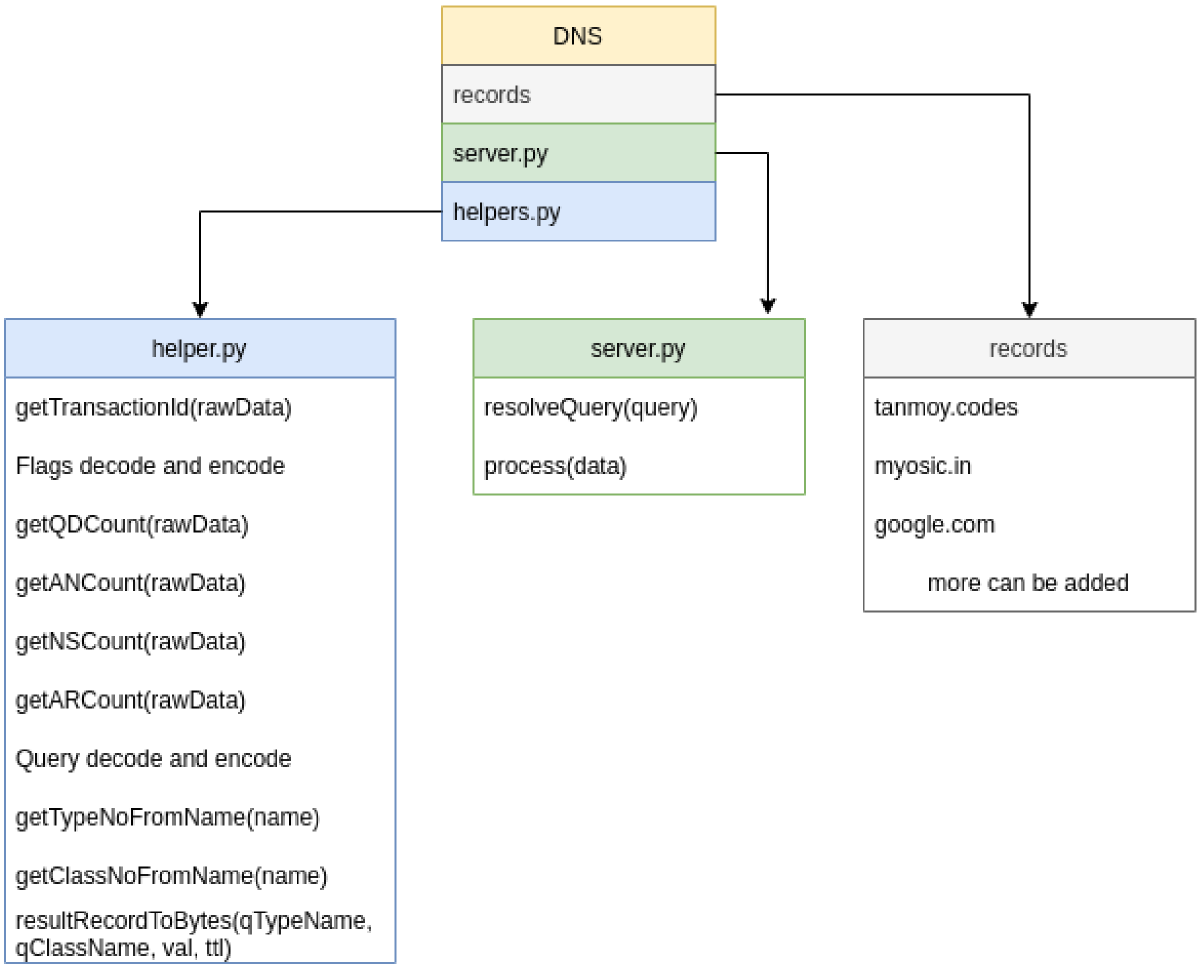
```
   0  1  2  3  4  5  6  7  8  9  0  1  2  3  4  5
 +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
 |                                               |
 /                                               /
 /                      NAME                     /
 |                                               |
 +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
 |                      TYPE                     |
 +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
 |                     CLASS                     |
 +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
 |                      TTL                      |
 +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
 |                   RDLENGTH                    |
 +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--|
 /                     RDATA                     /
 /                                               /
 +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

It consists of :

- NAME : a domain name to which this resource record pertains.
- TYPE : Type of record [A, NS, CNAME, MX, TXT etc.]

- CLASS : A two octet code that specifies the class of the query. For example, the QCLASS field is IN for the Internet
- TTL : a 32 bit unsigned integer that specifies the time interval (in seconds) that the resource record may be cached before it should be discarded
- RDLENGTH : an unsigned 16 bit integer that specifies the length in octets of the RDATA field.
- RDATA : a variable length string of octets that describes the resource.

**Folder Structure :**



**Code Implementation -**

*server.py*

