**TECHNICKÁ UNIVERZITA V KOŠICIACH**
**FAKULTA ELEKTROTECHNIKY A INFORMATIKY**

# POROVNANIE PRIEPUSTNOSTI A ODOZVY SIEŤOVÝCH KOMUNIKAČNÝCH PROTOKOLOV
## Príloha C – Systémová príručka

| | |
|---|---|
| Študijný program: | Inteligentné systémy |
| Študijný odbor: | Informatika |
| Školiace pracovisko: | Katedra kybernetiky a umelej inteligencie (KKUI) |
| Školiteľ: | doc. Ing. Ján Jadlovský, CSc. |
| Konzultant: | Ing. Milan Tkáčik |

**2022 Košice** **Matush Pavlo**

# Obsah

# Úvod

Predmetom tejto systémovej príručky je oboznámiť programátora s popisom jednotlivých funkcií a návodom na vybudovanie aplikácií. Konkrétne ide o tieto tri aplikácie: Klientsky program merania priepustnosti, Klientsky program merania odozvy a serverový program pre nich. V tomto dodatku bude popísaná každá použitá funkcia, príkaz, trieda, lokálna premenná a objekt.

# 1 Použité knižnice

Požadované knižnice:

- **<WinSock2.h>** - Microsoft knižnica, prostredníctvom ktorej sa vytvára spojenie a prebieha komunikácia (pre TCP a UDP)
- **<ServerLib.h>** - moja vlastná knižnica pre serverový program (pre TCP a UDP)
- **<ClientLib.h>** - moja vlastná knižnica pre klientsky program (pre TCP a UDP)
- **<chrono>** - vstavaná knižnica, používaná na meranie aktuálneho systémového času
- **<memory.h>** - používa sa pri práci s plnením alebo uvoľňovaním pamäte
- **<ctime>** - používaná na meranie aktuálneho systémového času
- **<stdio.h>** - aby pracovať s konzolou, zapisovať a čítať dáta z bufferov
- **<iostream>** - vstavaná knižnica, používa sa na prácu s konzolou (pre DIM)
- **<string>** - vstavaná knižnica, pre pracu s dátovým typom “string”
- **<dic.hxx>** - oficiálna knižnica protokolu DIM používaná na pripojenie a komunikáciu pre klienta (pre DIM)
- **<dis.hxx>** - oficiálna knižnica protokolu DIM používaná na pripojenie a komunikáciu pre server (pre DIM)

# 2 Funkcie, triedy, premenné a iné

## 2.1. Potrebné funkcie, premenné a triedy knižnice <WinSock2.h>:

- Trieda **SOCKET** - trieda, ktorá umožňuje vytvárať sokety pre klienta a pre server, pomocou ktorých prebieha spojenie a komunikácia
- Funkcia **SOCKET socket(int af, int type, int protocol)** - funkcia na vytvorenie soketu s parametrami *int af*, *int type* a *int protocol*. Takto to vyzerá pre protokol TCP - socket(AF_INET, SOCK_STREAM, IPPROTO_TCP) a pre protokol UDP - socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP). Vráti sa vytvorený soket
- Definované **AF_INET** - typ komunikácie (používa sa v protokoloch TCP, UDP a iných)
- Definované **SOCK_STREAM** - výber typu komunikácie medzi soketmi (v tomto prípade typ – “socket stream”)

- Definované **SOCK_DGRAM** - výber typu komunikácie medzi soketmi (v tomto prípade typ – “datagram socket”)
- Enum **IPPROTO_TCP** - premenná typu “enum” na výber protokolu TCP
- Enum **IPPROTO_UDP** - premenná typu “enum” na výber protokolu UDP
- Funkcia **int closesocket(SOCKET s)** - zatvorí soket, vráti sa 0 alebo 1
- Funkcia **`int` WSACleanup()** - vymaže pamäť a komunikačné tunely vytvorené programom (upratuje po sebe). Vráti sa 0 alebo 1
- Rozhranie **WSADATA** - vytvárajúce rozhranie Winsock API
- Funkcia **int WSAStartup(WORD wVersionRequested, LPWSADATA lpWSAData)** - inicializácia Winsock a výber verzie. Takto to vyzerá pre protokol TCP - WSAStartup(MAKEWORD(2, 2), &wsaData) (verzia Winsock 2.2). Vráti sa 0 alebo 1
- Definované **MAKEWORD(2, 2)** - generuje premennú typu “WORD” potrebnú na určenie verzie Winsock (v tomto prípade ide o verziu 2.2)
- Struct **sockaddr_in** - premenná typu „struct“, ktorá obsahuje informácie potrebné na komunikáciu, a to:
    - **`.sin_family`** – typ adresy (AF_INET)
    - **`.sin_addr.s_addr`** – adresa servera (pomocná funkcia na správne napísanie adresy - **inet_addr**; adresa “127.0.0.1” – localhost)
    - **`.sin_port`** – port servera (pomocná funkcia na správne zapísanie portu - **htons**)
- Funkcia **int bind(SOCKET s, const struct sockaddr *name, int namelen)** - slúži na naviazanie adresy a portu servera na soket (implementované tak - bind(listenSocket, (sockaddr*)&socketAddr, sizeof(socketAddr)). Vráti sa 0 alebo 1
- Funckia **int listen(SOCKET s, int backlog)** - funkcia na spustenie počúvania na sokete pre vstupné pripojenia. Takto to vyzerá pre protokol TCP - listen(listenSocket, SOMAXCONN). Vráti sa 0 alebo 1
- Definované **SOMAXCONN** - maximálna dĺžka radu špecifikovateľná počúvaním
- Funkcia **SOCKET accept(SOCKET s, struct sockaddr *addr, int *addrlen)** - akceptuje požiadavku na spustenie komunikácie (prijme a naviaže klienta na soket). Vráti sa soket
- Funkcia **int connect(SOCKET s, const struct sockaddr *name, int namelen)** - spája klienta so serverom. V našej implementácii to vyzerá takto - connect(clientSocket, (sockaddr*)&socketAddr, sizeof(socketAddr)). Vráti sa 0 alebo 1
- Definované **INVALID_SOCKET** - inými slovami -1

