

Appendix

Appendix A: CommonUtilities.h file

```
#pragma once
#ifndef COMMONUTILITIES_H_
#define COMMONUTILITIES_H_
#include <string>
#include <winsock2.h>
#include <ws2tcpip.h>
#include <iphlpapi.h>
#include <iostream>
#include <stdlib.h>
#include <string.h>
#include <sstream>
#include <time.h>
#include "work.h"
#include "pthread.h"
using namespace std;

class CommonUtilities {
public:
    static pthread_mutex_t mutexPoll;
    static pthread_mutex_t mutexPoll2;
    static pthread_mutex_t mutexCreateSocket;
    int sniffAPacket(const char* target, const char* port,
        string scanType, int protocol, Job* job, SOCKET sockDescProt,
        SOCKET sockDescCMP);

    static SOCKET createRawSocket(int protocol);
    void buildDestIPStruct(struct sockaddr_in* victim, const char* ip, const char* portNumber);
    string getServiceInfo(struct sockaddr_in victim, const char* port);
    string probeSSHVersion(struct sockaddr_in victim);
    string probeWHOISVersion(struct sockaddr_in victim);
    string probeHTTPVersion(struct sockaddr_in victim);
    string probePOPVersion(struct sockaddr_in victim);
    string probeIMAPVersion(struct sockaddr_in victim);
    string probeSMTPVersion(struct sockaddr_in victim);
    bool checkIfIPMatch(const char* ip, struct iphdr* ptrToIPHeader);
    int lookIntoThePacket(const char* ip, const char*
```

```

portNumber, char* ptrToRecievedPacket, string scanType, Job* job);
int parseUDPResponse(const char* ip, const char* portNumber,
unsigned char* ptrToRecievedPacket, Job*);
int parseICMPResponse(const char* ip, const char* portNumber,
unsigned char* sockReadBuffer, Job* job);
int ParseTCPResponse(const char* ip, const char* portNumber,
unsigned char* ptrToRecievedPacket, string scanType, Job* job);
SOCKET bindRawSocket(int protocol, struct sockaddr_in* victim,
const char* ip);
};
#endif /* COMMONUTILITIES_H_ */

```

Appendix B: DnsHeader.h file

```

#pragma once

#ifndef DNS_HEADER_H_
#define DNS_HEADER_H_

typedef struct
{
    unsigned short id;
    unsigned char rd : 1;
    unsigned char tc : 1;
    unsigned char aa : 1;
    unsigned char opcode : 4;
    unsigned char qr : 1;
    unsigned char rcode : 4;
    unsigned char cd : 1;
    unsigned char ad : 1;
    unsigned char z : 1;
    unsigned char ra : 1;
    unsigned short q_count;
    unsigned short ans_count;
    unsigned short auth_count;
    unsigned short add_count;
} DNS_HEADER;

typedef struct
{
    unsigned short qtype;

```

```
        unsigned short qclass;
    } QUESTION;
#endif
```

Appendix C: IcmpHeader.h file

```
//ICMP Header File
#pragma once
#include <cstdint>
struct icmphdr
{
    uint8_t type;
    uint8_t code;
    uint16_t checksum;
    uint16_t id;
    uint16_t seq;
};
```

Appendix D: IPHeader.h file

```
//IP Header File
#pragma once
#ifndef IPHEADER
#define IPHEADER

typedef struct iphdr
{
    unsigned char ip_header_len : 4;
    unsigned char ip_version : 4;
    unsigned char ip_tos;
    unsigned short ip_total_length;
    unsigned short ip_id;
    unsigned char ip_frag_offset : 5;

    unsigned char ip_more_fragment : 1;
    unsigned char ip_dont_fragment : 1;
    unsigned char ip_reserved_zero : 1;
    unsigned char ip_frag_offset1;
    unsigned char ip_ttl;
```

```

    unsigned char  protocol;
    unsigned short ip_checksum;
    unsigned int   ip_srcaddr;
    unsigned int   daddr;
} IPV4_HDR, * PIPV4_HDR, * LPIP4_HDR, IPHeader;
#endif

```

Appendix E: OptionsClass.h file

```

#pragma once

#include <stdio.h>
#include <iostream>
#include <string>
#include <map>
#include <vector>
#include <algorithm>
#include <iterator>
#include <string.h>
#include <winsock2.h>
#include <ws2tcpip.h>
#include <list>
#include <sstream>
#include "getopt.h"
#include <errno.h>
#include <fstream>
using namespace std;
class optionsManager {
    map<string, string> optionDict;
    static optionsManager* m_optManager;
    vector<string> scanList;
    vector<string> portList;
    vector<string> ipList;
public:
    void readOptions(int argc, char* argv[]);
    static optionsManager* Instance();
    string GetStandardUsageOptionScreen();
    map<string, string> getOptionDictionary();

    void setPeerInfo(int numOfPeers, char* ptrToPeerString);

```

```

list<string> getpeerInfoList();

vector<string> split(string input, char delimiter);

vector<string> getScanList();

void unRollPortRange();

void calculateIPaddresesBitwise(const char* ipWithPrefix);

void printHostAddresses(unsigned long networkAddress, unsigned
long broadcastAddress);

void processIPFile(string fContent);

vector<string> getIPList();

vector<string> getPortList();

void deleteAllList();

void deleteSingleTon();
string ReadIPFile(const char* filename);
};

```

Appendix F: TCPHeader.h file

```

#pragma once

#ifndef TCPHEADER
#define TCPHEADER

// TCP header
typedef struct tcp_header
{
    unsigned short source_port; // source port
    unsigned short dest_port;  // destination port

```

```

unsigned int sequence;    // sequence number - 32 bits
unsigned int acknowledge; // acknowledgement number - 32 bits

unsigned char ns : 1;     //Nonce Sum Flag Added in RFC 3540.
unsigned char reserved_part1 : 3; //according to rfc
unsigned char data_offset : 4; /*The number of 32-bit word in the
                                TCP header.This indicates where
                                the data begins. The length of
                                the TCP header is always a
                                multiple of 32 bits.*/

unsigned char fin : 1; //Finish Flag
unsigned char syn : 1; //Synchronise Flag
unsigned char rst : 1; //Reset Flag
unsigned char psh : 1; //Push Flag
unsigned char ack : 1; //Acknowledgement Flag
unsigned char urg : 1; //Urgent Flag

unsigned char ecn : 1; //ECN-Echo Flag
unsigned char cwr : 1; //Congestion Window Reduced Flag

////////////////////////////////////

unsigned short window; // window
unsigned short checksum; // checksum
unsigned short urgent_pointer; // urgent pointer
} TCP_HDR, * PTCP_HDR, *LPTCP_HDR , TCPHeader, TCP_HEADER;

#endif // !TCPHEADER

```

Appendix G: TCPClass.h file

```

#pragma once
/*
 * TCPUtilities.h
 */

#ifndef TCPUTILITIES_H_
#define TCPUTILITIES_H_

```

```

#include <string>
#include <string.h>
#include <stdio.h>
#include <winsock2.h>
#include <ws2tcpip.h>
#include <mstcpip.h>
#include <iostream>
#include <sstream>
#include <process.h> // For _beginthreadex
#include <errno.h>
#include "CommonUtilities.h"
#include "work.h"

#define PACKET_LENGTH 2048

using namespace std;

class TCPUilities
{
    //"comUtil" Object of Class "CommonUtilities"
    CommonUtilities comUtil;

    HANDLE createPacketLock = CreateMutex(NULL, FALSE, NULL);
    // Windows equivalent for pthread_mutex_t

public:

    //Default Constructor
    TCPUilities();

    unsigned short csum(uint8_t* data, int length);

    //Checksum Calculator
    uint16_t calculateChecksum(uint32_t ipSource, uint32_t ipDest,
        uint8_t protocol, uint16_t tcpLength, struct tcp_header tcpSegment);

    //Packet Creation
    void createPacket(string scanType, const char* destIP,
        const char* portNumber, char*, char*);

