1 ）编程实现web服务器

C++ 代码

#include <cstring>

#include <fstream>

#include <iostream>

#include <netinet/in.h>

#include <sstream>

#include <string>

#include <sys/socket.h>

#include <unistd.h>

using namespace std;

int main() {

int server\_fd, new\_socket, valread;

struct sockaddr\_in address;

int opt = 1;

int addrlen = sizeof(address);

char buffer[1024] = {0};

string hello = "HTTP/1.1 200 OK\nContent-Type: "

"text/plain\nContent-Length: 12\n\nHello world!";

// 创建套接字

if ((server\_fd = socket(AF\_INET, SOCK\_STREAM, 0)) == 0) {

perror("socket failed");

exit(EXIT\_FAILURE);

}

// 设置套接字选项

if (setsockopt(server\_fd, SOL\_SOCKET, SO\_REUSEADDR | SO\_REUSEPORT, &opt,

sizeof(opt))) {

perror("setsockopt failed");

exit(EXIT\_FAILURE);

}

// 绑定套接字到指定端口

address.sin\_family = AF\_INET;

address.sin\_addr.s\_addr = INADDR\_ANY;

address.sin\_port = htons(8080);

if (bind(server\_fd, (struct sockaddr \*)&address, sizeof(address)) < 0) {

perror("bind failed");

exit(EXIT\_FAILURE);

}

// 监听连接请求

if (listen(server\_fd, 3) < 0) {

perror("listen failed");

exit(EXIT\_FAILURE);

}

// 接受连接请求并处理请求

while (true) {

if ((new\_socket = accept(server\_fd, (struct sockaddr \*)&address,

(socklen\_t \*)&addrlen)) < 0) {

perror("accept failed");

exit(EXIT\_FAILURE);

}

valread = read(new\_socket, buffer, 1024);

printf("%s\n", buffer);

write(new\_socket, hello.c\_str(), hello.size());

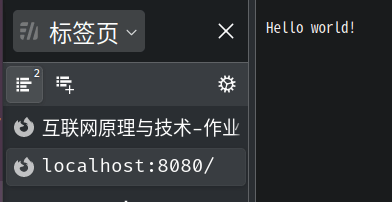
close(new\_socket);

}

return 0;

}

结果：



2）编程实现UDPping

C++ 代码

#include <arpa/inet.h>

#include <cstdio>

#include <cstdlib>

#include <cstring>

#include <ctime>

#include <iostream>

#include <netinet/in.h>

#include <ostream>

#include <sys/socket.h>

#include <unistd.h>

using namespace std;

const int MAX\_PACKET\_SIZE = 4096;

const int DEFAULT\_PORT = 12345;

const int DEFAULT\_TIMEOUT = 1; // seconds

// 计算时间差（毫秒）

int diff\_ms(struct timespec t1, struct timespec t2) {

return (t2.tv\_sec - t1.tv\_sec) \* 1000 + (t2.tv\_nsec - t1.tv\_nsec) / 1000000;

}

int main(int argc, char \*argv[]) {

int port = DEFAULT\_PORT;

int timeout = DEFAULT\_TIMEOUT;

if (argc < 2) {

cout << "Usage: " << argv[0] << " <ip\_address> [port] [timeout]" << endl;

return 1;

}

const char \*ip\_address = argv[1];

if (argc >= 3) {

port = atoi(argv[2]);

}

if (argc >= 4) {

timeout = atoi(argv[3]);

}

int sockfd = socket(AF\_INET, SOCK\_DGRAM, 0);

if (sockfd < 0) {

perror("socket");

return 1;

}

struct sockaddr\_in addr;

memset(&addr, 0, sizeof(addr));

addr.sin\_family = AF\_INET;

addr.sin\_port = htons(port);

if (inet\_pton(AF\_INET, ip\_address, &addr.sin\_addr) <= 0) {

perror("inet\_pton");

return 1;

