

PING COMMAND–ICMP ECHO REQUEST/REPLY

The Internet control message protocol (ICMP) is used by the ping application or traceroute. When a ping is initiated, the sender makes an ICMP request, and the receiver responds with an ICMP echo request. Ping is used to see if there is an active connection between the sender and the recipient. If no active connection exists, the receiver sends an ICMP echo reply to the sender indicating that the host is unavailable, destination host unknown, destination port unknown, and so on.

ICMP message has three parameters:

1. Type
2. Code
3. Checksum

Type:

The message type is 8 bits. There are various message types but usually, type 0 is used for echo reply and type 8 for echo.

Code:

The code field is also of 8 bits which stores the extra information about the error.

Checksum:

The last 16 digits are assigned to the checksum. Checksum performs the consistency check to make sure that the complete message is delivered.

We'll use a raw socket to create a connection between two nodes and perform a ping in this assignment. The request.txt file contains the request message received by the receiver, while the received.txt file contains the reply messages delivered to the sender.

Raw Socket establishes a connection between client and server. When the ethernet layer receives the packet, it sends that raw packet directly to the socket and bypasses the process of TCP/IP, and sends it to the user application.

Step 1: We will take the hostname as input from the user. Hostname either can be a IP address or website URL:

Example:

Google IP address: 8.8.8.8

Google website URL: www.google.com

Step 2:

1. If the hostname is a website URL (www.google.com) then we will convert the human-understandable website to an IP address (8.8.8.8) with the help of the **gethostbyname()** function and returns the **hostent** which contains an IP address separated by dots with binary notation. This process is known as DNS lookup.

Code:

```
// Performs a DNS lookup
char *dns_lookup(char *addr_host, struct sockaddr_in *addr_con)
{
    fptr1=fopen("request.txt","a"); //Create file for storing request messages
    printf("\nResolving DNS..\n");
    struct hostent *host_entity;
    char *ip=(char*)malloc(NI_MAXHOST*sizeof(char));
    int i;

    if ((host_entity = gethostbyname(addr_host)) == NULL)
    {
        // No ip found for hostname
        return NULL;
    }

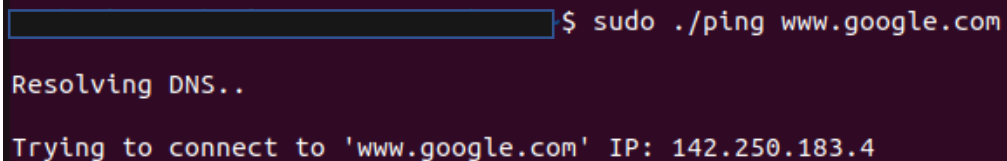
    //filling up address structure
    strcpy(ip, inet_ntoa(*(struct in_addr *)
        host_entity->h_addr));

    (*addr_con).sin_family = host_entity->h_addrtype;
    (*addr_con).sin_port = htons (PORT_NO);

    (*addr_con).sin_addr.s_addr = *(long*)host_entity->h_addr;
    sprintf(s1,"Source Port: %d\n" "Source Address: %d\n", (*addr_con).sin_port, (*addr_con).sin_addr.s_addr);
    fprintf(fptr1,"%s",s1);
    fclose(fptr1);

    return ip;
}
```

Output:



```
$ sudo ./ping www.google.com

Resolving DNS..

Trying to connect to 'www.google.com' IP: 142.250.183.4
```

2. If the hostname is an IP address then we will convert the IP address to a human-understandable website with the help of the **getnameinfo()** function. This process is known as reverse DNS lookup.

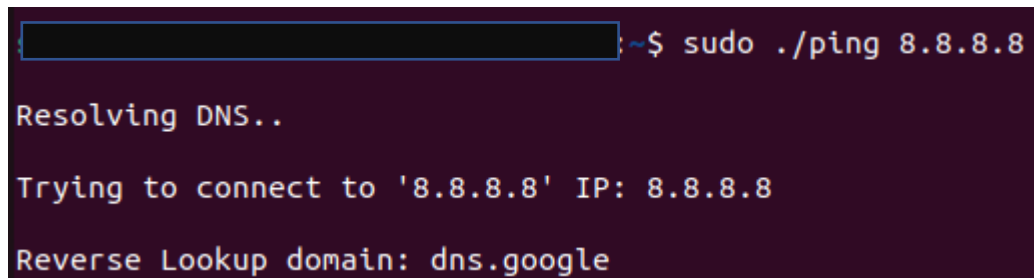
Code:

```
// Resolves the reverse lookup of the hostname
char* reverse_dns_lookup(char *ip_addr)
{
    struct sockaddr_in temp_addr;
    socklen_t len;
    char buf[NI_MAXHOST], *ret_buf;

    temp_addr.sin_family = AF_INET;
    temp_addr.sin_addr.s_addr = inet_addr(ip_addr);
    len = sizeof(struct sockaddr_in);

    if (getnameinfo((struct sockaddr *) &temp_addr, len, buf,
                    sizeof(buf), NULL, 0, NI_NAMEREQD))
    {
        printf("Could not resolve reverse lookup of hostname\n");
        return NULL;
    }
    ret_buf = (char*)malloc((strlen(buf) + 1)*sizeof(char) );
    strcpy(ret_buf, buf);
    return ret_buf;
}
```

