Ex4-computer networking

By Ariel Zidon 314789264 and Itamar Kraitman 208925578

Part A:

1. One of the main benefits is that DoH centralizes traffic to a few DoH servers, which improves load time performance.

2. A. Firstly, DoH encryption allows admins to see the DNS traffic while in other techniques (such as DoT) the encryption protects the data from other admins.
   B. Secondly, using DoH changes the browsing experience. If you're not familiar with DoH properly, using it can cause some blocked queries and some other security issues.

3. We chose to give a solution to the first problem we raised. Using a proxy server can get around the problem, the admins will get the DNS traffic of the proxy server instead of the user DNS traffic, which increases the security of using DoH.

4.

App-level	
Advantages	**Disadvantages**
App-level DoH uses its own format of queries.	The user may not be informed that he skipped the DoH which may cause some problems such as broken content.

Proxy server	
Advantages	**Disadvantages**
It uses methods for DNS queries and it receives replies by reaching DoH servers.	The end-user can watch it.

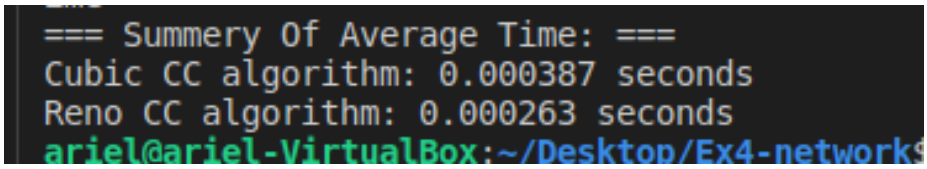
Local proxy server	
Advantages	**Disadvantages**
The queries are sent to a local proxy, which gives another layer of protection	Local proxy needs to be installed on each machine separately

DoH plugin	
Advantages	**Disadvantages**
Is not depend on specific implementation	Required installing/uninstalling and loses flexibility (only DoH can be used)

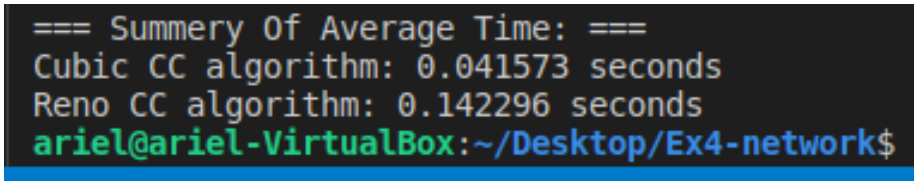
5. One main advantage of DoH over Do53 is that Do53 uses TCP\UDP protocols. This means, that in cases of a small amount (25 queries in this case) of data (such as loading a web file) Do53 would prefer to use UDP which may cause data loss in a network that allows packets loss. On the other hand, DoH uses TCP protocol which means data loss is not allowed so it ensures that the file will be loaded faster and without packets loss.

Part B:

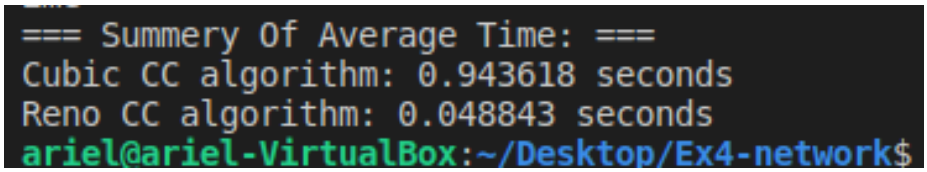
1. Regular run:



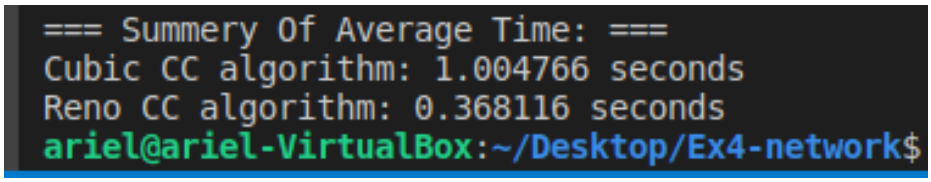
2. 10% loss run:



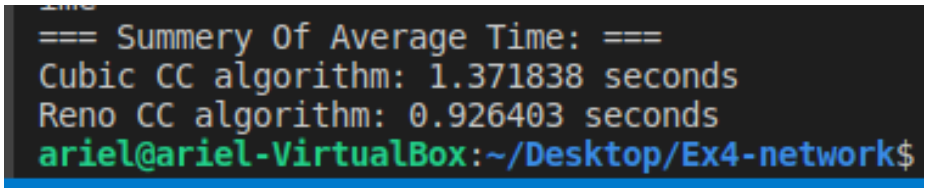
3. 15% loss run:



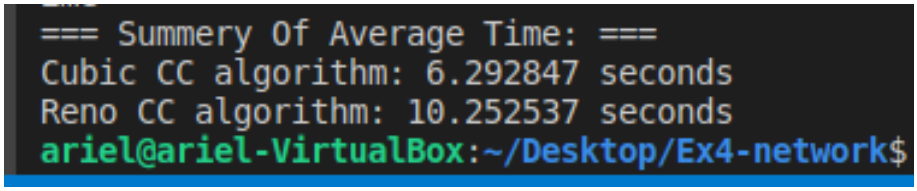
4. 20% loss run:



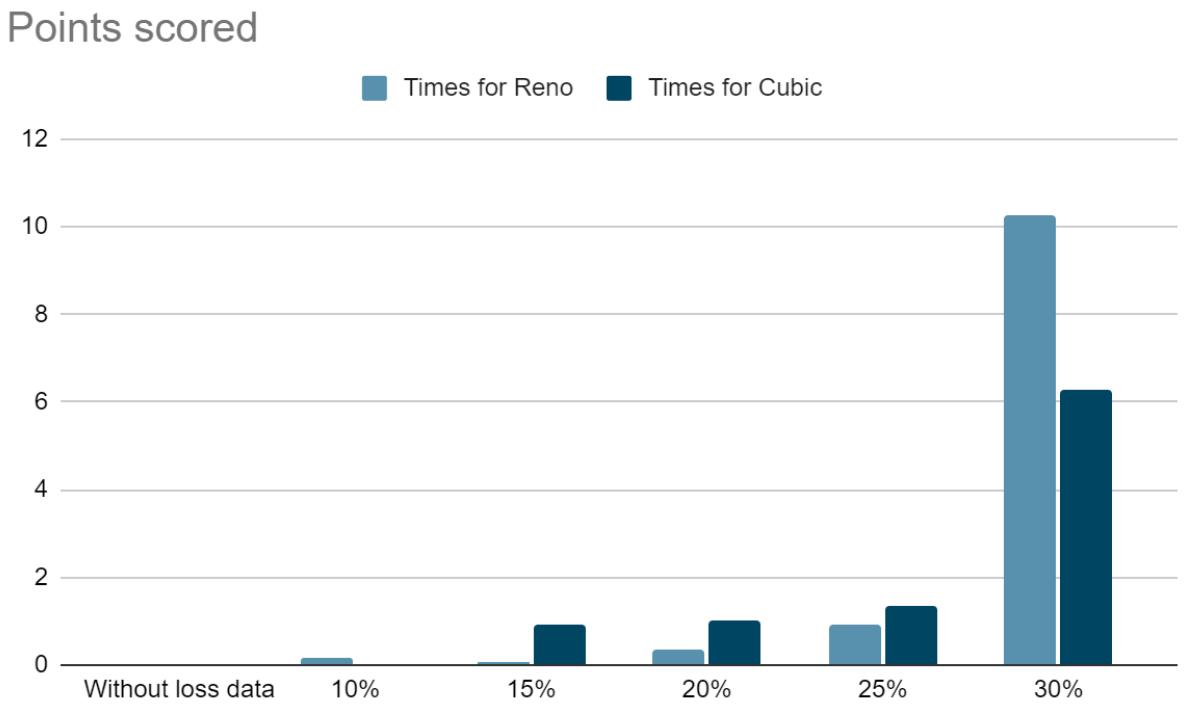
5. 25% loss run:



6. 30% loss run:



Percent(%) of loss data	Times for Cubic	Times for Reno
Without loss data	0.000387	0.000263
10%	0.041573	0.142296
15%	0.943618	0.048843
20%	1.004766	0.368116
25%	1.371838	0.926403
30%	6.292847	10.252537



As can be seen, we found out that cubic is a better algorithm. In terms of performance cubic grows slower than relo. Better than that, relo's run time doubles itself as the amount of loss data grows (from 25% to 30% it grows about 10 times!) when cubic grows much slower. We assume that the bump in the runtime in 30% loss caused by network hiccups. Therefore, we can conclude that cubic is a better algorithm and it is the default algorithm for a reason.