}

char sendbuf[MAX\_PACKET\_SIZE];

char recvbuf[MAX\_PACKET\_SIZE];

int seq = 0;

while (true) {

// 构造发送数据包

sprintf(sendbuf, "PING %d %ld\r\n", seq++, time(NULL));

// 发送数据包

struct timespec send\_time;

clock\_gettime(CLOCK\_MONOTONIC, &send\_time);

if (sendto(sockfd, sendbuf, strlen(sendbuf), 0, (struct sockaddr \*)&addr,

sizeof(addr)) < 0) {

perror("sendto");

return 1;

}

// 接收数据包

struct sockaddr\_in from\_addr;

socklen\_t from\_len = sizeof(from\_addr);

fd\_set readfds;

FD\_ZERO(&readfds);

FD\_SET(sockfd, &readfds);

struct timeval tv;

tv.tv\_sec = timeout;

tv.tv\_usec = 0;

int ret = select(sockfd + 1, &readfds, NULL, NULL, &tv);

if (ret < 0) {

perror("select");

return 1;

} else if (ret == 0) {

cout << "Request timed out" << endl;

continue;

}

int n = recvfrom(sockfd, recvbuf, sizeof(recvbuf), 0,

(struct sockaddr \*)&from\_addr, &from\_len);

if (n < 0) {

perror("recvfrom");

return 1;

}

recvbuf[n] = '\0';

// 解析接收数据包

int id, timestamp;

if (sscanf(recvbuf, "PONG %d %d", &id, &timestamp) == 2) {

struct timespec recv\_time;

clock\_gettime(CLOCK\_MONOTONIC, &recv\_time);

int rtt = diff\_ms(send\_time, recv\_time);

cout << "PONG received from " << inet\_ntoa(from\_addr.sin\_addr)

<< ": seq=" << id << " time=" << rtt << "ms" << endl;

} else {

cout << "Invalid response: " << recvbuf << endl;

}

sleep(1);

}

close(sockfd);

return 0;