```
import socket
from helpers import *
import  os

UDP_IP_ADDRESS = "127.0.0.1"
UDP_PORT_NO = 53


serverSock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
serverSock.bind((UDP_IP_ADDRESS, UDP_PORT_NO))
```

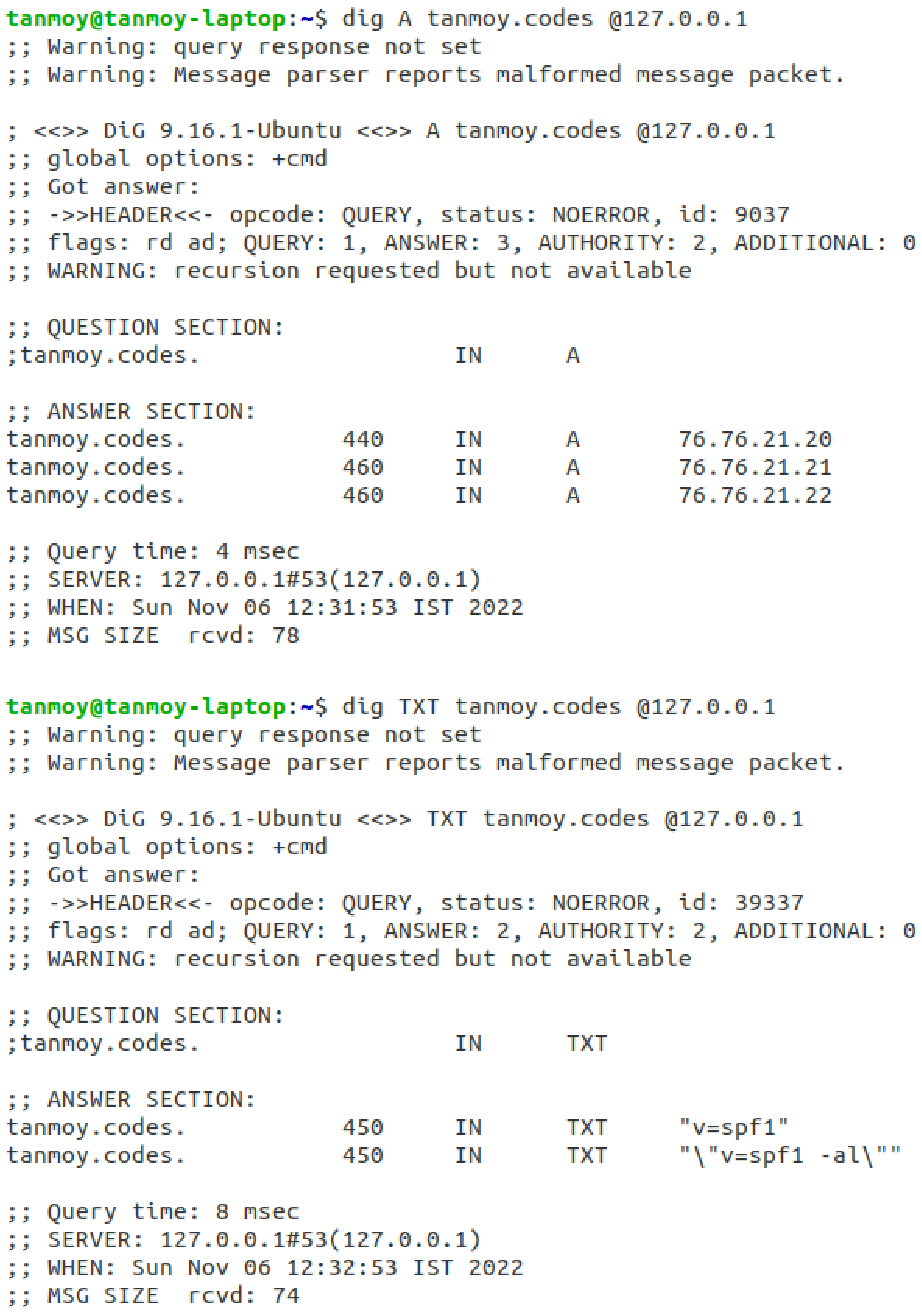
```
def resolveQuery(query:Query):
   record_path = os.path.join(os.path.dirname(__file__), "records",
query.baseDomainName())
   # If the file exists, then we have a record for this domain
   if os.path.exists(record_path):
       answers = []
       nameservers = []
       # Read the file and parse the answers
       with open(record_path, "r") as f:
           for curLine in f:
               curLine = curLine.strip() # Strip to remove newlines
               curLineParts = curLine.split("\t")
               if len(curLineParts) < 3: continue
               if curLineParts[0] == query.domainName(endDot=True) and curLineParts[1]
== query.queryClass():
                   if curLineParts[2] == query.queryType():
                       answers.append(curLine)
                   if curLineParts[2] == "NS":
                       nameservers.append(curLine)
       return (True, answers, nameservers)