- Definované **SOCKET_ERROR** - inými slovami -1
- Pre **TCP/IP** funkcia **int recv(SOCKET s, char *buf, int len, int flags)** - slúži na príjem dát. Na vstupe dostane socket, na ktorý má správa prísť, buffer, do ktorého sa majú zapisovať dáta, veľkosť (počet bajtov) balíka a špeciálne značky. Výstup môže byť -1 alebo 1. Ak komunikácia nebola prerušená, funkcia počká, kým príde táto správa. Príklad z našej implementácie: recv(clientSocket, rbuf, size, 0)
- Pre **TCP/IP** funkcia **int send(SOCKET s, const char *buf, int len, int flags)** - slúži na odosielanie údajov. Na vstupe dostane socket, do ktorého sa majú odosielať dáta, dátový buffer/balík na odoslanie, veľkosť (počet bajtov) balíka a špeciálne značky. Výstup môže byť -1 alebo 1. Príklad z našej implementácie: `send(clientSocket, sbuf, size, 0)`
- **int recvfrom(SOCKET s, char *buf, int len, int flags, struct sockaddr *from, int *fromlen)** – funkcia pre **UDP/IP**, ktorá slúži na príjem dát. Na vstupe dostane socket, na ktorý má správa prísť, buffer, do ktorého sa majú zapisovať dáta, veľkosť (počet bajtov) balíka, špeciálne značky, adresu odosielateľa a dĺžku jeho dátovej adresy. Výstup môže byť -1, 0 alebo 1. Ak komunikácia nebola prerušená, funkcia počká, kým príde táto správa. Príklad z našej implementácie: recvfrom(clientSocket, rbuf, size, 0, (sockaddr*)&serverAddr, &serverAddrSize)
- **int sendto(SOCKET s, const char *buf, int len, int flags, const struct sockaddr *to, int tolen)** - funkcia pre **UDP/IP**, ktorá slúži na odosielanie údajov. Na vstupe dostane socket, do ktorého sa majú odosielať dáta, dátový buffer/balík na odoslanie, veľkosť (počet bajtov) balíka, špeciálne značky, adresa príjemca a dĺžka jeho dátovej adresy. Výstup môže byť -1, 0 alebo 1. Príklad z našej implementácie: sendto(clientSocket, sbuf, strlen(sbuf)+1, 0, (sockaddr*)&serverAddr, serverAddrSize)

## 2.2. Potrebné funkcie knižnice <dis.hxx>:

- Trieda **Server(string name):DimRpc(name.c_str())** - trieda knižnice, ktorá vytvára server DIM s typom komunikácie RPC. Ako vstup akceptuje názov servera a voliteľné poznámky
- Funkcia **void rpcHandler()** - funkcia v rámci triedy, ktorá funguje ako slučka logiky hlavného server
- Funkcia **getString()** – funkcia servera a klienta na čítanie dát z komunikačného kanála
- Funkcia **setData()** - funkcia servera a klienta, ktorá umožňuje odoslanie reťazca alebo iného typu údajov príjemcovi
- Funkcia **DimServer::start(name.c_str())** - spustí server s úvodným názvom

## 2.3. Potrebné funkcie knižnice <dic.hxx>:

- Trieda **DimRpcInfo(string name, char *link)** - triedy sa po zadaní vstupných údajov pripojí k serveru buď menom alebo odkazom
- Táto knižnica má tiež funkcie **getString()** a **setData()**

## 2.4. Potrebné funkcie knižnice <memory.h>:

- Funkcia **void memset(void* _Dst, int _Val, size_t _Size)** – funkcia vyplní oblasť pamäte. Na vstupe preberá adresu pamäte, premennú pomocou ktorej má túto pamäť naplniť a veľkosť, ktorú je potrebné naplniť. Funkcia je “void” (žiadny výstup). Príklad z našej implementácie: memset(rbuf, 0, size)

## 2.5. Potrebné funkcie knižnice <chrono>:

- Typ premennej **auto** - tiež nazývaný **time_point**, používa sa ako časový bod, táto premenná zaznamenáva svoju hodnotu pomocou systémového času.
- Funkcia **static time_point now()** - umožňuje zapísať aktuálny systémový čas do premennej typu “auto”
- Lokálna premenná **duration<double>** - umožňuje vykonávať aritmetické výpočty premenných s typom “auto”. Má svoju funkciu:
    - **constexpr count()** - slúži na konverziu z lokálnej premennej na typ “double” alebo “int”

## 2.6. Potrebné funkcie knižnice <stdio.h>:

- Funkcia **int printf (char const* const _Format)** - zapíše do konzoly dáta prijaté na vstupe
- Funkcia **int scanf(char const* const _Format)** - načítava údaje z konzoly (klávesnice), vo zvolenom formáte
- Funkcia **int sprintf(char* const _Buffer, char const* const _Format)** - zapisuje dáta do bufferu vo zvolenom formáte
- Funkcia **int sscanf(char const* const _Buffer, char const* const _Format)** - zapisuje údaje buffera do premennej vo zvolenom formáte

## 2.7. Potrebné funkcie knižnice <string>:

- Funkcia **int stoi(const string& _Str, size_t* _Idx = nullptr, int _Base = 10)** - táto funkcia konvertuje typ údajov “string” na “int”. Prijíma reťazec ako vstup a číslo, ktoré bolo v reťazci, ako výstup

- Funkcia **string c_str()** - Vráti ukazovateľ na pole, ktoré obsahuje sekvenciu znakov ukončenú nulou (t. j. C-reťazec) predstavujúcu aktuálnu hodnotu objektu reťazca.