```

```

//TCP Header Creator
void createTCPHeader(struct tcp_header* tcpHeader, int sourcePort,
const char* destPort, string scanType);

//Send TCP Packet
void sendTCPPacket(Job* job, char*);
};

#endif /* TCPUTILITIES_H_ */

```

Appendix H: UDPHeader.h file

```

//UDP Header

#pragma once
#include <cstdint>

struct udphdr {
    uint16_t source;
    uint16_t dest;
    uint16_t length;
    uint16_t checksum;
};

\end{verbatim}

\section{Appendix I: UDPClass.h file}

\small
\begin{verbatim}
#pragma once
/*
 * UDPUtilities.h
 */

#ifndef UDPUTILITIES_H_
#define UDPUTILITIES_H_

```



```

#include <iostream>
#include <string.h>
#include <winsock2.h>
#include <ws2tcpip.h>
#include <vector>
#include "CommonUtilities.h"
#include "work.h"

#define PACKET_LENGTH 2048

using namespace std;

class UDPUtilities
{
    //"comUtil" Object of Class "CommonUtilities"
    CommonUtilities comUtil;

public:
    //Creating UDP Header Content
    void createUDPHeader(struct udphdr* udpHeader, int sourcePort,
        const char* destPort);

    //Creating DNS Header Content
    void createDNSPacket(char* ipAddress, char* packet);

    void convertToDNSNameFormat(unsigned char* dnsHeader,
        char* destinationHost);

    //Fills in the UDP Packet
    int createPacketUDP(int sourcePort, const char* destPort,
        char* destIpAddress, char* packet);

    //Send the UDP Packet
    void sendUDPPacket(Job* job);
};

#endif /* UDPUTILITIES_H_ */

```

Appendix J: Work.h file

```

#pragma once

#ifndef JOB_H_
#define JOB_H_
#include <string>

using namespace std;

enum Status
{
    ASSIGNED,
    INPROGRESS,
    COMPLETED,
    NOTNOW
};

class Job
{
public:
    string scanType;
    string port;
    string IP;
    Status jobStatus;
    string conclusion;
    string serviceName;
    string serviceVersion;
    string scanResult;

    Job();
    Job(string, string, string);
    void* (*funcPointer)(void*);
    Job* args;
    void setJob(void* (*funcPointer)(void*));
    void execute();
    ~Job();
};

```

Appendix K: CommonUtilities.cpp file

```
#include "CommonUtilities.h"
#include "tcp_header.h"
#include "ip_header.h"
#include "icmp_header.h"
#include "udp_header.h"
#include "work.h"
#include <winsock2.h>
#include <ws2tcpip.h>
#include <windows.h>
#include <iphlpapi.h>
```

```
int CommonUtilities::sniffAPacket(const char* target, const char*
portNumber, string scanType, int protocol, Job* job, SOCKET
sockDescProt, SOCKET sockDescICMP) {
```

```
    int status = -1;
    u_long mode = 1;
```

```
    ioctlsocket(sockDescProt, FIONBIO, &mode);
    ioctlsocket(sockDescICMP, FIONBIO, &mode);
```

```
    struct pollfd fileDesc[2];
    struct sockaddr_in recievedIPStruct;
```

```
    memset(&recievedIPStruct, 0, sizeof(recievedIPStruct));
```

```
    fileDesc[0].fd = sockDescProt; fileDesc[0].events = POLLIN;
    fileDesc[1].fd = sockDescICMP; fileDesc[1].events = POLLIN;
```

```
    int pollStat = WSPoll(fileDesc, 2, 4000);
    int packetRecievedType = -1, recievedSize = -1; int supposedToBeRecievedPacket = -1;
```

```
    socklen_t size = sizeof(recievedIPStruct);
```

```
    const int MAX_RECIEVED_PACKET_LENGTH = 200;
    char sockReadBuffer[MAX_RECIEVED_PACKET_LENGTH];
```

```
    memset(sockReadBuffer, '\0', MAX_RECIEVED_PACKET_LENGTH);
```

```

time_t startTime = time(0);
double timeout = 4;

while (pollStat == 1) {

    time_t current = time(0);
    double timeElapsed = difftime(current, startTime);

    if (timeElapsed > timeout) {
        break;
    }
    if (fileDesc[0].revents & POLLIN) {
        recievedSize = recvfrom(sockDescProt, sockReadBuffer,
MAX_RECIEVED_PACKET_LENGTH, 0, (sockaddr*)&
recievedIPStruct, &size);
    }
    if (fileDesc[1].revents & POLLIN) {
        recievedSize = recvfrom(sockDescICMP, sockReadBuffer,
MAX_RECIEVED_PACKET_LENGTH, 0,
(sockaddr*)&recievedIPStruct, &size);
    }
    if (recievedSize > 0) {
        status = lookIntoThePacket(target, portNumber,
sockReadBuffer, scanType, job);
        if (status >= 0)
            break;
    }
}

return status;
}

```

SOCKET CommonUtilities::createRawSocket(int protocol)

```

{
    SOCKET sockfd = INVALID_SOCKET;
    while (sockfd == INVALID_SOCKET)
        sockfd = socket(AF_INET, SOCK_RAW, protocol);
    if (sockfd != INVALID_SOCKET) {
        BOOL optval = TRUE;
        setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, (char*)&optval, sizeof(optval));
    }
}

```

```

    }
    return sockfd;
}

```

```

int CommonUtilities::lookIntoThePacket(const char* ip,
const char* portNumber, char* sockReadBuffer,
string scanType, Job* job)
{
    int status = -1;

    struct iphdr* ptrToIPHeader = NULL;
    struct tcp_header* ptrToTCPHeader = NULL;
    struct sockaddr_in ipSource {};
    struct servent* ptrToServiceInfo = NULL;

    unsigned char* ptrToRecievedPacket = NULL;

    ptrToRecievedPacket = (unsigned char*)sockReadBuffer;
    ptrToIPHeader = (struct iphdr*)ptrToRecievedPacket;
    ptrToRecievedPacket += sizeof(iphdr);

    if (checkIfIPMatch(ip, ptrToIPHeader)) {

        if (ptrToIPHeader->protocol == IPPROTO_TCP)
            status = ParseTCPResponse(ip, portNumber, ptrToRecievedPacket, scanType,
job);
        else if (ptrToIPHeader->protocol == IPPROTO_UDP)
            status = parseUDPResponse(ip, portNumber, ptrToRecievedPacket, job);
        else if (ptrToIPHeader->protocol == IPPROTO_ICMP)
            status = parseICMPResponse(ip, portNumber, ptrToRecievedPacket, job);
    }
    else if (ptrToIPHeader->protocol == IPPROTO_ICMP)
        status = parseICMPResponse(ip, portNumber, ptrToRecievedPacket, job);

    return status;
}

```

```

bool CommonUtilities::checkIfIPMatch(const char* ip, struct iphdr* ptrToIPHeader)
{
    struct sockaddr_in ipSource;

```

```

memset(&ipSource, 0, sizeof(ipSource));
ipSource.sin_addr.s_addr = ptrToIPHeader->ip_srcaddr;

if (strcmp(ip, inet_ntoa(ipSource.sin_addr)) == 0) {
    return true;
}
return false;
}

