

TECHNICKÁ UNIVERZITA V KOŠICIACH
FAKULTA ELEKTROTECHNIKY A INFORMATIKY

POROVNANIE PRIEPUSTNOSTI A ODOZVY SIEŤOVÝCH
KOMUNIKAČNÝCH PROTOKOLOV
BAKALÁRSKA PRÁCA

TECHNICKÁ UNIVERZITA V KOŠICIACH
FAKULTA ELEKTROTECHNIKY A INFORMATIKY

**POROVNANIE PRIEPUSTNOSTI A ODOZVY SIEŤOVÝCH
KOMUNIKAČNÝCH PROTOKOLOV**
BAKALÁRSKA PRÁCA

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

Abstrakt v SJ

Táto bakalárska práca slúži na vytvorenie spojenia Klient-Server protokolmi TCP/IP, UDP/IP, DIM a porovnanie ich priepustnosti dvoma spôsobmi:

1. Meranie priepustnosti;
2. Meranie odozvy.

Opisuje tiež spôsob pripojenia a výhody/nevýhody používania týchto konkrétnych sieťových protokolov. Naše programy boli vytvorené pomocou programovacieho jazyka C++ s použitím knižnice „WinSock2.h“ pre TCP/IP a UDP/IP, pre DIM knižnice „dic.hxx“ a „dis.hxx“. Pre DIM bol zvolený pracovný mechanizmus podobný TCP a UDP, ktorý sa nazýva RPC.

Kľúčové slová v SJ

TCP, UDP, DIM, IP, Buffer, RPC, Soket, Balík, Winsock

Abstrakt v AJ

This bachelor thesis is used to create a Client-Server connection using TCP/IP, UDP/IP, DIM protocols and compare their throughput in two ways:

1. Throughput measurement;
2. Latency measurement.

It also describes the connection method and the advantages / disadvantages of using these specific network protocols. Our programs were created using the C++ programming language with the library „WinSock2.h“ library for TCP/IP and UDP/IP, „dic.hxx“ and „dis.hxx“ libraries for DIM. A working mechanism similar to TCP and UDP, called RPC, was chosen for DIM.

Kľúčové slová v AJ

TCP, UDP, DIM, IP, Buffer, RPC, Socket, Package, Winsock

TECHNICKÁ UNIVERZITA V KOŠICIACH
FAKULTA ELEKTROTECHNIKY A INFORMATIKY
Katedra kybernetiky a umelej inteligencie

ZADANIE
BAKALÁRSKEJ PRÁCE

Študijný odbor: **Informatika**
Študijný program: **Inteligentné systémy**

Názov práce:

Porovnanie priepustnosti a odozvy sieťových komunikačných protokolov
Comparison of the throughput and response of network communication protocols

Študent: **Pavlo Matush**
Školiteľ: **doc. Ing. Ján Jadlovský, CSc.**
Školiace pracovisko: **Katedra kybernetiky a umelej inteligencie**
Konzultant práce: **Ing. Milan Tkáčik**
Pracovisko konzultanta: **Katedra kybernetiky a umelej inteligencie**

Pokyny na vypracovanie bakalárskej práce:

1. Naštudovať sieťové komunikačné protokoly TCP/IP, UDP/IP a DIM.
2. Programovo realizovať aplikáciu pre meranie priepustnosti uvedených protokolov z hľadiska množstva prenesených dát za zadefinované časové okno.
3. Programovo realizovať aplikáciu pre meranie odozvy uvedených protokolov z hľadiska doby čakania medzi odoslaním žiadosti a prijatím odpovede.
4. Vykonať merania s oboma typmi aplikácií s rôznymi veľkosťami posielaných správ.
5. Vyhodnotiť vykonané merania, porovnať priepustnosti a odozvy uvedených protokolov.
6. Vypracovať dokumentáciu podľa pokynov katedry a vedúceho práce (napr. hlavná časť 30-40 strán, tlačaná forma v nerozoberateľnej väzbe, používateľská a systémová príručka, CD/DVD obsahujúce príslušné dokumenty a softvérový výstup práce).