## 2.8. Potrebné funkcie knižnice <iostream>:

- Objekt **cout** - objekt triedy ostream, s ním je možné odosielať údaje do konzoly
- Objekt **cin** - objekt triedy ostream, s ním je možné čítať údaje z konzoly (klávesnice)

## 2.9. Príkazy na deaktiváciu veľkého počtu upozornení a chýb počas kompilácie:

- #define _CRT_SECURE_NO_WARNINGS (pre TCP a UDP)
- #define _WINSOCK_DEPRECATED_NO_WARNINGS (pre TCP a UDP)
- #pragma warning(disable:4996) (pre TCP a UDP)
- #pragma comment(lib,"Ws2_32.lib") (pre TCP a UDP)
- #pragma comment(lib, "dim") (pre DIM)

## 2.10. Potrebné funkcie knižnice <ServerLib.h>:

- Funkcia **SOCKET createListenSocket(int port)** - slúži na vytvorenie adresnej premennej, naviazanie tejto premennej na soket a začatie počúvania toho soketu. Na výstupe vráti sa pripravený počúvajúci soket
- Funkcia **SOCKET acceptClient(SOCKET listenSocket)** - akceptuje počúvajúci soket ako vstup a prijíma na ho klienta. Vráti sa na výstupe soket s akceptovaným klientom
- Funkcia **int receiveBuffSize(SOCKET clientSocket, int s)** - na vstupe prijíma klientsky soket a aktuálnu veľkosť komunikačných paketov, pomocou toho a funkcií prijímania / odosielania dát dostáva od klienta informáciu o novej veľkosti paketu, ktorá je výstupom.
- Trieda **ServerLib** - obsahuje vyššie uvedené funkcie

## 2.11. Potrebné funkcie knižnice <ClientLib.h>:

- Funkcia **SOCKET createClientSocket(const char* address, int port)** - vezme adresu a port servera ako vstup, inicializuje Winsock, vytvorí klientsky soket, vytvorí premennú adresy a pripojí tento soket ku klientovi. Výstupom je funkčný klientsky socket
- Funkcia **void sendBuffSize(SOCKET clientSocket, int size, int old)** - prijíma ako vstup klientsky soket pripojený k serveru, novú veľkosť bufferu a starú veľkosť bufferu. Odošle údaje o novej veľkosti bufferu na server. Funkcia je “void” (žiadny výstup)
- Funkcia **void show_info(int pack, int size, double time)** - na vstupe dostane výsledky štúdie, vypočíta ich a zapíše do konzoly. Táto funkcia sa používa na **meranie priepustnosti** a funkcia **void show_info(float min, float max, float avg)** sa používa na **meranie odozvy**. Funkcia je “void” (žiadny výstup)

- Trieda **ClientLib** - obsahuje vyššie uvedené funkcie

### 2.12. Iné:

- Funkcia **void free(void* _Block)** - uvoľní blok pamäte
- Operátor **void delete(void* _Block, size_t _Size)** - uvoľní zvolenú veľkosť v pamäťovom bloku. Ak nie je zadaná veľkosť, uvoľní sa celý blok.
- Funkcia **getchar()** - získa znak ("unsigned char") z stdin

# 3 Softvérové moduly

Ďalej budú napísané kódy samotných programov, vytvorené na porovnanie priepustnosti a odozvy protokolov TCP, UDP a DIM. Každý riadok bude napísaný sem a ich popis bude vyzerať takto – "`//popis`".

## 3.1. Vlastné knižnice

Pre TCP a UDP som vytvoril vlastnú knižnicu **ClientLib.h**, **ServerLib.h** a skripty k nim **ClientLib.cpp**, **ServerLib.cpp**. To umožňuje skrátiť hlavný súbor odstránením ťažkopádnych funkcií, ktoré sa vykonávajú iba raz, napríklad: vytváranie soketov, výpočet výsledkov. V knižniciach som pridal všetky ostatné knižnice, ktoré sa používajú a vytvoril triedu s definovanými verejnými funkciami, ktorých funkčnosť je v príslušnom .cpp súbore (ClientLib.h -> ClientLib.cpp; ServerLib.h -> ServerLib. cpp). Potom v hlavnom súbore používam tieto knižnice a funkcie vytvorenej triedy.

Knižnica klientov (**ClientLib.h**):

```
#define _CRT_SECURE_NO_WARNINGS
#define _WINSOCK_DEPRECATED_NO_WARNINGS
#pragma comment(lib,"Ws2_32.lib")

#include <chrono>
#include <ctime>
#include <stdio.h>
#include <WinSock2.h>
#include <memory.h>
#include <string>
#include <iostream>

class ClientLib {
public:
	SOCKET createClientSocket(const char* address, int port);
	// create client socket (with address 127.0.0.1 and port 2222)
	void   sendBuffSize(SOCKET clientSocket, int size, int old);
	// send new buffer size to server
	void   show_info(int pack, int size, double time);
	// display outputs
};
```

Tieto definované funkcie sú v príslušnom súbore (**ClientLib.cpp**) a su zodpovední za vytvorenie soketa, odoslanie údajov o veľkosti paketu a zobrazenie konečných výsledkov:

```
#include "ClientLib.h"

SOCKET ClientLib::createClientSocket(const char* address, int port)
// create client socket (with address 127.0.0.1 and port 2222)
{
    WSADATA wsaData;  // creating Winsock API
    WSAStartup(MAKEWORD(2, 2), &wsaData);
    // initialize Winsock and selecting version 2.2

    SOCKET clientSocket = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
    // creating client socket

    if (clientSocket == INVALID_SOCKET)
    {
        return clientSocket;
    }
    sockaddr_in socketAddr;   // creating sever address variable
    socketAddr.sin_family = AF_INET;  // address type (TCP/IP)
    socketAddr.sin_addr.s_addr = inet_addr(address); // server address
    socketAddr.sin_port = htons(port); // server port

    if (connect(clientSocket, (sockaddr*)&socketAddr, sizeof(socketAddr)) ==
    SOCKET_ERROR) //connect to server
    {
        printf("Error connecting to server\n");
        closesocket(clientSocket);
        WSACleanup();
        return INVALID_SOCKET;
    }
    return clientSocket;
}

void ClientLib::sendBuffSize(SOCKET clientSocket, int size, int old)
// send new buffer size to server
{
    char* sendSize = new char[old];
    sprintf(sendSize, "%i", size);
    if (send(clientSocket, sendSize, old, 0) == SOCKET_ERROR)
    {
        printf("Send failed\n");
        closesocket(clientSocket);
        WSACleanup();
    }
}

void ClientLib::show_info(int pack, int size, double time) // display outputs
{
    unsigned long bytes = (pack * size) / time;
    printf("%lu", bytes / 1000000);
    printf(" MB per second");
}
```