Sender:

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include <sys/socket.h>
#include <sys/types.h>

#include <netinet/in.h>
#include <netinet/tcp.h>

#include <unistd.h>
#include <arpa/inet.h>

#define PORT 6769
#define ADDRESS "127.0.0.1"
#define SIZE 1048576 // 1024 * 1024 * 100 = 100MB of data
#define MTU 1500

// void send_file(FILE *fp, int sock); // declaring this function for
later use, no header file needed here

int main()
{
    // creating a TCP socket
    int sock;
    struct sockaddr_in server_address;
    server_address.sin_family = AF_INET;
    server_address.sin_port = htons(PORT);
    int rval = inet_pton(AF_INET, (const char *)ADDRESS,
&server_address.sin_addr);
    if (rval <= 0)
    {
        perror("rvel");
        return -1;
    }

    // setting up buffer for CC algo switch later on
    char buf[256];
```

```
    socklen_t len;

    // defining file pointers
    FILE *fp;
    char *filename = "1mb.txt";
    char buffer[MTU];

    // recall socket is 'sock'
    int j, r;
    size_t n;
    for (r = 1; r <= 5; r++)
    {
        sock = socket(AF_INET, SOCK_STREAM, 0);
        int conn_status = connect(sock, (struct sockaddr
*)&server_address, sizeof(server_address));
        if (conn_status < 0)
        {
            perror("conn_status");
            printf("\n");
            exit(1);
        }
        len = sizeof(buf);
        if (getsockopt(sock, IPPROTO_TCP, TCP_CONGESTION, buf, &len) != 0)
        {
            perror("getsockopt");
            return -1;
        }

        fp = fopen(filename, "r");
        bzero(buffer, sizeof(buffer));
        while ((n = fread(buffer, 1, sizeof buffer, fp)) > 0)
        {
            j = send(sock, buffer, sizeof(buffer), 0);
            if (j < 0)
            {
                perror("send");
                exit(1);
            }
        }
        if (ferror(fp))
```

```
        {
            perror("Error: ");
        }
        close(sock);
        fclose(fp);
    }

    // END of send segment

    // send_file(fp, sock);
    printf("Sent Data 5 times using cubic CC algorithm\n");
    printf("Switching To Reno\n");

    // Switching the CC algorithm to be Reno
    // The code bit for switching algorithms is courtesy of StackOverflow

    for (r = 1; r <= 5; r++)
    {
        sock = socket(AF_INET, SOCK_STREAM, 0);
        int conn_status = connect(sock, (struct sockaddr
*)&server_address, sizeof(server_address));
        if (conn_status < 0)
        {
            perror("conn_status");
            printf("\n");
            exit(1);
        }
        strcpy(buf, "reno");
        len = strlen(buf);
        if (setsockopt(sock, IPPROTO_TCP, TCP_CONGESTION, buf, len) != 0)
        {
            perror("setsockopt");
            printf("\n");

            return -1;
        }
        len = sizeof(buf);
        if (getsockopt(sock, IPPROTO_TCP, TCP_CONGESTION, buf, &len) != 0)
        {
            perror("getsockopt");
```

```
            printf("\n");

            return -1;
        }
        fp = fopen(filename, "r");
        bzero(buffer, sizeof(buffer));
        while ((n = fread(buffer, 1, sizeof buffer, fp)) > 0)
        {
            j = send(sock, buffer, sizeof(buffer), 0);
            if (j < 0)
            {
                perror("send");
                exit(1);
            }
        }
        if (ferror(fp))
        {
            perror("Error: ");
        }
        close(sock);
        fclose(fp);
    }
    printf("Sent Data 5 times using reno CC algorithm\n");
    printf("Closing..\n");