```

```

int CommonUtilities::parseUDPResponse(const char* ip,
const char* portNumber, unsigned char* ptrToRecievedPacket,
Job* job) {
    int status = -1;
    struct udphdr* udpHeader = NULL;
    udpHeader = (struct udphdr*)ptrToRecievedPacket;
    if (atoi(portNumber) == ntohs(udpHeader->source)) {
        job->scanResult = "Open";
        status = 0;
    }
    return status;
}

```

```

int CommonUtilities::ParseTCPResponse(const char* ip, const char*
portNumber, unsigned char* ptrToRecievedPacket, string scanType,
Job* job)
{
    int status = -1;

    struct tcp_header* ptrToTCPHeader = NULL;
    struct servent* ptrToserviceInfo = NULL;

    ptrToTCPHeader = (struct tcp_header*)ptrToRecievedPacket;
    ptrToRecievedPacket += ptrToTCPHeader->data_offset * 4;

    if (atoi(portNumber) == ntohs(ptrToTCPHeader->source_port)) {

        if (scanType == "SYN") {

            if (ptrToTCPHeader->rst == 1) {
                job->scanResult = "Closed";
            }
        }
    }
}

```

```

        status = 1;
    }
    if (ptrToTCPHeader->syn == 1 && ptrToTCPHeader->ack == 1) {
        job->scanResult = "Open";
        status = 0;
    }
}
else if (scanType == "ACK") {
    if (ptrToTCPHeader->rst == 1) {
        job->scanResult = "Unfiltered";
        status = 1;
    }
}
else if (scanType == "NULL" || scanType == "XMAS" || scanType == "FIN") {
    if (ptrToTCPHeader->rst == 1) {
        job->scanResult = "Closed";
        status = 1;
    }
}
}
return status;
}

```

```

int CommonUtilities::parseICMPResponse(const char* ip,
const char* portNumber, unsigned char* ptrToPacketData,
Job* job)
{
    struct sockaddr_in ipDest;
    memset(&ipDest, 0, sizeof(ipDest));
    int status = -1;
    bool flag = true;
    struct icmphdr* icmpPtr = (struct icmphdr*)ptrToPacketData;
    ptrToPacketData += sizeof(struct icmphdr);
    struct iphdr* ipHeader = (struct iphdr*)ptrToPacketData;
    ptrToPacketData += sizeof(struct iphdr);
    ipDest.sin_addr.s_addr = ipHeader->daddr;
    if (strcmp(inet_ntoa(ipDest.sin_addr), ip) == 0)
    {
        if (ipHeader->protocol == IPPROTO_TCP)
        {

```

```

        struct tcp_header* tcpHeader = (struct tcp_header*)ptrToPacketData;
        if (atoi(portNumber) == ntohs(tcpHeader->dest_port))
            status = 1;
    }
    else if (ipHeader->protocol == IPPROTO_UDP)
    {
        struct udphdr* udpHeader = (struct udphdr*)ptrToPacketData;
        if (atoi(portNumber) == ntohs(udpHeader->dest)) {
            status = 1;
            flag = false;
        }
    }
    if (status == 1)
    {
        if (flag && icmpPtr->type == 3 && (icmpPtr->code == 1 || icmpPtr->code == 2 ||
icmpPtr->code == 3 || icmpPtr->code == 9 || icmpPtr->code == 10 || icmpPtr->code == 13))
            job->scanResult = "Filtered";
        else if (!flag && icmpPtr->type == 3 && (icmpPtr->code == 1 || icmpPtr->code
== 2 || icmpPtr->code == 9 || icmpPtr->code == 10 || icmpPtr->code == 13))
            job->scanResult = "Filtered";
        else if (!flag && icmpPtr->type == 3 && icmpPtr->code == 3)
            job->scanResult = "Closed";
    }
}
return status;
}

```

```

string CommonUtilities::probeHTTPVersion(sockaddr_in victim)
{
    int newSock;

    string getRequest;
    stringstream ss;

    int sentBytes, recievedSize = -1, versionLen;
    char sockReadBuffer[100];

    memset(sockReadBuffer, '\0', sizeof(sockReadBuffer));
    ss << "GET / HTTP/1.1 \r\nHost: " << inet_ntoa(victim.sin_addr)
<< "\r\nConnection: close\r\n\r\n";
}

```



```

getRequest = ss.str();
string stringedData;
size_t bytesToRead{}, bytesRead{};

struct timeval timeout;
fd_set fileDesc;

memset(&timeout, 0, sizeof(timeout));
newSock = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);

int status = connect(newSock, (struct sockaddr*)&victim, sizeof(victim));
if (status == -1)
{
    closesocket(newSock);
    return "No response";
}

timeout.tv_sec = 10; timeout.tv_usec = 0;

FD_ZERO(&fileDesc);
FD_SET(newSock, &fileDesc);

sentBytes = send(newSock, getRequest.c_str(), getRequest.length(), 0);
bytesToRead = sizeof(sockReadBuffer) - 1;

status = 0;

while (status < bytesToRead)
{
    status = select(newSock + 1, &fileDesc, NULL, NULL, &timeout);
    if (status > 0)
        status = recv(newSock, &sockReadBuffer[status], bytesToRead - status, 0);
}
closesocket(newSock);

stringedData = sockReadBuffer;
versionLen = stringedData.find("HTTP/1.1");

if (versionLen != string::npos)

```

```

{
    versionLen = stringedData.find("\r\n", versionLen);
    if (versionLen != string::npos)
        return stringedData.substr(0, versionLen);
}
return "No response";
}

```

```

string CommonUtilities::probeSMTPVersion(sockaddr_in victim) {

    char smtpRequest[10] = "EHLO\n";
    char sockReadBuffer[1000];

    int recievedSize = -1, sentBytes{}, versionLen = 1000;

    size_t pos{}, pos1{};

    memset(sockReadBuffer, '\0', sizeof(sockReadBuffer));

    int newSock;
    string stringedData;

    newSock = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);

    if (connect(newSock, (struct sockaddr*)&victim, sizeof(victim)) == 0) {
        recievedSize = recv(newSock, sockReadBuffer, 2048, 0);
        if (recievedSize < 0)
            return string("ERROR");
        else {
            stringedData = string(sockReadBuffer);
            if ((pos = stringedData.find("220")) != string::npos) {
                versionLen = stringedData.length() - pos;
                const int tup = 10000;
                char temp[tup];
                stringedData.copy(temp, versionLen, pos + strlen("220"));
                temp[versionLen] = '\0';
                stringedData = string(temp);
            }
        }
    }
}

```

```
        return stringedData;
    }
```

```
string CommonUtilities::getServiceInfo(struct sockaddr_in victim,
const char* port) {
```

```
    string versionInfo;
```

```
    switch (atoi(port)) {
```

```
        case 22: versionInfo = probeSSHVersion(victim); break;
```

```
        case 43: versionInfo = probeWHOISVersion(victim); break;
```

```
        case 80: versionInfo = probeHTTPVersion(victim); break;
```

```
        case 110: versionInfo = probePOPVersion(victim); break;
```

```
        case 143: versionInfo = probeIMAPVersion(victim); break;
```

```
        case 587: versionInfo = probeSMTPVersion(victim); break;
```

```
    }
```

```
    return versionInfo;
```

```
}
```

```
SOCKET CommonUtilities::bindRawSocket(int protocol,
struct sockaddr_in* victim, const char* ip)
```

```
{
```

```
    SOCKET sock = socket(AF_INET, SOCK_RAW, protocol);
```

```
    struct timeval timeout;
```

```
    timeout.tv_sec = 10; timeout.tv_usec = 0;
```

```
    memset(&timeout, 0, sizeof(timeout));
```

```
    if (sock == -1)
```

```
        return -1;
```

```
    if (protocol == IPPROTO_TCP)
```

```
        if (setsockopt(sock, IPPROTO_IP, IP_HDRINCL,
(const char*)&timeout, sizeof(timeout)) == -1)
```

```
            return -1;
```

```
    if (protocol == IPPROTO_ICMP)
```

```
        if (setsockopt(sock, SOL_SOCKET, SO_RCVTIMEO,
(const char*)&timeout, sizeof(timeout)) == -1)
```

```
            return -1;
```

```
    if (protocol == IPPROTO_UDP)
```

```

        if (setsockopt(sock, SOL_SOCKET, SO_SNDTIMEO,
(const char*)&timeout, sizeof(timeout)) == -1)
            return -1;
        memset(victim, 0, sizeof(struct sockaddr_in));
        victim->sin_family = AF_INET;
        victim->sin_addr.s_addr = inet_addr(ip);
        return sock;
}