Jazyk, v ktorom sa práca vypracuje: slovenský
Termín pre odovzdanie práce: 31.08.2022
Dátum zadania bakalárskej práce: 29.10.2021

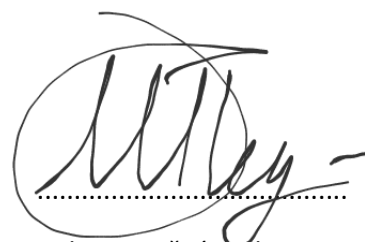


prof. Ing. Liberios Vokorokos, PhD.
dekan fakulty

Čestné vyhlásenie

Vyhlasujem, že som celú bakalársku prácu vypracoval samostatne s použitím uvedenej odbornej literatúry.

Košice, 05. septembra 2022



vlastnoručný podpis

PodĎakovanie

PodĎakovanie patrí vedúcemu mojej bakalárskej práce doc. Ing. Jánovi Jadlovskému, CSc., a konzultantovi Ing. Milanovi Tkáčikovi za neustálu pomoc a podporu pri práci. Taktiež sa chcem poďakovať Ing. Tomášovi Tkáčikovi za pomoc s dizajnom a za pomoc s opravou chýb. A tiež mojej priateľke Ing. Inne Zalužnej za pomoc s prekladom do slovenčiny.

Obsah

Zoznam tabuliek	8
Zoznam obrázkov	9
Zoznam symbolov a skratiek	11
Úvod	12
1. Úlohy a cieľ práce	13
2. Sieťové komunikačné protokoly.....	14
2.1. Transmission Control Protocol.....	16
2.2. User Datagram Protocol.....	22
2.3. Distributed Information Management.....	25
3. Spojenie klient-server.....	31
3.1. TCP/IP	31
3.2. UDP/IP	36
3.3. DIM.....	40
4. Softvérová implementácia	42
4.1. Meranie priepustnosti.....	42
4.2. Meranie odozvy.....	52
5. Porovnanie internetových protokolov	57
5.1. Meranie priepustnosti.....	57
5.2. Meranie odozvy.....	61
Záver.....	69
Zoznam použitej literatúry	70
Prílohy	72

Zoznam tabuliek

Tabuľka 1 TCP Formát hlavičky	18
Tabuľka 2 Stavy pripojenia TCP	21
Tabuľka 3 Porovnanie množstva a rýchlosti TCP/IP	57
Tabuľka 4 Porovnanie množstva a rýchlosti UDP/IP	58
Tabuľka 5 Porovnanie množstva a rýchlosti DIM	59
Tabuľka 6 Výsledky oneskorenia odozvy protokolu TCP/IP	62
Tabuľka 7 Výsledky oneskorenia odozvy protokolu UDP/IP	64
Tabuľka 8 Výsledky oneskorenia odozvy protokolu DIM	66