    // Closing the socket
    return 0;
}
```

Measure:

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <signal.h>
#include <unistd.h>
#include <stdbool.h>

#include <sys/socket.h>
#include <sys/types.h>
#include <sys/time.h>
#include <arpa/inet.h>

#include <netinet/in.h>
#include <netinet/tcp.h>

#define PORT 6769
#define ADDRESS "127.0.0.1"
#define SIZE 1048576
#define MTU 1024

int main()
{
    // Setting up variables for later use
    int conn_status;
    int sock, sock_recv;
    struct sockaddr_in server_addr;
    socklen_t length;
    double average_time_cubic;
    double average_time_reno;

    char buf[256]; // for CC algorithm change
    char buffer[MTU];

    struct timeval start, end;
    sock = socket(AF_INET, SOCK_STREAM, 0);
    if (sock < 0)
    {
        perror("socket");
```

```
    }

    // Default CC algorithm is Cubic, we just make sure this is the
correct now
    // So there will be no issues with recieving from sender.

    length = sizeof(buf);
    if (getsockopt(sock, IPPROTO_TCP, TCP_CONGESTION, buf, &length) != 0)
    {
        perror("getsockopt");
        return -1;
    }

    length = sizeof(buf);
    if (getsockopt(sock, IPPROTO_TCP, TCP_CONGESTION, buf, &length) != 0)
    {
        perror("getsockopt");
        return -1;
    }
    printf("Congestion Control Strategy: %s\n", buf);

    // Setting server_addr memory to 0 in order to make sure we open a
fresh port without overlapping
    // any existing data
    memset(&server_addr, 0, sizeof(server_addr));

    // Setting up socket data
    server_addr.sin_family = AF_INET;
    server_addr.sin_port = htons(PORT);
    server_addr.sin_addr.s_addr = INADDR_ANY;

    conn_status = bind(sock, (struct sockaddr *)&server_addr,
sizeof(server_addr));
    if (conn_status < 0)
    {
        perror("bind");
        printf("\n");
        close(sock);
        exit(1);
    }
```

```
    /*
    Next block of code is setting up variables to measure time as
requested in the assignment,
    We will be using the <sys/time.h> library to do all the required
calculations.
    Afterwards, we will switch to Reno CC algorithm and take the same
measurements.
    */
    int e;
    int recieved = 0;

    for (int i = 0; i < 5; i++)
    {
        e = listen(sock, 10);
        if (e < 0)
        {
            perror("listen");
        }
        sock_recv = accept(sock, NULL, NULL);
        if (sock_recv < 0)
        {
            perror("accept");
            exit(1);
        }
        int n = 0;
        gettimeofday(&start, 0);

        while ((n = recv(sock_recv, &buffer, sizeof(buffer), 0)) > 0)
        {
            recieved += n;
            if (recieved == SIZE * 30)
            {
                break;
            }
        }
        gettimeofday(&end, 0);
        average_time_cubic += (end.tv_sec - start.tv_sec) + ((end.tv_usec
- start.tv_usec) / 1e6);
```

```
        bzero(buffer, MTU);
        close(sock_recv);
    }
    printf("Recieved message 5 times, switching CC algorithm..\n");
    // Changing CC algorithm to Reno

    strcpy(buf, "reno");
    length = strlen(buf);
    if (setsockopt(sock, IPPROTO_TCP, TCP_CONGESTION, buf, length) != 0)
    {
        perror("setsockopt");
        return -1;
    }

    length = sizeof(buf);

    if (getsockopt(sock, IPPROTO_TCP, TCP_CONGESTION, buf, &length) != 0)
    {
        perror("getsockopt");
        return -1;
    }
    printf("New Congestion Control Strategy: %s\n", buf);

    // Same logic as above for measuring average time, this time for Reno
CC algorithm
    for (int i = 0; i < 5; i++)
    {
        e = listen(sock, 10);
        if (e < 0)
        {
            perror("listen");
        }
        sock_recv = accept(sock, NULL, NULL);
        if (sock_recv < 0)
        {
            perror("accept");
            exit(1);
        }
        int n = 0;
        int recieved = 0;
```

```
        gettimeofday(&start, 0);
        while ((n = recv(sock_recv, &buffer, sizeof(buffer), 0)) > 0)
        {
            recieved += n;
            if (recieved == SIZE * 30)
            {
                break;
            }
        }
        gettimeofday(&end, 0);
        average_time_reno += (end.tv_sec - start.tv_sec) + ((end.tv_usec -
start.tv_usec) / 1e6);
        bzero(buffer, MTU);
        close(sock_recv);
    }
    printf("Recieved message 5 times using Reno, calculating average
delivery time\n");
    printf("=== Summery Of Average Time: ===\n");
    printf("Cubic CC algorithm: %f seconds\n", average_time_cubic / 5);
    printf("Reno CC algorithm: %f seconds\n", average_time_reno / 5);
    close(sock);
    return 0;
}
```

```
MakeFile:

.PHONY: clean all

all: sender measure

sender: sender.o
    gcc -Wall -g -o sender sender.o

measure : measure.o
    gcc -Wall -g -o measure measure.o

sender.o: sender.c
    gcc -Wall -g -c sender.c
```

```
measure.o: measure.c
    gcc -Wall -g -c measure.c

clean:
    rm -f *.o *.a *.so sender measure
```