```

```

void CommonUtilities::buildDestIPStruct(struct sockaddr_in* victim,
const char* ip, const char* portNumber) {

```

```

    victim->sin_family = AF_INET;
    victim->sin_port = htons(atoi(portNumber));
    victim->sin_addr.s_addr = inet_addr(ip);
}

```

```

string CommonUtilities::probeSSHVersion(sockaddr_in victim) {
    char sockReadBuffer[50];
    int recievedSize = -1;

    memset(sockReadBuffer, '\0', 50);

    int newSock;
    string stringedData;

    newSock = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);

    if (connect(newSock, (struct sockaddr*)&victim, sizeof(victim)) == 0) {
        recievedSize = recv(newSock, sockReadBuffer, 50, 0);
        if (recievedSize < 0)
            return string("ERROR");
        else {
            stringedData = string(sockReadBuffer);
        }
    }
    return stringedData;
}

```

```

string CommonUtilities::probeWHOISVersion(sockaddr_in victim) {

```

```

char sockReadBuffer[512];
memset(sockReadBuffer, '\0', 512);

int recievedSize = -1, sentBytes{}, versionLen{} ;
size_t pos{}, pos1{};

string stringedData;

int newSock;

memset(sockReadBuffer, '\0', sizeof(sockReadBuffer));
newSock = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);

if (connect(newSock, (struct sockaddr*)&victim, sizeof(victim)) == 0) {
    recievedSize = recv(newSock, sockReadBuffer, 511, 0);
    if (recievedSize < 0)
        return string("ERROR");
    else {
        stringedData = string(sockReadBuffer);

        if ((pos = stringedData.find("Version")) != string::npos) {
            versionLen = pos1 - (pos + strlen("Version "));
            char temp[7];
            memset(temp, '\0', 7);
            stringedData.copy(temp, 6, pos + strlen("Version "));
            stringedData = string(temp);
        }
    }
}
return stringedData;
}

string CommonUtilities::probePOPVersion(sockaddr_in victim) {
    char popRequest[10] = "ABCD";
    char sockReadBuffer[100];
    int recievedSize = -1, sentBytes = 0, versionLen;
    size_t pos, pos1;
    memset(sockReadBuffer, '\0', sizeof(sockReadBuffer));
    int newSock; string stringedData;

```

```

newSock = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
if (connect(newSock, (struct sockaddr*)&victim, sizeof(victim)) == 0) {

    sentBytes = send(newSock, popRequest, 22, 0);
    recievedSize = recv(newSock, sockReadBuffer, 100, 0);
    stringedData = string(sockReadBuffer);
    if (recievedSize < 0)
        return string("ERROR");
    else {
        if ((pos = stringedData.find("+OK")) != string::npos) {
            if ((pos1 = stringedData.find("ready")) != string::npos) {
                versionLen = pos1 - (pos + strlen("+OK "));
                const int tup = 10000;
                char temp[tup];
                stringedData.copy(temp, versionLen, pos + strlen("+OK "));
                temp[versionLen] = '\0';
                stringedData = string(temp);
            }
        }
    }
}
return stringedData;
}

```

```

string CommonUtilities::probeIMAPVersion(sockaddr_in victim) {

    char imapRequest[10] = "\r\n";
    char sockReadBuffer[2048];

    int recievedSize = -1, sentBytes{}, versionLen{} ;
    size_t pos, pos1;
    int newSock;

    string stringedData;
    memset(sockReadBuffer, '\0', sizeof(sockReadBuffer));

    newSock = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);

    if (connect(newSock, (struct sockaddr*)&victim, sizeof(victim)) == 0) {
        recievedSize = recv(newSock, sockReadBuffer, 2048, 0);
    }
}

```

```

        if (recievedSize < 0)
            return string("ERROR");
        else {
            stringedData = string(sockReadBuffer);
            if ((pos = stringedData.find("]")) != string::npos) {
                if ((pos1 = stringedData.find("ready")) != string::npos) {
                    versionLen = pos1 - (pos + strlen("] "));
                    const int tup = 10000;
                    char temp[tup];
                    stringedData.copy(temp, versionLen, pos + strlen("] "));
                    temp[versionLen] = '\0';
                    stringedData = string(temp);
                }
            }
        }
    }
    return stringedData;
}

```

Appendix L: NetProbe.cpp file

//Entry Point of the Main File

```

#include <stdio.h>
#include <errno.h>
#include <pthread.h>
#include <winsock2.h>
#include <ws2tcpip.h>
#include <iphlpapi.h>
#include <iostream>
#include <string.h>
#include <vector>
#include <map>
#include <math.h>
#include <windows.h>
#include "optionsClass.h"
#include "tcpClass.h"
#include "udpClass.h"
#include "work.h"

```

```

#include <iomanip>

#define PACKET_LENGTH 2048

#pragma comment(lib, "Ws2_32.lib")
#pragma comment(lib, "Iphlpapi.lib")

using namespace std;

vector<Job*> jobQueue;
map<string, bool> activeJobs;
typedef map<string, vector<Job*>> innerMap;
map<string, innerMap> reportMap;

pthread_mutex_t perJob = PTHREAD_MUTEX_INITIALIZER, perActiveJob =
PTHREAD_MUTEX_INITIALIZER;

pthread_mutex_t jobindex = PTHREAD_MUTEX_INITIALIZER;
int maxJobSize = 0; int jobsTaken = 0; size_t maxJobId = 0;

string getService(const char* protocol, const char* portNumber) {

    string serviceName = "NA";
    struct servent* serviceInfo;
    serviceInfo = getservbyport(htons(atoi(portNumber)), protocol);
    if (serviceInfo != NULL)
        serviceName = string(serviceInfo->s_name);
    return serviceName;
}

void getCurrentSystemIP(char* ip) {
    PIP_ADAPTER_INFO AdapterInfo;
    DWORD dwBufLen = sizeof(AdapterInfo);
    char* ipAddr = nullptr;
    AdapterInfo = (PIP_ADAPTER_INFO*)malloc(sizeof(IP_ADAPTER_INFO));
    if (AdapterInfo == NULL) {
        printf("Error allocating memory needed to call GetAdaptersinfo\n");
        return;
    }
}