Knižnica server (**ServerLib.h**) má rovnakú štruktúru, ale funkcie v nej sú zodpovedné za vytvorenie počúvajúceho soketa, prijatie klienta na tejto soket a príjem údajov o novej veľkosti paketu:

```
#include <stdio.h>
#include <WinSock2.h>
#include <memory.h>
#include <string>
```

```
#include <chrono>
#include <ctime>
#include <iostream>

class ServerLib {
public:
    SOCKET  createListenSocket(int port);
    // creating server (with port 2222)
    SOCKET  acceptClient(SOCKET listenSocket);
    // waiting and accepting users
    int     receiveBuffSize(SOCKET clientSocket, int s);
    // receiving new buffer size of packages
};
```

A takto vyzerá funkčnosť súboru **ServerLib.cpp**:

```
#include "ServerLib.h"
#pragma warning(disable:4996)
#define _CRT_SECURE_NO_WARNINGS
#define _WINSOCK_DEPRECATED_NO_WARNINGS
#pragma comment(lib,"Ws2_32.lib")

SOCKET ServerLib::createListenSocket(int port)
{
    WSADATA wsaData;    // creating Winsock API
    WSAStartup(MAKEWORD(2, 2), &wsaData);// selecting version 2.2
    SOCKET listenSocket = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
    // creating listen socket
    if (listenSocket == INVALID_SOCKET){
        return listenSocket;
    }

    sockaddr_in socketAddr; // creating address variable
    socketAddr.sin_family = AF_INET; // address type (TCP/IP)
    socketAddr.sin_addr.s_addr = inet_addr("127.0.0.1");
    //server address (local network)
    socketAddr.sin_port = htons(port); // server port

    if (bind(listenSocket, (sockaddr*)&socketAddr, sizeof(socketAddr)) ==
    SOCKET_ERROR) // bind address variable to unconnected socket
    {
        printf("Error binding socket\n");
        closesocket(listenSocket);
        WSACleanup();
        return INVALID_SOCKET;
    }
    if (listen(listenSocket, SOMAXCONN)==SOCKET_ERROR){//start listening
        printf("Error listening on socket\n");
        closesocket(listenSocket);
        WSACleanup();
        return INVALID_SOCKET;
    }
    return listenSocket;
}
SOCKET ServerLib::acceptClient(SOCKET listenSocket){
// waiting and accepting users
    SOCKET clientSocket = accept(listenSocket, NULL, NULL);
    if (clientSocket == INVALID_SOCKET)
    {
        printf("Error accepting client\n");
        return INVALID_SOCKET;
    }
    return clientSocket;
}
```

```
int ServerLib::receiveBuffSize(SOCKET clientSocket, int s){
// receiving new buffer size of packages
    int size = 0;
    char* r = new char[s];
    memset(r, 0, s);

    if (recv(clientSocket, r, s, 0) == SOCKET_ERROR){
        printf("Receive failed\n");
        closesocket(clientSocket);
        WSACleanup();
        return -1;
    }
    sscanf(r, "%i", &size);
    delete(r);
    return size;
}
```

Pre protokol UDP som zredukoval počet funkcií, keďže kompilátor často nadával na premenné, ktoré by mali byť definované v hlavnom súbore.

Tieto knižnice **neboli** použité pri implementácii protokolu DIM.

## 3.2. Spojenie klient-server

Pomocou knižnice "**WinSock2.h**" sa v serverovom programe vytvorí soket, ku ktorému sa potom pripojí klient. Na tento účel sa najprv inicializuje rozhranie **Winsock API**. Potom sa vytvorí priamo ***listenSocket***, ktorý sa však len tak nenazýva, pretože bude počúvať na akomkoľvek type pripojenia na adrese, ktorá sa naň následne naviaže. Preto je tiež potrebné vytvoriť premennú, ktorá bude uchovávať údaje o adrese servera a porte. Potom pripojme listenSocket k týmto hodnotám a začnime počúvať prichádzajúce požiadavky. ***ServerLib*** a ***ClientLib*** sú knižnice, ktoré sme vytvorili, aby sme zmenšili množstvo hlavného kódu a zlepšili navigáciu.