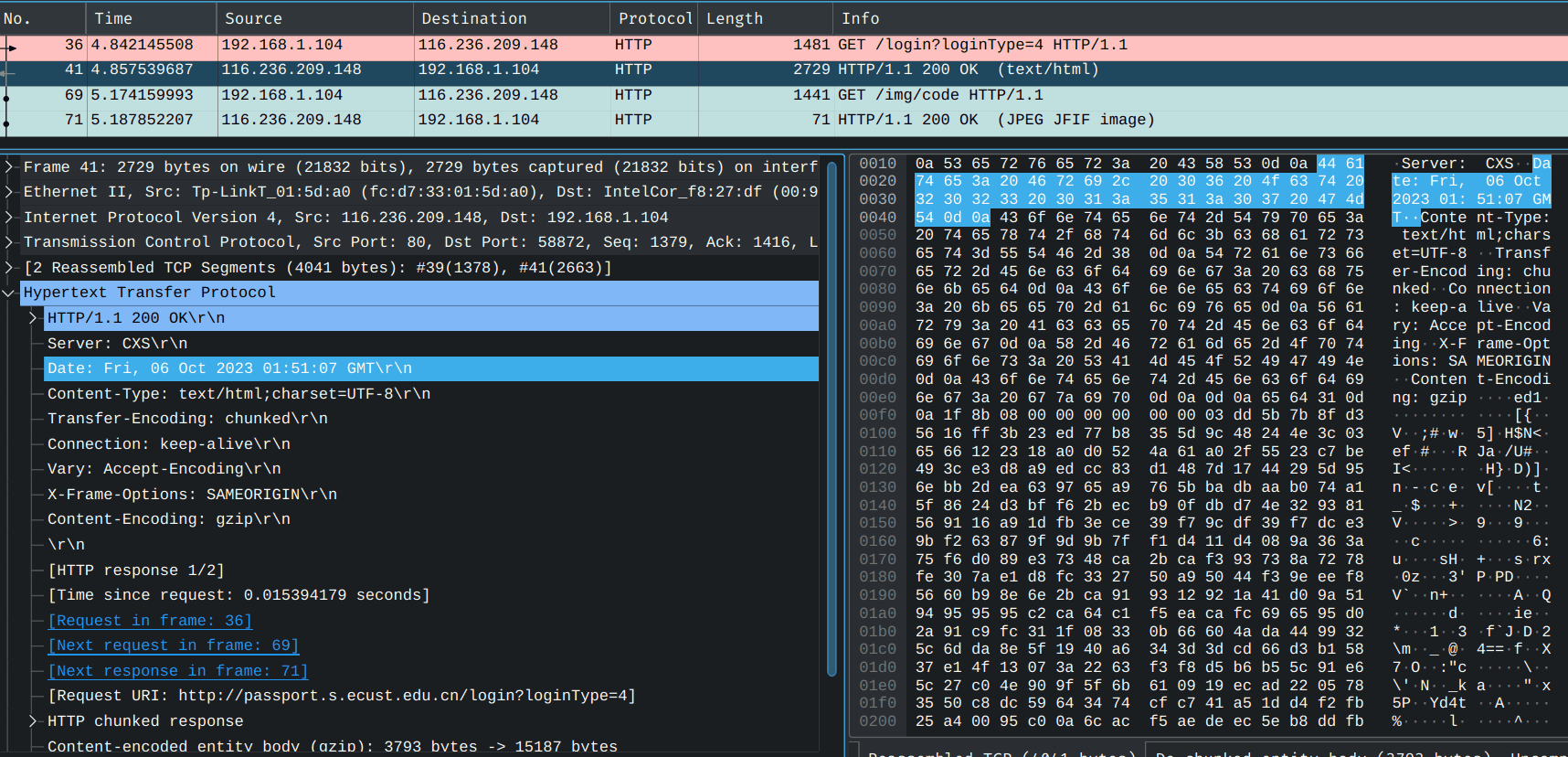
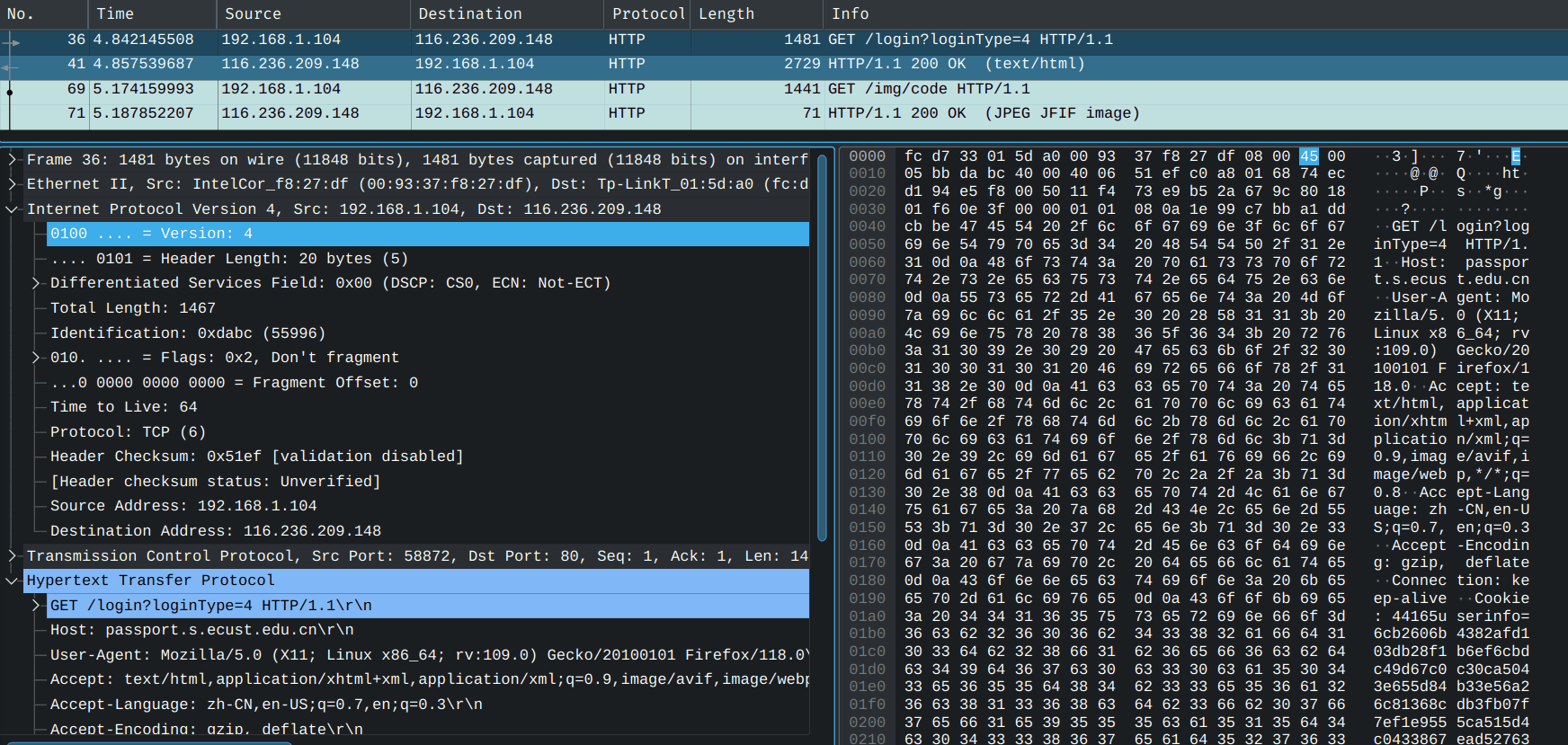
}