```



```

    if (GetAdaptersInfo(AdapterInfo, &dwBufLen) == ERROR_BUFFER_OVERFLOW) {
        AdapterInfo = (IP_ADAPTER_INFO*)malloc(dwBufLen);
        if (AdapterInfo == NULL) {
            printf("Error allocating memory needed to call GetAdaptersinfo\n");
            return;
        }
    }

    if (GetAdaptersInfo(AdapterInfo, &dwBufLen) == NO_ERROR) {
        PIP_ADAPTER_INFO pAdapterInfo = AdapterInfo;
        while (pAdapterInfo) {
            if (pAdapterInfo->Type == MIB_IF_TYPE_ETHERNET || pAdapterInfo->Type ==
IF_TYPE_IEEE80211) {
                ipAddr = pAdapterInfo->IpAddressList.IpAddress.String;
                if (ipAddr && strcmp(ipAddr, "0.0.0.0") != 0) {
                    strcpy(ip, ipAddr);
                    printf("Current SYSTEM IP: %s\n", ip);
                    break;
                }
            }
            pAdapterInfo = pAdapterInfo->Next;
        }
    }
    if (AdapterInfo)
        free(AdapterInfo);
}

bool checkIfActiveJobWithSameIPandPort(Job* job) {

    if (!activeJobs.empty() && job != NULL) {
        if (activeJobs.find(job->IP + job->port) != activeJobs.end()) {
            return false;
        }
    }
    return true;
}

string conclude(int inputArray[5]) {
    int big = 0, iOfBig{};
    string conclusion;
    for (int i = 0; i < 4; i++) {
        if (inputArray[i] > big) {

```

```

        big = inputArray[i];
        iOfBig = i;
    }
}
switch (iOfBig) {
    cout << "inside switch";
    case 0: conclusion = "Filtered"; break;
    case 1: conclusion = "Open|Filtered"; break;
    case 2: conclusion = "Unfiltered"; break;
    case 3: conclusion = "Closed"; break;
    case 4: conclusion = "Open"; break;
}
return conclusion;
}

void printJobStats() {

    std::cout << "Entering printJobStats() Function" << "\n";

    map<string, innerMap>::iterator reportMapItr;
    map<string, vector<Job*>>::iterator innerMapItr;
    vector<Job*> jobList, openList, closedList;
    vector<Job*>::iterator jobListItr, try1, openListItr, closedListItr;
    reportMapItr = reportMap.begin();
    innerMap templm; string tempScanList;
    int openfiltered = 0, unfiltered = 0, filtered = 0, open = 0, closed = 0;
    int portConclusion[5]; string protocolType, serviceName;
    memset(&portConclusion, 0, sizeof(portConclusion));
    vector<string> tempScanResult;
    cout << "----- Scanned Results Stats-----
-----" << endl;
    while (reportMapItr != reportMap.end()) {
        cout << "" << endl;
        cout << "IP Address: " << reportMapItr->first << endl;
        templm = reportMapItr->second;
        innerMapItr = templm.begin();
        openList.clear(); closedList.clear();
        while (innerMapItr != templm.end()) {
            tempScanList.clear();
            jobList = innerMapItr->second;

```

```

try1 = jobListIter = jobList.begin();
string Conclusion = "Unknown";
while (jobListIter != jobList.end()) {
    tempScanList.append((*jobListIter)->scanType);
    tempScanList.append("(");
    tempScanList.append((*jobListIter)->scanResult);
    tempScanList.append(") ");
    if (((*jobListIter)->scanType == "SYN" && (*jobListIter)->scanResult ==
"Open") || (((*jobListIter)->scanType == "UDP") && (*jobListIter)->scanResult == "Open"))
        Conclusion = "Open";
    else if ((*jobListIter)->scanType == "SYN" && (*jobListIter)->scanResult
== "Closed")
        Conclusion = "Closed";
    else if ((*jobListIter)->scanResult == "Filtered")
        portConclusion[0] = ++filtered;
    else if ((*jobListIter)->scanResult == "Open|Filtered")
        portConclusion[1] = ++openfiltered;
    else if ((*jobListIter)->scanResult == "Unfiltered")
        portConclusion[2] = ++unfiltered;
    else if ((*jobListIter)->scanResult == "Closed")
        portConclusion[3] = ++closed;
    else if ((*jobListIter)->scanResult == "Open")
        portConclusion[4] = ++open;
    jobListIter++;
}
(*try1)->scanResult = tempScanList;
if (Conclusion == "Unknown")
    Conclusion = conclude(portConclusion);
(*try1)->conclusion = Conclusion;
if ((*try1)->conclusion == "Open")
    openList.push_back(*try1);
else
    closedList.push_back(*try1);
memset(&portConclusion, 0, sizeof(portConclusion));
openfiltered = 0; unfiltered = 0; filtered = 0; open = 0; closed = 0;
innerMapItr++;
}
cout << endl << endl;
cout << "Open Ports: " << endl;
cout << left << setw(7) << "Port" << left << setw(15)

```

```

<<"Service Name" << left << setw(50) << "Results" << left
<< setw(25) << "Version" << setw(10) << "Conclusion" << endl;
    cout << "-----"
-----" << endl;
    if (openList.size() > 0) {
        openListIter = openList.begin();
        while (openListIter != openList.end()) {
            if ((*openListIter)->scanType == "UDP")
                protocolType = "udp";
            else
                protocolType = "tcp";
            serviceName = getService(protocolType.c_str(), ((*openListIter)-
>port).c_str());

            cout << left << setw(7) << (*openListIter)->port
            << left << setw(15) << serviceName << left << setw(50) << (*openListIter)->scanResult << left <<
            setw(25) << (*openListIter)->serviceVersion << setw(10) << (*openListIter)->conclusion << endl;
            openListIter++;
        }
    }
    cout << endl << endl;
    cout << "Closed/Filtered/Unfiltered Ports: " << endl;
    cout << left << setw(7) << "Port" << left << setw(15)
    s<<"Service Name" << left << setw(50) << "Results" << left <<
    setw(25) << "Version" << setw(10) << "Conclusion" << endl;
    cout << "-----"
-----" << endl;
    if (closedList.size() > 0) {
        closedListIter = closedList.begin();
        while (closedListIter != closedList.end()) {
            if ((*closedListIter)->scanType == "UDP")
                protocolType = "udp";
            else
                protocolType = "tcp";
            serviceName = getService(protocolType.c_str(), ((*closedListIter)-
>port).c_str());

            cout << left << setw(7) << (*closedListIter)->port << left << setw(15) <<
            serviceName << left << setw(50) << (*closedListIter)->scanResult << left << setw(25) << (*closedListIter)-
>serviceVersion << setw(10) << (*closedListIter)->conclusion << endl;
            closedListIter++;
        }
    }
}

```

```

        reportMapItr++;
    }
}

void reportCompletedJob(Job* job) {
    innerMap portMap;
    map<string, vector<Job*>>::iterator innerMapItr;
    vector<Job*> tempJobs;
    auto ipvalue = reportMap.find(job->IP);
    if (ipvalue != reportMap.end()) {
        portMap = ipvalue->second;
        auto portvalue = portMap.find(job->port);
        if (portvalue != portMap.end()) {
            tempJobs = portvalue->second;
            tempJobs.push_back(job);
            portMap.erase(portvalue);
            portMap.insert(pair<string, vector<Job*>>{job->port, tempJobs});
        }
        else {
            tempJobs.push_back(job);
            portMap.insert(pair<string, vector<Job*>>{job->port, tempJobs});
        }
        reportMap.erase(ipvalue);
        reportMap.insert(pair<string, innerMap>{job->IP, portMap});
    }
    else
    {
        tempJobs.push_back(job);
        portMap.insert(pair<string, vector<Job*>>{job->port, tempJobs});
        reportMap.insert(pair<string, innerMap>{job->IP, portMap});
    }
}

```

```

void* sendPacket(void* message) {

```

```

    TCPUilities tcpUtil;
    UDPUtilities udpUtil;
    Job* job;
    int returnValue{};
    char* ip = (char*)message;