### 3.2.1. TCP

Serverový program na vytvorenie soketu pre klienta a čakanie na pripojenie:

```
ServerLib slib; // class from my own library
int port = 2222; // stable (faster to test)

SOCKET listenSocket = slib.createListenSocket(port);
// creating server (listen socket)
if (listenSocket == INVALID_SOCKET)
{
    return -1;
}
SOCKET clientSocket = slib.acceptClient(listenSocket);
// accepting users
if (clientSocket == INVALID_SOCKET)
{
    closesocket(listenSocket);
    WSACleanup();
    return -1;
}
printf("Client connected\n");
closesocket(listenSocket);  // close listen socket
```

Klient musí inicializovať WinsockAPI, vytvoriť svoj vlastný soket, vytvoriť premennú s adresou a portom servera a v skutočnosti sa k nemu pripojiť (odoslať požiadavku na pripojenie):

```
char address[16] = "127.0.0.1"; // local network
int port = 2222; // just easier and faster to test it
ClientLib client; // my own class, which contains a few functions
SOCKET clientSocket = client.createClientSocket(address, port);
// connection to server with client socket

if (clientSocket == INVALID_SOCKET)
{
    return -1;
}

printf("Connected to server\n");
```

### 3.2.2. UDP

Spojenie klient-server s protokolom UDP sa robí podobným spôsobom ako TCP. Na rozdiel od TCP, server nemusí dostať pokyn, aby počúval, UDP ho má v režime vždy zapnutý. Vytvorenie servera a čakanie na klienta:

```
ServerLib slib;  // my own class
int port = 2222; // stable (faster to test)

SOCKET serverSocket = slib.createListenSocket(port); // creating listen socket
if (serverSocket == INVALID_SOCKET)
{
    return -1;
}

printf("UDP socket created\n");

sockaddr_in clientAddr; // variable for client address
int clientAddrSize = sizeof(clientAddr);
```

Pripojenie klienta k server:

```
char address[16] = "127.0.0.1"; // stable address and port
int port = 2222;                // so it's faster to test it
ClientLib client;
SOCKET clientSocket = client.createClientSocket(address, port);
// connection to server with client socket

if (clientSocket == INVALID_SOCKET)
{
    return -1;
}
sockaddr_in serverAddr;          // variable for server address
serverAddr.sin_family = AF_INET; // AF_INET - IPv4, AF_INET6 - IPv6
serverAddr.sin_addr.s_addr = inet_addr(address); // server address
serverAddr.sin_port = htons(port); // server port
//this is here, because i will use it in the future to send/receive data
int serverAddrSize = sizeof(serverAddr);
```

### 3.2.3. DIM

Najprv musíme zadať názov servera a službu, pomocou ktorej sa môže klient pripojiť. Potom, vytvoriť požadované triedy a zapnuť tento server. Funkcia "**void rpcHandler**" je hlavný cyclus servera:

```
class Server : public DimRpc
{
public:
    Server(string name) : DimRpc(name.c_str(), "C", "C")
    // class to create server using type RPC
    {
    }

private: // initialize server (service) logic
    void rpcHandler() // server cycle
          {

          }

int main()
    {
    string serverName, serviceName;
    // create variables for server's and service's names
    serverName = "DIM_Meranie";
    serviceName = "DIM_Meranie1";

    Server service(serviceName); // make server
    DimServer::start(serverName.c_str()); // start server

    do
    {

    } while (getchar() != 0);

    return 0;
    }
```

Klient potrebuje vytvoriť premennú s názvom servera a vnútri premennej triedy knižnice, ktorá bude sama o sebe uchovávať informácie o službe. Klient sa tiež okamžite pripojí k serveru:

```
    string serviceName;
    serviceName = "DIM_Meranie1"; // make service name
    char nolink[] = "No RPC link";
    DimRpcInfo service(serviceName.c_str(), nolink); // connect to server
```

## 3.3. Modul servera

Program servera je rovnaký pre oba typy meraní.

### 3.3.1. TCP Server

```
    int size = 0; // buffer size
    int res; // message from client

    size = slib.receiveBuffSize(clientSocket, 1024);
    // waiting for new buffer size from client
    char* rbuf = new char[size]; // creating receive buffer for that size
    char* sbuf = new char[size]; // creating send buffer for that size
```

```
    memset(rbuf, 0, size); // filling it with zerous
    memset(sbuf, 0, size);
    do // endless cycle
    {
        res = 0;
        int result = recv(clientSocket, rbuf, size, 0);
        // receiving package from user
        sscanf(rbuf, "%i", &res); // translating and writing it to res
        if (res == 666) // if the message is "666", then restart
        {
            int unno = size; // present size
            size = slib.receiveBuffSize(clientSocket, unno);
            // waiting for new size
            free(rbuf); // free buffers memo
            free(sbuf);
            rbuf = new char[size]; // create new buffers with new size
            sbuf = new char[size];
            memset(rbuf, 0, size); // fill them
            memset(sbuf, 0, size);// fill it

            printf("Restarted\n"); // to confirm that it's working
        }

        else if (result == SOCKET_ERROR)
       // if there is troubles with receiving information, then close
       // application
        {
            printf("Receive failed\n");
            closesocket(clientSocket);
            WSACleanup();
            return -1;
        }
        if (send(clientSocket, sbuf, size, 0) == SOCKET_ERROR)
        {
            printf("Send failed\n");
            closesocket(clientSocket);
            WSACleanup();
            return -1;
        }

    } while (true); // endless cycle

    closesocket(clientSocket);
    WSACleanup();
    return 0;
```

