Computer Networks Lab
Model Examinations

Shivanirudh S G
CSE-C
185001146

1. Write a socket program to perform the following using UDP.
a.Server maintains the mapping of webpage with the ip address.
b.Client sends the request for a webpage.
c.Server checks the table and responds with the appropriate ip address.
d.if not found send an error message to the client.

Table structure:

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<sys/types.h>
#include<sys/socket.h>
#include<netinet/in.h>
#include<unistd.h>
#include<arpa/inet.h>

#define ADDR_LIMIT 5
#define DOMAIN_LIMIT 10

//Record for each domain-address pair
struct record{
char *domain;
char *address[ADDR_LIMIT];
};

typedef struct record Record;

void initialize(Record *r){
r->domain = (char*)calloc(100, sizeof(char));
for(int i =0;i<ADDR_LIMIT;i++)
r->address[i] = (char*)calloc(100, sizeof(char));
}

//Print table
void table(Record table[DOMAIN_LIMIT]){
printf(" _____ \n");
printf("|___Domain Name___|_______Address______|\n");
for (int i = 0; i < DOMAIN_LIMIT; i++){
if (table[i].domain[0]){
```

```
printf("| %-15s | %-20s |\n", table[i].domain, table[i].address[0]);

for (int j = 1; j < ADDR_LIMIT && table[i].address[j][0]; j++)
printf("| %-15s | %-20s |\n", "", table[i].address[j]);
printf("|_________________|____________________|\n");
}
}
printf("\n");
}

//Check if newly specified address is valid
int checkAddress(Record *table, char *address){
char* addr_copy = (char*)calloc(100, sizeof(char));
strcpy(addr_copy, address);
char *split;
int val;
split = strtok(addr_copy, ".");
//Check if all octets lie within 0 and 255.
while (split){
val = atoi(split);
if (val < 0 || val > 255){
printf("\nError: Invalid Address - Octet out of range.\n");
return 0;
}
split = strtok(NULL, ".");
}

//Check if new address already exists in the table
for (int i = 0; i < DOMAIN_LIMIT; i++){
if (!table[i].domain[0])
continue;

for (int j = 0; j < ADDR_LIMIT && table[i].address[j][0]; j++)
if (strcmp(address, table[i].address[j]) == 0){
printf("\nError: IP address already exists.\n");
return 0;
}
}

return 1;
}

//Create domain-address pair in the table
int createRecord(Record table[DOMAIN_LIMIT], char *domain, char *address){
int ix = -1;
int flag = 0;
//Check if entry exists already
int addr_valid = checkAddress(table, address);
if (!addr_valid)
return flag;
```

```
for (int i = 0; i < DOMAIN_LIMIT; i++){
if (strcmp(table[i].domain, domain) == 0){
for (int j = 0; j < DOMAIN_LIMIT; j++){
if (!table[i].address[j][0]){
strcpy(table[i].address[j], address);
flag = 1;
break;
}
}
break;
}
if (!table[i].domain[0] && ix == -1)
ix = i;
}

// If record can be created
if (!flag){
strcpy(table[ix].domain, domain);
strcpy(table[ix].address[0], address);
flag = 1;
}

return flag;
}

char *getAddress(Record *table, char *domain){
char* addresses = (char*)calloc(ADDR_LIMIT*20, sizeof(char));

for (int i = 0; i < DOMAIN_LIMIT; i++){
if (strcmp(table[i].domain, domain) == 0){
for (int j = 0; j < ADDR_LIMIT; j++) {
strcat(addresses, table[i].address[j]);
strcat(addresses, " ");
}
break;
}
}
return addresses;
}
```

Server:

```
#include "table.h"

int main(int argc, char **argv){
Record webpage_table[DOMAIN_LIMIT];
for(int i = 0;i<DOMAIN_LIMIT;i++)
initialize(&webpage_table[i]);

if(argc > 1){
```

```
printf("\n No arguments needed for server. ");
exit(0);
}

//UDP connection addresses
struct sockaddr_in server_address, client_address;
//Buffer to handle messages
char buffer[1024];
//Parameters for communication
char* webpage = (char*)calloc(100, sizeof(char));
char* address = (char*)calloc(100, sizeof(char));