Output:



```
~$ sudo ./ping 8.8.8.8

Resolving DNS..

Trying to connect to '8.8.8.8' IP: 8.8.8.8

Reverse Lookup domain: dns.google
```

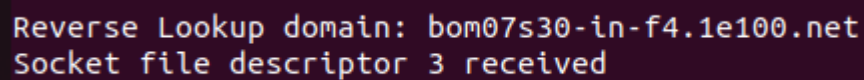
Step 3: Open a raw socket passing AF_INET, SOCK_RAW, and IPPROTO_ICMP as parameters.

AF_INET refers to the ipv4 address family. SOCK_RAW is a type of socket which helps in sending and receiving data. IPPROTO_ICMP refers to the ICMP protocol. The socket requires admin rights to run this code.

Code:

```
//socket()
sockfd = socket(AF_INET, SOCK_RAW, IPPROTO_ICMP);
if(sockfd<0)
{
    printf("\nSocket file descriptor not received!!\n");
    return 0;
}
else
    printf("\nSocket file descriptor %d received\n", sockfd);
```

Output:



```
Reverse Lookup domain: bom07s30-in-f4.1e100.net
Socket file descriptor 3 received
```

Step 4: Now let's focus on how to send and receive packets when we initiate ping.

1. Using the setsockopt() method or function set the TTL value where TTL is time to live which allows the packet to cross only those numbers of hops. In our code, we set the TTL value to 64 and passed it to the setsockopt function.

Code:

```
// set socket options at ip to TTL and value to 64,
// change to what you want by setting ttl_val
if (setsockopt(ping_sockfd, SOL_IP, IP_TTL,
               &ttl_val, sizeof(ttl_val)) != 0)
{
    printf("\nSetting socket options to TTL failed!\n");
    return;
}
else
{
    printf("\nSocket set to TTL..\n");
}
```

2. Using the same setsockopt() set the timeout for receiving function. It specifies the receiving timeout until the error is reported. If this is not set then the code will run in a loop and never halt.

Code:

```
,  
  
// setting timeout of recv setting  
setsockopt(ping_sockfd, SOL_SOCKET, SO_RCVTIMEO,  
           (const char*)&tv_out, sizeof tv_out);
```

3. Here we are creating a while loop so that the packets will be sent only twice and we receive replies twice. Every time when the pointer reaches the end of the while loop we are incrementing the count and check the condition. If the count is less than the pingloop then come out of the loop.
4. Fill up the ICMP packet with ICMP ECHO, random message, id as a PID process, and checksum field with calculated checksum.

Code:

```
pckt.hdr.type = ICMP_ECHO;  
//printf("ICMP_ECHO: %d\n",pckt.hdr.type);  
pckt.hdr.un.echo.id = getpid();  
  
for ( i = 0; i < sizeof(pckt.msg)-1; i++ )  
    pckt.msg[i] = i+'0';  
  
pckt.msg[i] = 0;  
pckt.hdr.un.echo.sequence = msg_count++;  
pckt.hdr.checksum = checksum(&pckt, sizeof(pckt));
```

5. Now send a packet to the receiver.

Code:

```
... - - -  
  
//send packet  
clock_gettime(CLOCK_MONOTONIC, &time_start);  
if ( sendto(ping_sockfd, &pckt, sizeof(pckt), 0,  
(struct sockaddr*) ping_addr,  
         sizeof(*ping_addr)) <= 0 )  
{  
    printf("\nPacket Sending Failed!\n");  
    flag=0;  
}
```

6. Once the request is sent we are creating a file and store the messages in the request.txt file with the mode "append".
7. As a packet is sent now it's time to receive the packet. The messages received by the sender are stored in the received.txt file. Here, we are calculating everything manually without using any libraries and printing in the output. For example, we have calculated round trip time and passed it as an argument to the print function.