### 3.3.2. UDP Server

```
    int size = 0; // buffer size
    int res; // message from client
    char* r = new char[1024]; // just for first communication
    char* rbuf; // buffers
    char* sbuf;
    memset(r, 0, 1024);
    int result = recvfrom(serverSocket, r, 1024, 0, (sockaddr*)&clientAddr,
&clientAddrSize); // receive buff size
    if (result == SOCKET_ERROR)
    {
        printf("Receive failed\n");
        closesocket(serverSocket);
        WSACleanup();
        return -1;
    }
    sscanf(r, "%i", &size); // write it to size
```

```
    delete(r); // delete(free) allocated memory


    rbuf = new char[size]; // create buffers for size of "size"
    sbuf = new char[size];
    memset(sbuf, 0, size); // fill them
    memset(rbuf, 0, size);

    do // endless cycle
    {
        res = 0;
        int result = recvfrom(serverSocket, rbuf, size, 0,
(sockaddr*)&clientAddr, &clientAddrSize); // receiving package from user
        sscanf(rbuf, "%i", &res); // translating and writing it to res
        if (res == 666) // if the message is "666", then restart
        {
            int unno = size; // present size
            char* r = new char[unno];
            // create buffer for old communication size
            memset(r, 0, unno);
            int result = recvfrom(serverSocket, r, unno, 0,
(sockaddr*)&clientAddr, &clientAddrSize); // receive new buffer size
            if (result == SOCKET_ERROR)
            {
                printf("Receive failed\n");
                closesocket(serverSocket);
                WSACleanup();
                return -1;
            }
            sscanf(r, "%i", &size); // write new size to "size"
            delete(r); // free memo
            free(rbuf);
            free(sbuf);// deleting allocated memo
            rbuf = new char[size];
            sbuf = new char[size]; // create new buffer with new size
            memset(sbuf, 0, size);// create new buffer with new size
            memset(rbuf, 0, size); // fill it

            printf("Restarted\n"); // to confirm that it's working
        }

        else if (result == SOCKET_ERROR)
        // if there is troubles with receiving information, then close
        // application
        {
            printf("Receive failed\n");
            closesocket(serverSocket);
            WSACleanup();
            return -1;
        }
        if (sendto(serverSocket, sbuf, strlen(sbuf) + 1, 0,
(sockaddr*)&clientAddr, clientAddrSize) == SOCKET_ERROR) // send answer
        {
            printf("Send failed\n");
            closesocket(serverSocket);
            WSACleanup();
            return -1;
        }
    } while (true); // endless cycle

    closesocket(serverSocket);
    WSACleanup();
    return 0;
```

### 3.3.3. DIM Server

```
string text = ""; // receive buffer
string response = ""; // send buffer
int size; // size of packages
int packages = 0;

bool rerun = true; // restart bool

void rpcHandler() // server cycle
{
    text = getString(); // get data
    if (text == "666") // check if it's restart command
    {
        rerun = true;
    }
    if (rerun == true) // restart command
    {
        text = getString(); // get new buffer size
        rerun = false;
        size = stoi(text); // write it to int
        text = ""; // clear buffers
        response = "";
        cout << packages << endl;;
        packages = 0;
        for (int i = 0; i < size / 8; i++)
        {
            response += "0"; // create new response
        }
    }
    else // send response
    {
        setData((char*)response.c_str());
        packages++;
    }
}
```

## 3.4. Modul klienta pri meraní priepustnosti

### 3.4.1. TCP

```
// define necessary values
int size; // size of send/receive packages
int old = 1024; // old size of packages (useful for restarting cycle)
int sec = 60; // time window
char* sbuf; // send buffer
char* rbuf; // receive buffer
int packs = 0; // ammount of sent packages

printf("Enter the buffer size: ");
while (scanf("%i", &size) != NULL)
// infinite cycle, but it waits for an input
{
    client.sendBuffSize(clientSocket, size, old);
    // send new buffer size to the server
    sbuf = new char[size];
    // create new send buffer with the size of "size"
    rbuf = new char[size];
    // create new receive buffer with the size of "size"
    memset(rbuf, 0, size);
    memset(sbuf, 0, size); // fill them

    auto start = chrono::system_clock::now(); // start the timer
    auto end = chrono::system_clock::now();
```

```
        chrono::duration<double> elapsed_seconds = end - start;
        // elapsed time
        double delta = elapsed_seconds.count();
        // making delta != NULL to start the cycle
        while (delta <= sec)
        // sec = casove okno, delta = now - start  is time (kolko casu trva)
        {
            if (send(clientSocket, sbuf, size, 0) == SOCKET_ERROR)
            // send this buffer to server
            {
                printf("Send failed\n");
                closesocket(clientSocket);
                WSACleanup();
                return -1;
                // if send failed, stop the script
            }
            if (recv(clientSocket, rbuf, size, 0) == SOCKET_ERROR)
            // wait for an answer
            {
                printf("Error receiving data\n");
                // if the answer was received, then proceed. else - stop the
                // program
                closesocket(clientSocket);
                WSACleanup();
                return -1;
            }

            auto end = chrono::system_clock::now(); // check now time
            chrono::duration<double> elapsed_seconds = end - start;
            delta = elapsed_seconds.count(); // make a new delta
            packs++; // succeeded sending a package. All sended packages + 1
        }

        sprintf(sbuf, "%i", 666);
        // server is always checking received buffer if it's "666" then he
        // starts waiting for new buffer size
        if (send(clientSocket, sbuf, size, 0) == SOCKET_ERROR)
        // send server command to restart
        {
            printf("Send failed\n");
            closesocket(clientSocket);
            WSACleanup();
            return -1;
        }
        client.show_info(packs, size, sec); // show all necessary outputs
        printf("\nEnter the buffer size: "); // print this message for a user
        old = size;
        // server is still working with packages of, for example, 1024. To
        // communicate with him, client needs to now this size
        packs = 0;
        // cycle restarts
    }
    closesocket(clientSocket);
    WSACleanup();
    return 0;
```