   # else , we have no record for this domain
   return (False, [], [])


def process(data):
   transactionID = getTransactionID(data)
   flags = Flags.fromBytes(data, debug=False)
   QDCount = getQDCount(data) # Usually 1
   ANCount = getANCount(data)
   NSCount = getNSCount(data)
   # ARCount = getARCount(data)
   ARCount = 0
   query = Query.fromBytes(data)

   print("Transaction ID: " + transactionID)

   #  Find the record for this query
   result = resolveQuery(query)

   if result[0]:
       transactionID = data[:2]
       QDCount = (QDCount).to_bytes(2, "big")
       ANCount = len(result[1]).to_bytes(2, "big")
       NSCount = len(result[2]).to_bytes(2, "big")
       ARCount = (0).to_bytes(2, "big")
   else:
       transactionID = data[:2]
       QDCount = (QDCount).to_bytes(2, "big")
       ANCount = (0).to_bytes(2, "big")
```

```
        NSCount = (0).to_bytes(2, "big")
        ARCount = (0).to_bytes(2, "big")

    # Build the response
    # Header
    responseHeader = transactionID + flags.toBytes() + QDCount + ANCount + NSCount +
ARCount
    print("Response Header: " + str(responseHeader))

    # Question
    responseQuestion = query.toBytes()
    print("Response Question: " + str(responseQuestion))

    # Body
    responseBody = b""
    for record in result[1]:
        record_split = record.split("\t")
        # Name | Class | Type | Val | TTL
        responseBody += resultRecordToBytes(record_split[2], record_split[1],
record_split[3], int(record_split[4]))

    return responseHeader + responseQuestion + responseBody

while True:
    data, addr = serverSock.recvfrom(512)
    serverSock.sendto(process(data), addr)
```

*helpers.py*

```
### Required datas
queryTypesName = ["A", "NS", "MD", "MF", "CNAME", "SOA", "MB", "MG", "MR", "NULL", "WKS",
"PTR", "HINFO", "MINFO", "MX", "TXT"]
queryClassesName = ["IN", "CS", "CH", "HS"]

### Functions
# Get Transaction ID from data
def getTransactionID(rawData:bytes):
    transactionIDRaw = rawData[:2]
    transactionID = ""
    for x in transactionIDRaw:
        transactionID += hex(x)[2:]
    return transactionID

# Get flags from data
class Flags:
    def __init__(self, QR, OPCODE, AA, TC, RD, RA, Z, RCODE):
        self.QR = QR
        self.OPCODE = OPCODE
        self.AA = AA
        self.TC = TC
        self.RD = RD
        self.RA = RA
        self.Z = Z
        self.RCODE = RCODE

    def __str__(self):
        return str(self.QR) + " " + str(self.OPCODE) + " " + str(self.AA) + " " +
str(self.TC) + " " + str(self.RD) + " " + str(self.RA) + " " + str(self.Z) + " " +
str(self.RCODE)