```

```

while (true) {

    pthread_mutex_lock(&jobindex);
    if (maxJobId < jobQueue.size()) {
        job = jobQueue.at(maxJobId);
        maxJobId++;
        if (!checkIfActiveJobWithSameIPandPort(job)) {
            --maxJobId;
            job->jobStatus = NOTNOW;
        }
        else {
            job->jobStatus = ASSIGNED;
            activeJobs.insert(make_pair(job->IP + job->port, true));
        }
    }
    else {
        pthread_mutex_unlock(&jobindex);
        break;
    }
    pthread_mutex_unlock(&jobindex);
    if (job->jobStatus != NOTNOW) {

        if (job->scanType.compare("UDP") == 0)
            udpUtil.sendUDPPacket(job);
        else
            tcpUtil.sendTCPPacket(job, ip);

        pthread_mutex_lock(&perActiveJob);
        if (job->jobStatus == COMPLETED) {
            auto value = activeJobs.find(job->IP + job->port);
            if (value->second) {
                activeJobs.erase(value->first);
                reportCompletedJob(job);
            }
        }
        pthread_mutex_unlock(&perActiveJob);
    }

}

return NULL;

```

```
}
```

```
pthread_t createThreads(int threadCount)
```

```
{
```

```
    vector<pthread_t> threads(threadCount);
```

```
    int createStatus;
```

```
    pthread_t thread;
```

```
    for (int i = 0; i < threadCount; i++) {
```

```
        createStatus = pthread_create(&threads[i], NULL, sendPacket, (void*)NULL);
```

```
        if (createStatus != 0) {
```

```
            cout << "Create thread failed" << endl;
```

```
        }
```

```
        else {
```

```
            cout << "Thread " << i << " created successfully." << endl;
```

```
        }
```

```
        thread = threads[i];
```

```
    }
```

```
    return thread;
```

```
}
```

```
void destroyJobQueue() {
```

```
    for (vector<Job*>::iterator jobIter = jobQueue.begin(); jobIter != jobQueue.end(); ++jobIter)
```

```
        delete* jobIter;
```

```
}
```

```
void createJobQueue() {
```

```
    std::cout << "Entered createJobQueue() Function" << '\n';
```

```
    vector <string> ipList = optionsManager::Instance()->getIPList();
```

```
    vector <string> scanList = optionsManager::Instance()->getScanList();
```

```
    vector <string> portList = optionsManager::Instance()->getPortList();
```

```
    for (vector<string>::iterator sc = scanList.begin(); sc != scanList.end(); ++sc) {
```

```
        for (vector<string>::iterator iplter = ipList.begin();
```

```
        iplter != ipList.end(); ++iplter) {
```

```
            for (vector<string>::iterator portIter =
```

```

portList.begin(); portIter != portList.end(); ++portIter) {
    jobQueue.push_back(new Job(*ipIter, *portIter,
*sc));
    }
}

cout << "Jobs created: " << jobQueue.size() << endl;

optionsManager::Instance()->deleteAllList();
}

int processCommand(map<string, string> opDict) {

    int returnVal = 0;
    string ip;
    string targetPort;
    auto value = opDict.find("help");
    if (value != opDict.end()) {
        cout << endl;
        cout << value->second;
        return 0;
    }

    value = opDict.find("ipaddressfile");
    if (value != opDict.end()) {
        string ipAddressFile = value->second;
        cout << "IP File: " << ipAddressFile << endl;
        optionsManager::Instance()->processIPFile(ipAddressFile);
    }

    value = opDict.find("prefix");
    if (value != opDict.end())
        optionsManager::Instance()->calculateIPAddresesBitwise(value->second.c_str());

    return returnVal;
}

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

```



```

WSADATA wsaData;
int iResult = WSASStartup(MAKEWORD(2, 2), &wsaData);
if (iResult != 0) {
    cout << "WSASStartup failed: " << iResult << endl;
    return 1;
}

time_t start, end = 0, elapsed = 0;
cout << "Scanning....." << endl;
if (argc < 2)
    cout << " For Usage type : ./portScanner -h" << endl;
else {
    optionsManager::Instance()->readOptions(argc, argv);

    map<string, string> opDict = optionsManager::Instance()->getOptionDictionary();

    auto value = opDict.find("help");
    if (value != opDict.end()) {
        cout << endl;
        cout << value->second;
        return 0;
    }
    else {

        start = time(NULL);

        int numberOfThreads = 1;
        value = opDict.find("speedup");
        if (value != opDict.end())
            numberOfThreads = stoi(value->second);

        //Processing Command
        processCommand(opDict);

        //Creating Job
        createJobQueue();

        vector<pthread_t> threads;

```

```

int createStatus;
char ip[INET_ADDRSTRLEN];

//Getting System IP
getCurrentSystemIP(ip);
pthread_t thread;

for (int i = 0; i < numberOfThreads; i++) {
    createStatus = pthread_create(&thread, NULL, sendPacket, (void*)ip);
    if (createStatus != 0) {
        cout << "Create thread failed" << endl; //return;
    }
    threads.push_back(thread);
}

for (int i = 0; i < numberOfThreads; i++) {
    pthread_join(threads[i], NULL);
}

}
end = time(NULL);
elapsed = end - start;
cout << "Scanning took: " << elapsed << " seconds" << endl;

printJobStats();

destroyJobQueue();
optionsManager::Instance()->deleteSingleTon();
}

WSACleanup();
}

```

Appendix M: OptionsClass.cpp file

```

#include "optionsClass.h"
#include <iostream>
#include <sstream>
#include <fstream>
#include <vector>
#include <map>
#include <cstring>
#include <cstdlib>
#include <winsock2.h>
#include <WS2tcpip.h>
#include "getopt.h"

using namespace std;

optionsManager* optionsManager::m_optManager = NULL;

void optionsManager::readOptions(int argc, char* argv[])
{
    int getOptChar = 0;
    int option_index = 0;
    const char* shortOptions = "hp:i:r:f:s:u:";

    struct option longOptions[] =
    {
        {"help",      no_argument,   NULL, 'h'},
        {"ports",     required_argument, NULL, 'p'},
        {"ip",        required_argument, NULL, 'i'},
        {"prefix",    required_argument, NULL, 'x'},
        {"file",      required_argument, NULL, 'f'},
        {"scan",      required_argument, NULL, 's'},
        {"speedup",   required_argument, NULL, 'u'},
        {NULL,        0,              NULL, 0 }
    };

    while ((getOptChar = getopt_long(argc, argv, shortOptions,
    longOptions, &option_index)) != -1)
    {
        switch (getOptChar)
        {

```

```

case 'h':
    optionDict.insert(pair<string, string>("help", GetStandardUsageOptionScreen()));
    break;
case 'p':
    optionDict.insert(pair<string, string>("ports", optarg));
    portList = split(optarg, ',');
    break;
case 'i':
    optionDict.insert(pair<string, string>("ip", optarg));
    ipList.push_back(string(optarg));
    break;
case 'x':
    optionDict.insert(pair<string, string>("prefix", optarg));
    break;
case 'f':
    optionDict.insert(pair<string, string>("ipaddressfile", optarg));
    break;
case 's':
    optionDict.insert(pair<string, string>("scan", optarg));
    if (strcmp(optarg, "SYN") == 0 || strcmp(optarg, "NULL") == 0
        || strcmp(optarg, "ACK") == 0 || strcmp(optarg, "UDP") == 0
        || strcmp(optarg, "XMAS") == 0 || strcmp(optarg, "FIN") == 0) {
        scanList.push_back(optarg);
    }
    else {
        cout << "INVALID SCAN " << endl;
        exit(0);
    }
    break;
case 'u':
    optionDict.insert(pair<string, string>("speedup", optarg));
    break;
default:
    fprintf(stderr, "ERROR: Unknown option '-%c'\n", getoptChar);
    exit(1);
}
}

if (portList.size() == 0)
{