3）Wireshark实验：http

打开浏览器并清空浏览器缓存。访问web网站 （http或https）

Wireshark 中启动数据包捕获

结果：



回答：

1 您的浏览器是否运行 HTTP 版本 1.0 或 1.1，2?服务器运行的是什么版本的 HTTP?

客户端：HTTP 1.1 服务端：HTTP 1.1

2 如果是http2，请给出并分析http格式

不是

3 web服务器互联网地址是什么? 您的计算器的 Internet 地址是什么?

服务器 116.236.209.148 客户端 192.168.1.104 （内网）

4 从服务器返回到浏览器的状态代码是什么?

200

5 在服务器上最后修改的 HTML 文件是什么时候?

响应头中不存在 "Last-Modified" 字段；而 Date 字段显示 “Fri, 06 Oct 2023 01:51:07 GMT” 可能并非 HTML 文件的最后修改时间。

6 如果是https，请给出并分析SSL过程

1. 客户端向服务器发起HTTPS请求，请求中包含一个随机数和支持的加密算法列表。

2. 服务器收到请求后，向客户端返回一个数字证书，证书中包含了服务器的公钥和证书的颁发机构等信息。

3. 客户端收到数字证书后，会验证证书的合法性，包括证书是否过期、证书颁发机构是否可信等。如果验证通过，客户端会生成一个随机数，使用服务器公钥加密后发送给服务器。