    @staticmethod
    def fromBytes(rawData:bytes, debug=False):
        #       first byte       |second byte
        # | 1 |   4    | 1 | 1 | 1 | 1 | 3|   4  |
        # | QR|Opcode| AA| TC| RD| RA| Z|RCODE|

        first_byte = ord(rawData[2:3])
        second_byte = ord(rawData[3:4])

        if debug:
            print("[DEBUG] First byte: " + str(first_byte)+" "+bin(first_byte))
            print("[DEBUG] Second byte: " + str(second_byte)+" "+bin(second_byte))

        QR = first_byte >> 7
        OPCODE = (first_byte  & 0b01111000 ) >> 3
        AA = (first_byte & 0b00000100) >> 2
        TC = (first_byte & 0b00000010) >> 1
        RD = (first_byte & 0b00000001)

        RA = second_byte >> 7
        Z = (second_byte & 0b1110000) >> 4
        RCODE = second_byte & 0b1111

        if debug:
            print("[DEBUG] QR: " + str(QR))
            print("[DEBUG] OPCODE: " + str(OPCODE))
            print("[DEBUG] AA: " + str(AA))
            print("[DEBUG] TC: " + str(TC))
            print("[DEBUG] RD: " + str(RD))
            print("[DEBUG] RA: " + str(RA))
            print("[DEBUG] Z: " + str(Z))
            print("[DEBUG] RCODE: " + str(RCODE))

        return Flags(QR, OPCODE, AA, TC, RD, RA, Z, RCODE)

    def toBytes(self):
        first_byte = (self.QR << 7) + (self.OPCODE << 3) + (self.AA << 2) + (self.TC << 1)
+ self.RD
        second_byte = (self.RA << 7) + (self.Z << 4) + self.RCODE
        return first_byte.to_bytes(1, "big") + second_byte.to_bytes(1, "big")

# Get QD count from data
def getQDCount(rawData:bytes):
    QDCountRaw = rawData[4:6]
    QDCount = ""
    for x in QDCountRaw:
        QDCount += hex(x)[2:]
    return int(QDCount)

# Get AN count from data
def getANCount(rawData:bytes):
    ANCountRaw = rawData[6:8]
    ANCount = ""
    for x in ANCountRaw:
        ANCount += hex(x)[2:]
    return int(ANCount)

# Get NS count from data
def getNSCount(rawData:bytes):
    NSCountRaw = rawData[8:10]
    NSCount = ""
    for x in NSCountRaw:
        NSCount += hex(x)[2:]
    return int(NSCount)

# Get AR count from data
def getARCount(rawData:bytes):
    ARCountRaw = rawData[10:12]
    ARCount = ""
    for x in ARCountRaw:
        ARCount += hex(x)[2:]
    return int(ARCount)

# Query Questions parser
class Query:
    def __init__(self, nameParts, type, qclass):
        self.nameParts = nameParts
        self.type = type
        self.qclass = qclass

    def baseDomainName(self):
        return self.nameParts[-2] + "." + self.nameParts[-1]

    def domainName(self, endDot=False):
        res = '.'.join(self.nameParts)
        if endDot and res[-1] != ".":
            res += "."
        return res

    def queryType(self):
        if self.type > 16:
            return "Unknown"
        return queryTypesName[self.type - 1]

    def queryClass(self):
        if self.qclass > 4:
            return "Unknown"
        return queryClassesName[self.qclass - 1]

    def __str__(self):
        return self.domainName() + " " + str(self.type) + " " + str(self.qclass)