```

```

    portList = split("1-1024", ',');
}

if (optind < argc)
{
    while (optind < argc)
    {
        if (strcmp(argv[optind], "SYN") == 0 || strcmp(argv[optind], "NULL") == 0
            || strcmp(argv[optind], "ACK") == 0 || strcmp(argv[optind], "UDP") == 0
            || strcmp(argv[optind], "XMAS") == 0 || strcmp(argv[optind], "FIN") == 0) {
            scanList.push_back(argv[optind++]);
        }
        else
        {
            optind++;
        }
    }
}

unRollPortRange();
}

optionsManager* optionsManager::Instance()
{
    if (!m_optManager)
        m_optManager = new optionsManager();
    return m_optManager;
}

vector<string> optionsManager::split(string input, char delimiter)
{
    stringstream ss(input);
    vector<string> outputList;
    string temp;

    while (getline(ss, temp, delimiter))
    {
        outputList.push_back(temp);
    }
}

```

```

    return outputList;
}

```

```

void optionsManager::unRollPortRange()
{
    vector<string> tempList;
    for (auto& port : portList)
    {
        size_t pos = port.find('-');
        if (pos != string::npos)
        {
            int start = stoi(port.substr(0, pos));
            int end = stoi(port.substr(pos + 1));

            for (int i = start; i <= end; ++i)
            {
                tempList.push_back(to_string(i));
            }
        }
        else
        {
            tempList.push_back(port);
        }
    }
    portList.swap(tempList);
}

```

```

string optionsManager::GetStandardUsageOptionScreen()
{
    return ". /portScanner [option1, ..., optionN] \n \
        --help. Example: ". /portScanner --help". \n \
        --ports <ports to scan>. Example: ". /portScanner --ports 1,2,3-5". \n \
        --ip <IP address to scan>. Example: ". /portScanner --ip 127.0.0.1". \n \
        --prefix <IP prefix to scan>. Example: ". /portScanner --prefix 127.143.151.123/24". \n \
        --file <file name containing IP addresses to scan>. Example: ". /portScanner --file filename.txt". \n \
        --speedup <parallel threads to use>. Example: ". /portScanner --speedup 10". \n \
        --scan <one or more scans>. Example: ". /portScanner --scan SYN NULL FIN XMAS". \n";
}

```

```

map<string, string> optionsManager::getOptionDictionary()

```

```
{  
    return optionDict;  
}
```

```
vector<string> optionsManager::getScanList()  
{  
    return scanList;  
}
```

```
vector<string> optionsManager::getIPList()  
{  
    return ipList;  
}
```

```
vector<string> optionsManager::getPortList()  
{  
    return portList;  
}
```

```
void optionsManager::deleteAllList()  
{  
    ipList.clear();  
    portList.clear();  
    scanList.clear();  
    optionDict.clear();  
}
```

```
void optionsManager::deleteSingleTon()  
{  
    delete m_optManager;  
}
```

```
void optionsManager::printHostAddresses(unsigned long  
networkAddress, unsigned long broadcastAddress)  
{  
    struct in_addr address;  
  
    for (unsigned long i = ntohl(networkAddress) + 1; i < ntohl(broadcastAddress); ++i)  
    {  
        address.s_addr = htonl(i);
```

```

        ipList.push_back(string(inet_ntoa(address)));
    }
}

void optionsManager::calculateIPAddresesBitwise(const char* ipWithPrefix)
{
    struct in_addr ipaddress;
    struct in_addr ipMask;

    char* inputIP;

    int prefix;
    unsigned long networkID, hostBits, broadcastID;

    char* pch = strtok((char*)ipWithPrefix, "/");
    inputIP = pch;
    pch = strtok(NULL, "/");
    sscanf(pch, "%d", &prefix);

    inet_pton(AF_INET, inputIP, &ipaddress);
    unsigned long subnetMask = 0;
    for (int i = 0; i < prefix; ++i)
    {
        subnetMask |= 1 << (31 - i);
    }

    ipMask.s_addr = htonl(subnetMask);

    networkID = ntohl(ipaddress.s_addr) & ntohl(ipMask.s_addr);
    ipaddress.s_addr = htonl(networkID);

    ipList.push_back(inet_ntoa(ipaddress));
    hostBits = ~ntohl(ipMask.s_addr);

    broadcastID = networkID | hostBits;
    ipaddress.s_addr = htonl(broadcastID);

    ipList.push_back(inet_ntoa(ipaddress));
    printHostAddresses(networkID, broadcastID);
}

```



```

void optionsManager::processIPFile(string fileName)
{
    string fileContent = ReadIPFile(fileName.c_str());
    if (!fileContent.empty())
    {
        istringstream iss(fileContent);
        string line;
        while (getline(iss, line))
        {
            ipList.push_back(line);
        }
    }
}

```

```

string optionsManager::ReadIPFile(const char* filename)
{
    ifstream file(filename);
    stringstream buffer;
    buffer << file.rdbuf();
    return buffer.str();
}

```

Appendix N: TcpClass.cpp file

```

#include "tcpClass.h"
#include <iostream>
#include <string>
#include <cstring>
#include <ctime>
#include <winsock2.h>
#include <ws2tcpip.h>
#include "CommonUtilities.h"
#include "work.h"
#include "tcp_header.h"

#define PACKET_LENGTH 2048

using namespace std;

```

```
TCPUtilities::TCPUtilities() {}
```

```
unsigned short TCPUtilities::csum(uint8_t* data, int length)
{
    long checkSum = 0;

    while (length > 0)
    {
        checkSum += (*data << 8 & 0xFF00) + (*(data + 1) & 0xFF);
        data += 2;
        length -= 2;
    }
    if (checkSum >> 16)
        checkSum = ((checkSum >> 16) & 0x00ff) + (checkSum & 0xFFFF);

    uint16_t finalSum = (uint16_t)(~checkSum);

    return finalSum;
}
```

```
uint16_t TCPUtilities::calculateChecksum(uint32_t ipSource,
uint32_t ipDest, uint8_t protocol, uint16_t tcpLength,
struct tcp_header tcpSegment)
{
    char packet[PACKET_LENGTH];
    int checksumLength = 0;

    memcpy(packet, &ipSource, sizeof(ipSource));
    checksumLength += sizeof(ipSource);

    memcpy(packet + checksumLength, &ipDest, sizeof(ipDest));
    checksumLength += sizeof(ipDest);

    packet[checksumLength] = 0;
    checksumLength += 1;

    memcpy(packet + checksumLength, &protocol, sizeof(protocol));
    checksumLength += sizeof(protocol);

    memcpy(packet + checksumLength, &tcpLength, sizeof(tcpLength));
```

```

    checksumLength += sizeof(tcpLength);

    char* tcpheader = (char*)&tcpSegment;
    memcpy(packet + checksumLength, tcpheader, 20);
    checksumLength += 20;

    return csum((uint8_t*)packet, checksumLength);
}

void TCPUilities::createPacket(string scanType, const char* destIP,
const char* portNumber, char* packetData, char* srcIP)
{
    struct tcp_header* tcp = (struct tcp_header*)packetData;
    memset(tcp, 0, sizeof(struct tcp_header));

    int min = 30000, max = 60000;
    srand(static_cast<unsigned int>(time(nullptr)));
    int sourcePort = min + rand() % (max - min + 1);

    createTCPHeader(tcp, sourcePort, portNumber, scanType);
    tcp->checksum = htons(calculateChecksum(inet_addr(srcIP), inet_addr(destIP), IPPROTO_TCP,
    htons(sizeof(struct tcp_header)), *tcp));
}

void TCPUilities::createTCPHeader(struct tcp_header* tcpHeader,
int sourcePort, const char* destPort, string scanType) {
    tcpHeader->source_port = htons(static_cast<uint16_t>(sourcePort));
    tcpHeader->dest_port = htons(static_cast<uint16_t>(atoi(destPort)));
    tcpHeader->syn = 0;
    tcpHeader->sequence = 0;
    tcpHeader->ack = 0;
    tcpHeader->window = htons(1024);
    tcpHeader->checksum = 0;
    tcpHeader->rst = 0;
    tcpHeader->urgent_pointer = 0;
    tcpHeader->data_offset = 5;

    if (scanType == "SYN") {
        tcpHeader->syn = 1;
        tcpHeader->sequence = htonl(1);
    }
}