4. 服务器收到客户端发送的加密随机数后，使用自己的私钥进行解密，得到客户端发送的随机数。

5. 服务器和客户端使用这两个随机数生成一个对称密钥，用于加密和解密数据传输。

6. 服务器和客户端开始使用对称密钥加密和解密数据传输。

总结：证书验证，通过非对称加密传输随机数算子，再将随机数用于对称加密。

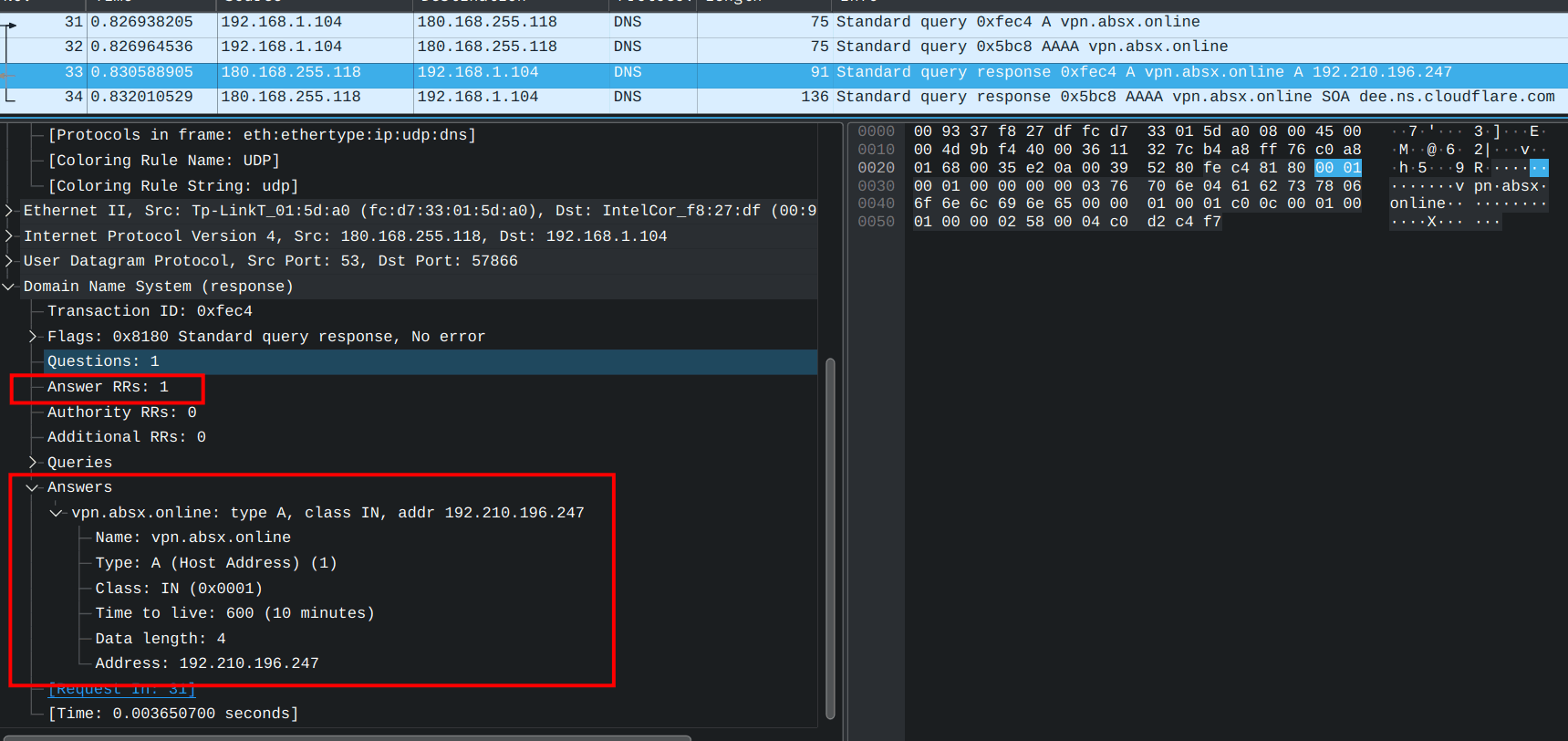
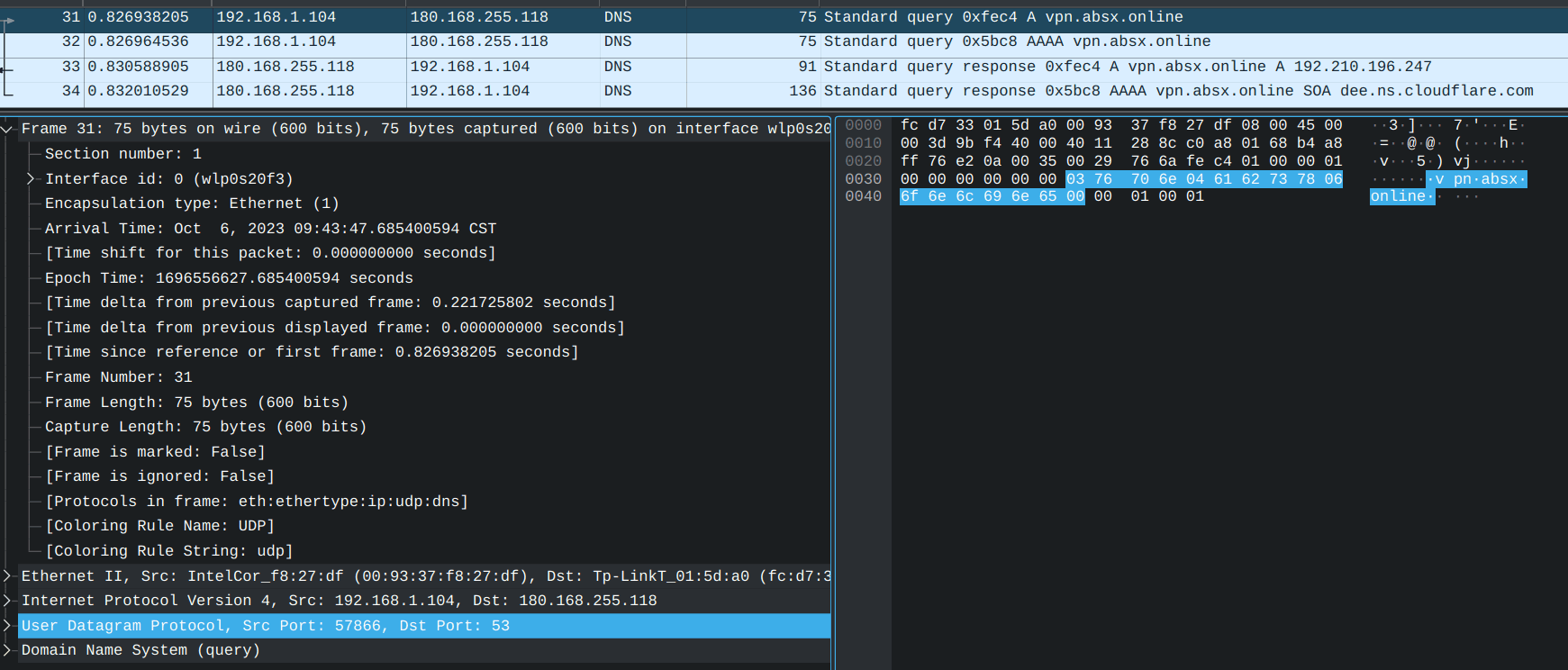
4）Wireshark实验：DNS

ipconfig /flushdns清除缓存

打开浏览器并清空浏览器缓存。访问web网站

Wireshark 中启动数据包捕获

结果：



回答：

1 找到 DNS 查询和响应消息。它们是否通过 UDP 或 TCP 发送?

通过 UDP 发送

2 DNS 查询消息的目标端口是什么? DNS 响应消息的源端口是什么?

目标端口 53 响应源端口 57866

3 DNS 查询消息发送到哪个 IP 地址?使用 ipconfig 来确定本地 DNS 服务器的 IP 地址。这两个 IP 地址是否相同?

发送到180.168.255.118 Arch Linux 默认情况下没有安装本地 DNS 服务器

4 检查 DNS 响应消息。提供了多少个"answers"?这些答案具体包含什么？

1 个 answer

包含了 Name, Type, Class, Time to live, Data length, Address