Code:

```
,

//receive packet
addr_len=sizeof(r_addr);

if ( recvfrom(ping_sockfd, &pckt, sizeof(pckt), 0,
              (struct sockaddr*)&r_addr, &addr_len) <= 0
    && msg_count>1)
{
    printf("\nPacket receive failed!\n");
}

else
{
    clock_gettime(CLOCK_MONOTONIC, &time_end);

    double timeElapsed = (((double)(time_end.tv_nsec -
                                     time_start.tv_nsec))/1000000.0);
    rtt_nsec = (time_end.tv_sec -
                time_start.tv_sec) * 1000.0
                + timeElapsed;

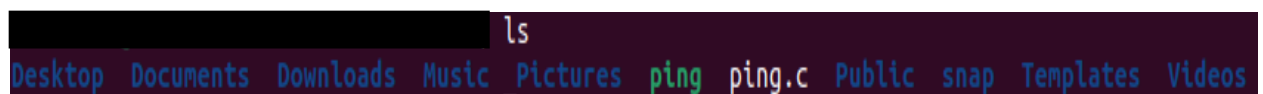
    // if packet was not sent, don't receive
    if(flag)
    {
        if(!(pckt.hdr.type ==09 && pckt.hdr.code==0))
        {
            printf("Error..Packet received with ICMP type %d code %d\n", pckt.hdr.type, pckt.hdr.code);
        }
        else
        {
            printf("%d bytes from %s (h: %s) (%s) msg_seq=%d ttl=%d rtt = %Lf ms.\n", PING_PKT_S, ping_dom, rev_host, ping_ip, msg_count, ttl_val, rtt_nsec);
            sprintf(s,"%d bytes from %s (h: %s) (%s) msg_seq=%d ttl=%d rtt = %Lf ms.\n", PING_PKT_S, ping_dom, rev_host, ping_ip, msg_count, ttl_val, rtt_nsec);

            //printf("%s",s);
            fprintf(fpPtr,"%s",s);
            fclose(fpPtr);

            msg_received_count++;
        }
    }
}
```

Now let's go through the outputs:

1. Before issuing the ping command there is no receive or request text files in the directory.



2. After issuing the ping command the request is sent and got a reply from the server. As mentioned the request is sent only twice.

```
sudo ./ping 8.8.8.8

Resolving DNS..

Trying to connect to '8.8.8.8' IP: 8.8.8.8

Reverse Lookup domain: dns.google
Socket file descriptor 3 received

Socket set to TTL..
256 bytes from dns.google (h: 8.8.8.8) (8.8.8.8) msg_seq=1 ttl=64 rtt = 27.461418 ms.
256 bytes from dns.google (h: 8.8.8.8) (8.8.8.8) msg_seq=2 ttl=64 rtt = 27.674048 ms.

===8.8.8.8 ping statistics===

2 packets sent, 2 packets received, 0.000000 percent packet loss. Total time: 2069.648902 ms.
```

3. Now let's check if the files are created.

```
~$ ls
Desktop Documents Downloads Music Pictures ping ping.c Public receive.txt request.txt snap Templates Videos
```

4. Let's check the content inside the request.txt file. The output shows the source port, source address, ICMP_ECHO, ICMP request message, and ICMP checksum.

```
Source Port: 0
Source Address: 134744072
ICMP_ECHO: 8
ICMP_Request_Message 0123456789::<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~^?<80><81><82><83><84><85>
<89><90><91><92><93><94><95><96><97><98><99> ;Ç£¤¥¦§¨ª«¬®¯°±²³´µ¶·¸¹º»¼½¾¿ÀÁÂÃÄÅÆÇÈÉÊËÌÍÎÏÐÑÒÓÔÕÖ×ØÙÚÛÜÝÞßàáâãäåæçèéêëìíîïðñóôõö÷øùúûüýþÿ
ICMP_Request_checksum 30969
ICMP_ECHO: 8
ICMP_Request_Message 0123456789::<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~^?<80><81><82><83><84><85>
<89><90><91><92><93><94><95><96><97><98><99> ;Ç£¤¥¦§¨ª«¬®¯°±²³´µ¶·¸¹º»¼½¾¿ÀÁÂÃÄÅÆÇÈÉÊËÌÍÎÏÐÑÒÓÔÕÖ×ØÙÚÛÜÝÞßàáâãäåæçèéêëìíîïðñóôõö÷øùúûüýþÿ
ICMP_Request_checksum 30968
```

5. Let's check the content inside the receive.txt file. It shows the messages of the reply but not the statistics.

```
~$ cat receive.txt
256 bytes from dns.google (h: 8.8.8.8) (8.8.8.8) msg_seq=2 ttl=64 rtt = 27.674048 ms.
256 bytes from dns.google (h: 8.8.8.8) (8.8.8.8) msg_seq=1 ttl=64 rtt = 27.461418 ms.
```