//Socket fd: IPv4, UDP, IP
int sockfd = socket(AF_INET, SOCK_DGRAM, 0);
if(sockfd<0)
perror("\n Error: Unable to create connection");
bzero(&server_address, sizeof(server_address));

server_address.sin_family = AF_INET; //IP family
server_address.sin_addr.s_addr = htonl(INADDR_ANY); //Any free address
server_address.sin_port = htons(5555); //Port number 5555

//Bind socket and fd
if(bind(sockfd, (struct sockaddr *)&server_address, sizeof(server_address)) < 0){
perror("\n Error: Binding error");
}

int len = sizeof(client_address);

//Create records in table
createRecord(webpage_table, "google.com", "13.14.168.192");
createRecord(webpage_table, "ssn.edu.in", "77.88.99.100");
createRecord(webpage_table, "youtube.com", "92.68.3.4");
createRecord(webpage_table, "ssn.edu.in", "99.111.223.123");

//Modify table at runtime
char option = 'n';
do{
table(webpage_table);
printf("\n Modify table? (y/n) ");scanf(" %c", &option);
if(option == 'y' || option == 'Y'){
printf("\n Enter webpage name: "); scanf(" %[^\n]", webpage);
printf("\n Enter address: "); scanf(" %[^\n]", address);
int result = createRecord(webpage_table, webpage, address);
if(result){
printf("\n Table modified successfully\n");
}
}
}while(option == 'y'|| option == 'Y');

printf("\nWebpage server set up complete. \n");
```

```
char *response = (char*)calloc(ADDR_LIMIT*2, sizeof(char));
while(1){
bzero(&buffer, sizeof(buffer));
recvfrom(sockfd, buffer, sizeof(buffer), MSG_WAITALL, (struct sockaddr *)&client_address,
&len);
strcpy(response, getAddress(webpage_table, buffer));
sendto(sockfd, response, sizeof(buffer), MSG_CONFIRM, (struct sockaddr *)&client_address,
len);
}
close(sockfd);
}
```

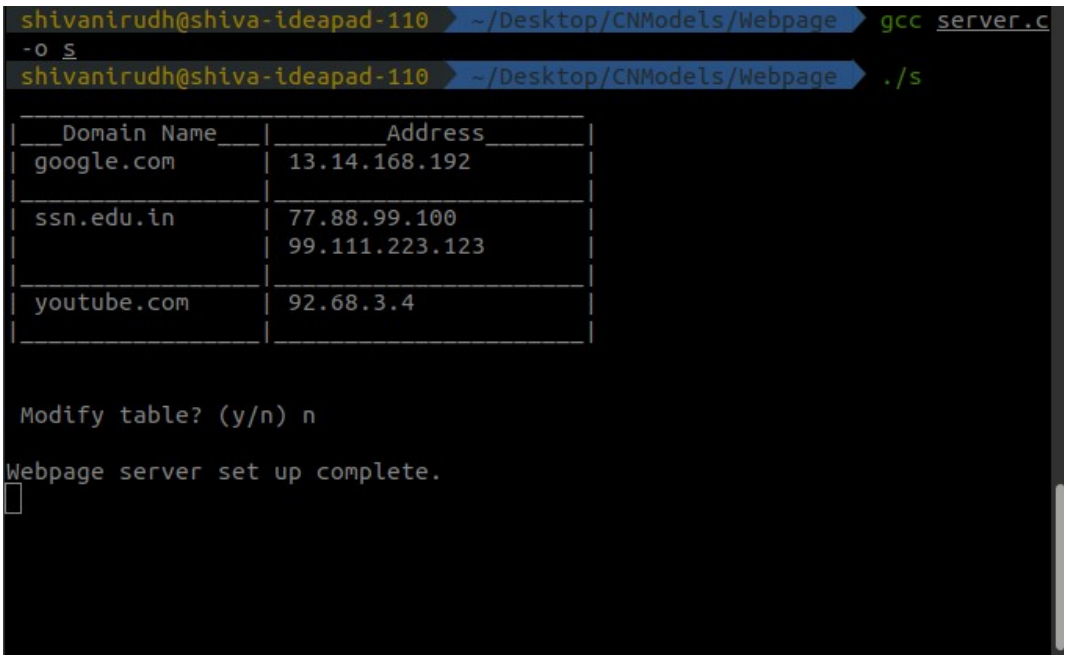
Client:

```
#include "table.h"

int main(int argc, char **argv){

if(argc < 2){
perror("\n Error: Client needs server IP address.");
exit(0);
}

struct sockaddr_in server_address;
char buffer[1024];
Record *query = (Record*)malloc(sizeof(Record));

int sockfd = socket(AF_INET, SOCK_DGRAM, 0);
if(sockfd<0)
perror("\n Error: Unable to create connection");
bzero(&server_address, sizeof(server_address));

server_address.sin_family = AF_INET; //IP family
server_address.sin_addr.s_addr = htonl(INADDR_ANY); //Any free address
server_address.sin_port = htons(5555); //Port number 5555

int len = sizeof(Record);
while(1){
initialize(query);

printf("\n Enter webpage name: ");scanf(" %[^\n]", query->domain);
if(strcmp(query->domain, "end") == 0){
break;
}

//Request for webpage
sendto(sockfd, query->domain, sizeof(buffer), MSG_CONFIRM, (struct sockaddr
*)&server_address, sizeof(server_address));
```

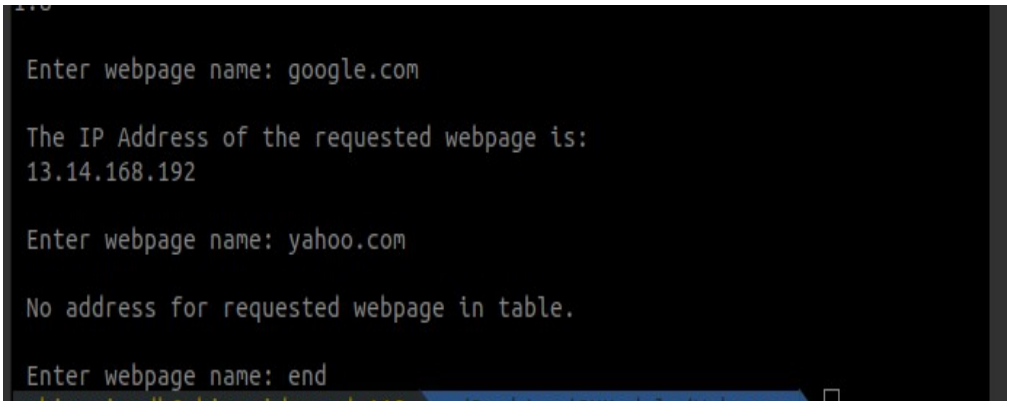
```
bzero(&buffer, sizeof(buffer));
recvfrom(sockfd, buffer, sizeof(buffer), MSG_WAITALL, (struct sockaddr *)&server_address,
&len);
char* split = strtok(buffer, " ");
if(split){
printf("\n The IP Address of the requested webpage is: ");
while(split){
printf("\n %s", split);
split = strtok(NULL, " ");
}
printf("\n");
}
else{
printf("\n No address for requested webpage in table.\n");
}
}