```

```

    }
    else if (scanType == "XMAS") {
        tcpHeader->psh = 1;
        tcpHeader->urg = 1;
    }
    else if (scanType == "FIN") {
        tcpHeader->fin = 1;
    }
    else if (scanType == "ACK") {
        tcpHeader->ack = 1;
    }
}

```

```

void TCPUilities::sendTCPPacket(Job* job, char* srcIP)
{
    const char* ip = job->IP.c_str();
    const char* portNumber = job->port.c_str();
    string scanType = job->scanType;
    int probeCounter = 3;
    struct sockaddr_in victim, victim_copy;
    memset(&victim, 0, sizeof(struct sockaddr_in));
    comUtil.buildDestIPStruct(&victim, ip, portNumber);
    memcpy(&victim_copy, &victim, sizeof(victim));
    char packData[PACKET_LENGTH];
    createPacket(scanType, ip, portNumber, packData, srcIP);
}

```

```

WSADATA wsaData;
int wsResult = WSASStartup(MAKEWORD(2, 2), &wsaData);
if (wsResult != 0)
{
    cerr << "WSASStartup failed with error: " << wsResult << endl;
    return;
}

```

```

SOCKET sockDesc = socket(AF_INET, SOCK_RAW, IPPROTO_TCP);
if (sockDesc == INVALID_SOCKET)
{
    cerr << "Socket creation failed with error: "
    << WSAGetLastError() << endl;
    WSACleanup();
}

```

```

        return;
    }

    int status = -1;
    while (status < 0 && probeCounter > 0)
    {
        if (sendto(sockDesc, packData, sizeof(struct tcp_header), 0, (sockaddr*)&victim, sizeof(struct
sockaddr_in)) > 0)
        {
            status = comUtil.sniffAPacket(ip, portNumber, scanType, IPPROTO_TCP, job, sockDesc, sockDesc);
        }
        probeCounter--;
    }

    closesocket(sockDesc);
    WSACleanup();

    if (status == 0)
    {
        static HANDLE createPacketLock = CreateMutex(NULL, FALSE,
NULL);

        WaitForSingleObject(createPacketLock, INFINITE);

        job->serviceVersion = comUtil.getServiceInfo(victim_copy, portNumber);

        ReleaseMutex(createPacketLock);
    }

    job->jobStatus = COMPLETED;
}

```

Appendix O: UcpClass.cpp file

```

#include "udpClass.h"
#include "DNS_Header.h"
#include "udp_header.h"
#include "work.h"

void UDPUilities::createUDPHeader(struct udphdr* udpHeader,

```

```
int sourcePort, const char* destPort)
```

```
{  
    udpHeader->source = htons(sourcePort);  
    udpHeader->dest = htons(atoi(destPort));  
    udpHeader->length = htons(sizeof(struct udphdr));  
    udpHeader->checksum = 0;  
}
```

```
void UDPUilities::createDNSPacket(char* ipAddress, char* packet)
```

```
{  
    DNS_HEADER* dnsHeader = (DNS_HEADER*)packet;  
  
    dnsHeader->id = htons(rand());  
    dnsHeader->qr = 0;  
    dnsHeader->opcode = 0;  
    dnsHeader->aa = 0;  
    dnsHeader->tc = 0;  
    dnsHeader->rd = 1;  
    dnsHeader->ra = 0;  
    dnsHeader->z = 0;  
    dnsHeader->ad = 0;  
    dnsHeader->cd = 0;  
    dnsHeader->rcode = 0;  
    dnsHeader->q_count = htons(1);  
    dnsHeader->ans_count = 0;  
    dnsHeader->auth_count = 0;  
    dnsHeader->add_count = 0;  
}
```

```
void UDPUilities::convertToDNSNameFormat(unsigned char* dnsHeader,  
char* destinationHost)
```

```
{  
    unsigned char* rvIterator = dnsHeader;  
    int count = 0;  
  
    while (*destinationHost)  
    {  
        if (*destinationHost == '.')  
        {  
            *rvIterator++ = count;  

```

```

        count = 0;
    }

    else
    {
        *rvIterator++ = *destinationHost;
        count++;
    }
    destinationHost++;
}
*rvIterator++ = count;
*rvIterator = '\0';
}

int UDPUilities::createPacketUDP(int sourcePort, const char*
destPort, char* destIpAddress, char* packet)
{
    struct udphdr* udpPack = (struct udphdr*)packet;

    size_t totalSize = sizeof(struct udphdr);

    createUDPHeader(udpPack, sourcePort, destPort);
    if (strcmp(destPort, "53") == 0)
    {
        createDNSPacket(destIpAddress, packet + sizeof
(struct udphdr));
    }
    return totalSize;
}

void UDPUilities::sendUDPPacket(Job* job)
{
    const char* destPort = job->port.c_str();
    const char* destIpAddress = job->IP.c_str();
    string scanType = job->scanType;

    WSADATA wsaData;
    if (WSAStartup(MAKEWORD(2, 2), &wsaData) != 0)
    {
        cout << "Failed to initialize Winsock.\n";
    }
}

```

```

        return;
    }

    SOCKET sockDesc = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
    if (sockDesc == INVALID_SOCKET) {
        cout << "Socket creation failed.\n";
        WSACleanup();
        return;
    }

    char packData[PACKET_LENGTH];

    memset(packData, 0, PACKET_LENGTH);
    size_t totalSize = sizeof(struct udphdr);

    int min = 30000, max = 60000;
    srand((unsigned int)time(NULL));

    int sourcePort = min + rand() % (max - min + 1);
    totalSize = createPacketUDP(sourcePort, destPort,
    (char*)destIpAddress, packData);

    struct sockaddr_in destAddr;
    destAddr.sin_family = AF_INET;
    destAddr.sin_port = htons(atoi(destPort));
    destAddr.sin_addr.s_addr = inet_addr(destIpAddress);

    int bytesSent = sendto(sockDesc, packData, totalSize, 0,
    (struct sockaddr*)&destAddr, sizeof(destAddr));
    if (bytesSent == SOCKET_ERROR) {
        cout << "Send failed with error: " << WSAGetLastError()
        << "\n";
    }

    closesocket(sockDesc);
    WSACleanup();

    job->jobStatus = COMPLETED;
}

```


Appendix P: Work.cpp file

```
#include "work.h"

void Job::setJob(void* (*fptr)(void*))
{
    funcPointer = fptr;
}

void Job::execute()
{
    (*funcPointer)(this);
}

Job::Job(string ipAddress, string portNum, string scan)
{
    IP = ipAddress;
    port = portNum;
    scanType = scan;
    jobStatus = NOTNOW;
    serviceName = "NA";
    serviceVersion = "NA";
    conclusion = "NOTAVAILABLE";
}

Job::Job() {}

Job::~Job() {};
```