    @staticmethod
    def fromBytes(rawData:bytes, debug=False):
        data = rawData[12:]
        if debug:
            print("[DEBUG] Data: ")
            print(data)
        # Format : {length of characters + [string]} untill 0x00

        domainParts = []

        totalLengthOfDomainPartsWithLengthCharacter = 0
        length = -1
        tmp = ""
        for byte in data:
            totalLengthOfDomainPartsWithLengthCharacter += 1
            if byte == 0: # End of parts
                if tmp != "":
                    domainParts.append(tmp)
                break

            if length == -1: # Yet not started, so first byte is length
                length = byte
            elif length == 0: # Read one part, so next byte is length
                domainParts.append(tmp)
                tmp = ""
                length = byte
            else: # Its part of domain name
                tmp += chr(byte)
                length -= 1

        tmp =
data[totalLengthOfDomainPartsWithLengthCharacter:totalLengthOfDomainPartsWithLengthCharact
er+4]
        # QType
        QTypeRaw = tmp[:2]
        QType = ""
        for x in QTypeRaw:
            QType += hex(x)[2:]
        QType = int(QType, 16)
        # QClass
        QClassRaw = tmp[2:]
        QClass = ""
        for x in QClassRaw:
            QClass += hex(x)[2:]
        QClass = int(QClass, 16)
        if debug:
            print("Domain Parts: " + str(domainParts))
            print("QType: " + str(QType))
            print("QClass: " + str(QClass))

        return Query(domainParts, QType, QClass)

    def toBytes(self):
        res = b""
        for part in self.nameParts:
            res += bytes([len(part)]) + bytes(part, "utf-8")
        res += b"\x00"
        res += self.type.to_bytes(2, "big")
        res += self.qclass.to_bytes(2, "big")
        return res

# Get type no from Query type name
def getTypeNoFromName(name):
    for i in range(len(queryTypesName)):
        if name == queryTypesName[i]:
            return i + 1
    return 0

# Get class no from Query class name
def getClassNoFromName(name):
    for i in range(len(queryClassesName)):
        if name == queryClassesName[i]:
            return i + 1
    return 0

# Result Record to bytes
def resultRecordToBytes(qTypeName, qClassName, val:str, ttl:int):
    print("Converting " + qTypeName + " " + qClassName + " " + val + " " + str(ttl))
    res = b"\xc0\x0c"
    res += getTypeNoFromName(qTypeName).to_bytes(2, "big")
    res += getClassNoFromName(qClassName).to_bytes(2, "big")
    res += ttl.to_bytes(4, "big")
    if qTypeName == "A":
        res += b"\x00\x04"
        res += bytes(map(int, val.split(".")))
    elif qTypeName == "AAAA":
        val = [int(x) for x in val.split(":")]
        res += len(val).to_bytes(2, "big")
        for x in val:
            res += x.to_bytes(2, "big")
    else:
        totalLen = 0
        data = b""
        for x in val.split("\n"):
            r = bytes(x, "utf-8")
            rLen = len(r)
            totalLen += rLen+1
            data += rLen.to_bytes(1, "big") + r

        res += totalLen.to_bytes(2, "big")
        res += data

    return res
```

**Working Demo [via dig] :**



```
tanmoy@tanmoy-laptop:~$ dig A tanmoy.codes @127.0.0.1
;; Warning: query response not set
;; Warning: Message parser reports malformed message packet.

; <<>> DiG 9.16.1-Ubuntu <<>> A tanmoy.codes @127.0.0.1
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 9037
;; flags: rd ad; QUERY: 1, ANSWER: 3, AUTHORITY: 2, ADDITIONAL: 0
;; WARNING: recursion requested but not available

;; QUESTION SECTION:
;tanmoy.codes.                  IN      A

;; ANSWER SECTION:
tanmoy.codes.           440     IN      A       76.76.21.20
tanmoy.codes.           460     IN      A       76.76.21.21
tanmoy.codes.           460     IN      A       76.76.21.22

;; Query time: 4 msec
;; SERVER: 127.0.0.1#53(127.0.0.1)
;; WHEN: Sun Nov 06 12:31:53 IST 2022
;; MSG SIZE  rcvd: 78

tanmoy@tanmoy-laptop:~$ dig TXT tanmoy.codes @127.0.0.1
;; Warning: query response not set
;; Warning: Message parser reports malformed message packet.