### 3.4.2. UDP

```
// define necessary values
int size; // size of send/receive packages
int old = 1024; // old size of packages (useful for restarting cycle)
int sec = 60; // time window
char* sbuf; // send buffer
char* rbuf; // receive buffer
```

```
    int packs = 0; // ammount of sent packages
    char* sendSize = new char[old]; // for first transaction

    printf("Enter the buffer size: ");

    while (scanf("%i", &size) != NULL)
    // infinite cycle, but it waits for an input
    {
        sendSize = new char[old];
        sprintf(sendSize, "%i", size);
        if (sendto(clientSocket, sendSize, strlen(sendSize)+1, 0,
(sockaddr*)&serverAddr, serverAddrSize) == SOCKET_ERROR)
        {
            printf("Send failed\n");
            closesocket(clientSocket);
            WSACleanup();
        } // send new buffer size to the server
        sbuf = new char[size]; // create new buffer with the size of "size"
        rbuf = new char[size]; // create new buffer with the size of "size"
        memset(rbuf, 0, size);
        memset(sbuf, 0, size); // fill it

        auto start = chrono::system_clock::now(); // start the timer
        auto end = chrono::system_clock::now();
        chrono::duration<double> elapsed_seconds = end - start;
        double delta = elapsed_seconds.count();
        // making delta != NULL to start the cycle

        while (delta <= sec)
        // sec = casove okno, delta = now - start time (kolko casu trva)
        {
            if (sendto(clientSocket, sbuf, strlen(sbuf)+1, 0,
(sockaddr*)&serverAddr, serverAddrSize) == SOCKET_ERROR)
// send this buffer to server
            {
                printf("Send failed\n");
                closesocket(clientSocket);
                WSACleanup();
                return -1;  // if send failed, stop the script
            }
            if (recvfrom(clientSocket, rbuf, size, 0, (sockaddr*)&serverAddr,
&serverAddrSize) <= 0) // wait for an answer
            {
                printf("Error receiving data\n");
                // if the answer was received, then proceed. else - stop the
                // program
                closesocket(clientSocket);
                WSACleanup();
                return -1;
            }
            auto end = chrono::system_clock::now(); // check now time
            chrono::duration<double> elapsed_seconds = end - start;
            delta = elapsed_seconds.count(); // make a new delta
            packs++; // succeeded sending a package. All sended packages + 1
        }

        sprintf(sbuf, "%i", 666);
        // server is always checking received buffer if it's "666" then he
        // starts waiting for new buffer size
        if (sendto(clientSocket, sbuf, strlen(sbuf) + 1, 0,
(sockaddr*)&serverAddr, serverAddrSize) == SOCKET_ERROR)
// send server command to restart
        {
            printf("Send failed\n");
            closesocket(clientSocket);
```

```
            WSACleanup();
            return -1;
        }
        client.show_info(packs, size, sec); // show all necessary outputs
        printf("\nEnter the buffer size: "); // print this message for a user
        old = size;
        // server is still working with packages of, for example, 1024.
        // To communicate with him, client needs to now this size
        packs = 0;
        free(sendSize);
        // cycle restarts
    }
    closesocket(clientSocket);
    WSACleanup();
    return 0;
```

### 3.4.3. DIM

```
int size; // size of buffer
int sec = 60; // time window
int packs = 0; // ammount of sent packages
string text = ""; // input text and send buffer
string response; // receive buffer
string restart = "666"; // restart command
while (1) // always true
{
    cout << "Enter size of buffer: ";
    cin >> text; // new buffer size

    service.setData((char*)text.c_str()); // send it to server
    size = stoi(text); // write it to "size"
    text = ""; // clear it
    for (int i = 0; i < size / 8; i++)
    {
        text += "0"; // making filled buffer
    }

    auto start = chrono::system_clock::now(); // start the timer
    auto end = chrono::system_clock::now();
    chrono::duration<double> elapsed_seconds = end - start;
    double delta = elapsed_seconds.count(); // making delta != NULL

    while (delta <= sec) // start testing
    {
        service.setData((char*)text.c_str()); // send buffer to server
        response = service.getString(); // receive the response

        auto end = chrono::system_clock::now(); // check now time
        chrono::duration<double> elapsed_seconds = end - start;
        delta = elapsed_seconds.count(); // make a new delta
        packs++; // increment number of succeed transactions by one
    }

    service.setData((char*)restart.c_str()); // send restart command
    float bytes = ((packs * size) / 1000) / sec; // calculate results
    cout << bytes << " KB per second" << endl; // write it to the console
    cout << packs << endl;
    packs = 0; // clear variables
    text = "";
}

return 0;
```

## 3.5. Modul klienta pri meraní odozvy

### 3.5.1. TCP

```
int size;          // size - size of buffer
int old = 1024;    // old - previous size of buffer
char* sbuf;        // sbuf - send buffer
char* rbuf;        // rbuf - receive buffer
int times = 1000000;
// times - send the packages with chosen size this many times
float min = 1000;  // min - minimal delay
float max = -1000; // max - maximum delay
float avg = 0;     // avg - average delay
double ping = 0;
// ping - time delay(time spent between sending and receiving)

printf("Enter the buffer size: ");

while (scanf("%i", &size) != NULL) // infinite cycle
{
    client.sendBuffSize(clientSocket, size, old);
    // send new buffer size to
    // server, (size - new size of the buffer, old - previous size of the
    // buffer (1024 by default))

    sbuf = new char[size]; // create send and receive buffers
    rbuf = new char[size];
    memset(sbuf, 0, size); // fill them
    memset(rbuf, 0, size);

    for (int i = 0; i < times; i++)
    // a cycle, which will repeat "times" times
    {
        auto start = chrono::system_clock::now(); // starting timer
        if (send(clientSocket, sbuf, size, 0) == SOCKET_ERROR)
        // send package to server
        {
            printf("Send failed\n");
            closesocket(clientSocket);
            WSACleanup();
            return -1;
        }
        if (recv(clientSocket, rbuf, size, 0) == SOCKET_ERROR)
        // wait for an answer
        {
            printf("Error receiving data\n");
            // if the answer was received, then proceed. else - stop the
            // program
            closesocket(clientSocket);
            WSACleanup();
            return -1;
        }
        auto end = chrono::system_clock::now();
        // after answer received, end the timer
        chrono::duration<double> elapsed_seconds = end - start;
        // calculate how much time was spent
        double ping = elapsed_seconds.count();  // write it to "ping"

        if (max == -1000 && min == 1000)
        // if it's first attempt, then "avg = delay"
        {
            avg = ping;
        }
        else // if not, then just a general formula
```

```
            {
                avg = (avg + ping) / 2;
            }
            if (ping > max) // check if it is new maximum
            {
                max = ping;
            }
            if (ping < min) // or minimum
            {
                min = ping;
            }
        }

        sprintf(sbuf, "%i", 666);
        // send server package, which makes it to restart
        if (send(clientSocket, sbuf, size, 0) == SOCKET_ERROR)
        {
            printf("Send failed\n");
            closesocket(clientSocket);
            WSACleanup();
            return -1;
        }
        client.show_info(min,max,avg); // show outputs
        printf("\nEnter the buffer size: "); // restarting client
        old = size; // restarting values
        avg = 0;
        ping = 0;
        min = 1000;
        max = -1000;
    }
    closesocket(clientSocket);
    WSACleanup();
    return 0;
```