close(sockfd);
}
```

Input/Output:

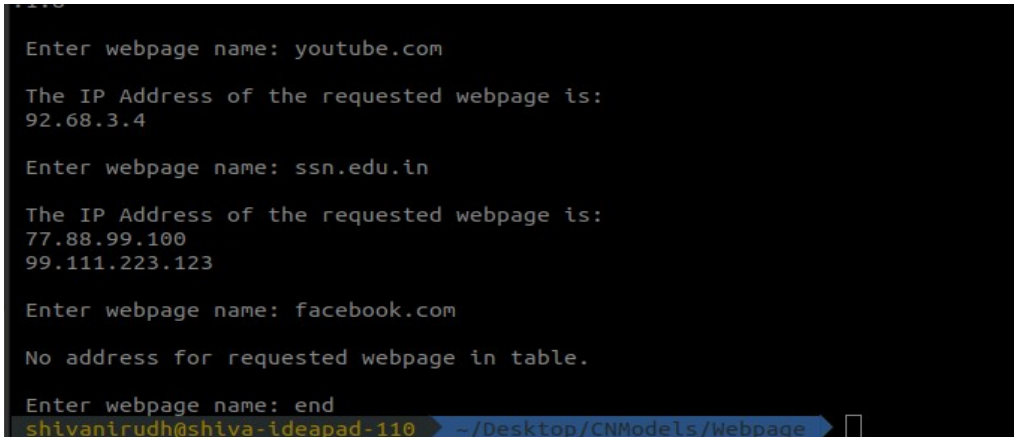
Server:



Client 1:



Client 2:



2. Use simulator to analyse the bottleneck problem in TCP and UDP.Assume Blue for TCP and Red for UDP.

```
set ns [new Simulator]

$ns color 1 blue
$ns color 2 red

set n1 [$ns node]
set n2 [$ns node]
set n3 [$ns node]
set n4 [$ns node]
set n5 [$ns node]
set n6 [$ns node]

set f [open out.tr w]
$ns trace-all $f
set nf [open out.nam w]
$ns namtrace-all $nf

$ns duplex-link $n1 $n3 2Mb 10ms DropTail
$ns duplex-link $n2 $n3 2Mb 10ms DropTail
$ns simplex-link $n3 $n4 0.3Mb 100ms DropTail
$ns simplex-link $n4 $n3 0.3Mb 100ms DropTail
$ns duplex-link $n4 $n5 0.5Mb 40ms DropTail
$ns duplex-link $n4 $n6 0.5Mb 40ms DropTail

$ns duplex-link-op $n1 $n3 orient right-down
$ns duplex-link-op $n2 $n3 orient right-up
$ns simplex-link-op $n3 $n4 orient right
$ns simplex-link-op $n4 $n3 orient left
$ns duplex-link-op $n4 $n5 orient right-up
$ns duplex-link-op $n4 $n6 orient right-down

$ns queue-limit $n3 $n4 10

set tcp [new Agent/TCP/Newreno]
$ns attach-agent $n2 $tcp
set sink [new Agent/TCPSink/DelAck]
$ns attach-agent $n6 $sink
$ns connect $tcp $sink
$tcp set fid_ 1
$tcp set window_ 8000
$tcp set packetSize_ 512
$tcp set class_ 1

set ftp [new Application/FTP]
$ftp attach-agent $tcp

set udp [new Agent/UDP]
$ns attach-agent $n1 $udp
$udp set class_ 2

set cbr [new Application/Traffic/CBR]
$cbr attach-agent $udp
$cbr set type_ CBR
$cbr set packet_size_ 1024
$cbr set rate_ 0.01mb
```

```
$cbr set random_ false

set null [new Agent/Null]
$ns attach-agent $n5 $null
$ns connect $udp $null

$ns at 0.1 "$cbr start"
$ns at 0.5 "$ftp start"
$ns at 4.7 "$ftp stop"
$ns at 5.0 "$cbr stop"

$ns at 7.0 "finish"

proc finish {} {
        global ns f nf
        $ns flush-trace
        close $f
        close $nf

        puts "running nam..."
        exec nam out.nam &
        exit 0
}

$ns run
```

Output:

Average Throughput[kbps] = 7.92
Amount of data transferred in CBR[kb] = 34.00
StartTime = 0.10
StopTime = 4.39
Amount of data dropped in CBR[kb] = 1.00