; <<>> DiG 9.16.1-Ubuntu <<>> TXT tanmoy.codes @127.0.0.1
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 39337
;; flags: rd ad; QUERY: 1, ANSWER: 2, AUTHORITY: 2, ADDITIONAL: 0
;; WARNING: recursion requested but not available

;; QUESTION SECTION:
;tanmoy.codes.                  IN      TXT

;; ANSWER SECTION:
tanmoy.codes.           450     IN      TXT     "v=spf1"
tanmoy.codes.           450     IN      TXT     "\"v=spf1 -al\""

;; Query time: 8 msec
;; SERVER: 127.0.0.1#53(127.0.0.1)
;; WHEN: Sun Nov 06 12:32:53 IST 2022
;; MSG SIZE  rcvd: 74
```

**Conclusion :** The implementation of the DNS server is working fine with the DNS client. However it can only send results based on the zone records available to it, So we can extend that by giving support for querying the root server.

---

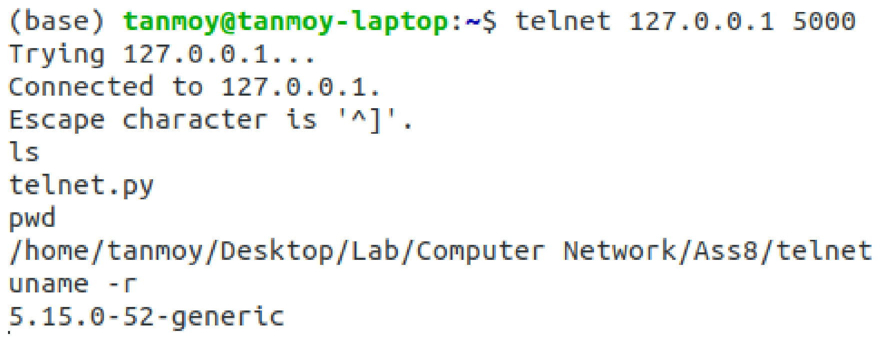
## Telnet Protocol

**Overview -**Telnet is a network protocol used to virtually access a computer and to provide a two-way, collaborative and text-based communication channel between two machines. It follows a user command Transmission Control Protocol/Internet Protocol (TCP/IP) networking protocol for creating remote sessions

**Code Implementation -**

```
import socket
import subprocess
from threading import Thread
from time import sleep

class TelentServer:
    def __init__(self, host="127.0.0.1", port=5000):
        self.host = host
        self.port = port
        self.exited = False
        self.exitedListening = False
        self.sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        self.sock.bind((self.host, self.port))
        self.sock.settimeout(2)
        self.sock.listen()
        self.threads_pool = []
        print("Server is listening on port", port)

    def startProcess(self):
        while True:
            try:
                conn, addr = self.sock.accept()
                conn.settimeout(2)
                print("Connected by", addr)
                t = Thread(target=self.handleClient, args=(conn, addr))
                t.start()
                self.threads_pool.append(t)
            except socket.timeout:
                if self.exited:
                    break
                else :
                    continue
            except:
                break
        self.exitedListening = True
    def handleClient(self, conn:socket.socket, addr):
        while True:
            try:
                data = conn.recv(1024)
                res = subprocess.run(data.decode().strip(), shell=True,
capture_output=True)
                if res.returncode == 0:
                    conn.sendall(res.stdout)
                else:
                    conn.sendall(res.stderr)
                if not data:
                    break
                if data.decode().strip() == "exit":
                    break
            except socket.timeout:
                if self.exited:
                    break
                continue
            except:
                break
        conn.close()

    def stopProcess(self):
        self.exited = True
        while not self.exitedListening:
            sleep(1)
        for t in self.threads_pool:
            try:
                t.join()
            except:
                pass
        self.sock.close()

if __name__ == "__main__":
    server = TelentServer()

    try:
        server.startProcess()
    except KeyboardInterrupt:
        server.exited = True
        server.stopProcess()
        print("Server is closed")
```

**Working Demo [via Telnet Client] :**



```
(base) tanmoy@tanmoy-laptop:~$ telnet 127.0.0.1 5000
Trying 127.0.0.1...
Connected to 127.0.0.1.
Escape character is '^]'.
ls
telnet.py
pwd
/home/tanmoy/Desktop/Lab/Computer Network/Ass8/telnet
uname -r
5.15.0-52-generic
```

**Conclusion :** Telnet is a very important protocol to operate remote systems. We have built the Telnet Server on top of the TCP/IP stack and it’s compatible with any telnet client