### 3.5.2. UDP

```
    int size;           // size - size of buffer
    int old = 1024;     // old - previous size of buffer
    char* sbuf;         // sbuf - send buffer
    char* rbuf;         // rbuf - receive buffer
    int times = 1000000;
    // times - send the packages with chosen size this many times
    float min = 1000;  // min - minimal delay
    float max = -1000; // max - maximum delay
    float avg = 0;     // avg - average delay
    double ping = 0;
    // ping - time delay(time spent between sending and receiving)

    char* sendSize = new char[old];
    printf("Enter the buffer size: ");
    while (scanf("%i", &size) != NULL) // infinite cycle
    {
        sendSize = new char[old];
        sprintf(sendSize, "%i", size);
        if (sendto(clientSocket, sendSize, strlen(sendSize) + 1, 0,
(sockaddr*)&serverAddr, serverAddrSize) == SOCKET_ERROR)
        {
            printf("Send failed\n");
            closesocket(clientSocket);
            WSACleanup();
        } // send new buffer size to the server

        sbuf = new char[size]; // create send and receive buffers
        rbuf = new char[size];
```

```
        memset(sbuf, 0, size); // fill them
        memset(rbuf, 0, size);

        for (int i = 0; i < times; i++) // a cycle, which will repeat "times"
times
        {
            auto start = chrono::system_clock::now(); // starting timer
            if (sendto(clientSocket, sbuf, strlen(sbuf) + 1, 0,
(sockaddr*)&serverAddr, serverAddrSize) == SOCKET_ERROR)
// send package to server (starting point)
            {
                printf("Send failed\n");
                closesocket(clientSocket);
                WSACleanup();
                return -1;
            }
            if (recvfrom(clientSocket, rbuf, size, 0, (sockaddr*)&serverAddr,
&serverAddrSize) <= 0) // wait for an answer
            {
                printf("Error receiving data\n");
                // if the answer was received, then proceed. else - stop the
                // program
                closesocket(clientSocket);
                WSACleanup();
                return -1;
            }

            auto end = chrono::system_clock::now();
            // after answer received, end the timer
            chrono::duration<double> elapsed_seconds = end - start;
            // calculate how much time was spent
            double ping = elapsed_seconds.count();  // write it to "ping"


            if (max == -1000 && min == 1000)
            // if it's first attempt, then "avg = delay"
            {
                avg = ping;
            }
            else // if not, then just a general formula
            {
                avg = (avg + ping) / 2;
            }
            if (ping > max) // check if it is new maximum
            {
                max = ping;
            }
            if (ping < min) // or minimum
            {
                min = ping;
            }
        }

        sprintf(sbuf, "%i", 666); // send server package, which makes it to
restart
        if (sendto(clientSocket, sbuf, strlen(sbuf) + 1, 0,
(sockaddr*)&serverAddr, serverAddrSize) == SOCKET_ERROR)
        {
            printf("Send failed\n");
            closesocket(clientSocket);
            WSACleanup();
            return -1;
        }
        client.show_info(min, max, avg); // show outputs
        printf("\nEnter the buffer size: "); // restarting client
```

```
        old = size;  // restarting values
        avg = 0;
        ping = 0;
        min = 1000;
        max = -1000;
        free(sendSize);
    }
    closesocket(clientSocket);
    WSACleanup();
    return 0;
```

### 3.5.3. DIM

```
int times = 1000000; // repeat times
int size; // size of buffer
string text = ""; // input text and send buffer
string response; // receive buffer
string restart = "666"; // restart command

float min = 1000;  // min - minimal delay
float max = -1000; // max - maximum delay
float avg = 0;     // avg - average delay
double ping = 0;   // spent time for transaction


while (1) // always true
{
    cout << "Enter size of buffer: ";
    cin >> text; // new buffer size

    service.setData((char*)text.c_str()); // send it to server
    size = stoi(text); // write it to "size"
    text = ""; // clear it
    for (int i = 0; i < size; i++)
    {
        text += "0"; // making filled buffer
    }

    for(int i = 0; i < times; i++)
    {
        auto start = chrono::system_clock::now(); // start the timer

        service.setData((char*)text.c_str()); // send buffer to server
        response = service.getString(); // receive the response

        auto end = chrono::system_clock::now();
        // after answer received, end the timer
        chrono::duration<double> elapsed_seconds = end - start;
        // calculate how much time was spent
        double ping = elapsed_seconds.count(); // write it to "ping"

        if (max == -1000 && min == 1000)
        // if it's first attempt, then "avg = delay"
            avg = ping;

        else // if not, then just a general formula
            avg = (avg + ping) / 2;

        if (ping > max) // check if it is new maximum
            max = ping;

        if (ping < min) // or minimum
            min = ping;
    }
```

```
        service.setData((char*)restart.c_str()); // send reset command

        cout << "Minimum: " << min << " s" << endl; // print results
        cout << "Maximum: " << max << " s" << endl;
        cout << "Average: " << avg << " s" << endl;

        avg = 0; // reset variables
        ping = 0;
        min = 1000;
        max = -1000;
        text = "";
    }

    return 0;
```

# Záver

Táto systémová príručka informuje programátora o popise jednotlivých funkcií, tried, lokálnych premenných na prípadné rozšírenie aplikácie. Informuje tiež o tom, ako a prečo sa používa funkcia alebo trieda a do ktorej knižnice patria.