Zoznam obrázkov

Obrázok 1 OSI/ISO referenčný model	14
Obrázok 2 Referenčný model TCP/IP	17
Obrázok 3 TCP vs UDP konektivita	23
Obrázok 4 UDP hlavička [10]	24
Obrázok 5 DIM diagram [12]	26
Obrázok 6 RCP mechanizmus	26
Obrázok 7 DIM mechanizmus	27
Obrázok 8 Vývojový diagram funkcie „createListenSocket“ TCP	32
Obrázok 9 Vývojový diagram funkcie „acceptClient“ TCP.....	33
Obrázok 10 Vytvorenie servera a pripojenie klienta k nemu v hlavnom programe pomocou protokolu TCP.....	34
Obrázok 11 Vývojový diagram funkcie „createClientSocket“ TCP	35
Obrázok 12 Vytvorenie klientskeho soketa a pripojenie ho ku serveru v hlavnom súbore pomocou protokolu TCP.....	36
Obrázok 13 Vývojový diagram funkcie „createListenSocket“ UDP	37
Obrázok 14 Vytvorenie servera v hlavnom programe pomocou protokolu UDP	38
Obrázok 15 Vývojový diagram funkcie „createClientSocket“ UDP	39
Obrázok 16 Pripojenie klienta k serveru v hlavnom súbore pomocou protokolu UDP	39
Obrázok 17 Vytvorenie servera s typom komunikácie RPC pomocou protokolu DIM	40
Obrázok 18 Vytvorenie klienta s typom komunikácie RPC pomocou protokolu DIM	41
Obrázok 19 Vývojový diagram funkcie „receiveBufferSize“ TCP a UDP.....	43
Obrázok 20 Logika serverového programu pri práci s protokolmi TCP a UDP	44
Obrázok 21 Logika serverového programu pri práci s protokolom DIM.....	45
Obrázok 22 Vývojový diagram funkcie „sendBufferSize“ TCP a UDP.....	46
Obrázok 23 Vývojový diagram funkcie pri meraní priepustnosti „show_info“ TCP a UDP	47
Obrázok 24 Klientsky program pre meranie priepustnosti pomocou protokolov TCP a UDP. Časť 1	48
Obrázok 25 Klientsky program pre meranie priepustnosti pomocou protokolov TCP a UDP. Časť 2	49
Obrázok 26 Klientsky program pre meranie priepustnosti pomocou protokolu DIM	51
Obrázok 27 Vývojový diagram funkcie pri meraní odozvy „show_info“ TCP a UDP	52
Obrázok 28 Klientsky program pre meranie odozvy pomocou protokolov TCP a UDP. Časť 1	53
Obrázok 29 Klientsky program pre meranie odozvy pomocou protokolov TCP a UDP. Časť 2	54

Obrázok 30 Klientsky program pre meranie odozvy pomocou protokolu DIM. Časť 1	55
Obrázok 31 Klientsky program pre meranie odozvy pomocou protokolu DIM. Časť 2	56
Obrázok 32 Priepustnosť TCP/IP	57
Obrázok 33 Priepustnosť UDP/IP	58
Obrázok 34 Priepustnosť DIM	59
Obrázok 35 Porovnanie výsledkov protokolov pri meraní priepustnosti	60
Obrázok 36 Minimálne oneskorenie protokolu TCP/IP	61
Obrázok 37 Maximálne oneskorenie protokolu TCP/IP	61
Obrázok 38 Priemerné oneskorenie protokolu TCP/IP	62
Obrázok 39 Minimálne oneskorenie protokolu UDP/IP	63
Obrázok 40 Maximálne oneskorenie protokolu UDP/IP	63
Obrázok 41 Priemerné oneskorenie protokolu UDP/IP	64
Obrázok 42 Minimálne oneskorenie protokolu DIM	65
Obrázok 43 Maximálne oneskorenie protokolu DIM	65
Obrázok 44 Priemerné oneskorenie protokolu DIM	66
Obrázok 45 Porovnanie minimálneho oneskorenia protokolov	67
Obrázok 46 Porovnanie maximálneho oneskorenia protokolov	67
Obrázok 47 Porovnanie priemerného oneskorenia protokolov	68

Zoznam symbolov a skratiek

ACK – Acknowledgment

API – Application Programming Interface

DIM – Distributed Information Management (System)

DLL – Data Link Layer

IP – Internet Protocol

ISO – International Organization for Standardization

MAC – Media Access Control

OSI – Open Systems Interconnection

SPI – Service Provider Interface

SYN – Synchronize Sequence Numbers

TCP – Transmission Control Protocol

UDP – User Datagram Protocol

Úvod

Vždy ma zaujímalo, ako sa vytvárajú spojenia klient-server a ako si vymieňajú dáta. Sieťové protokoly slúžia na výmenu dát na počítači pripojenom na internet, od prehliadania stránky až po komunikáciu s priateľmi pomocou sociálnych sietí. Pri výbere bakalárskej práce ma táto téma hneď zaujala, keďže mi tieto znalosti pomôžu v mnohých oblastiach IT, napríklad pri tvorbe online hry. Podľa môjho názoru, výber správneho a rýchleho protokolu zohráva veľkú úlohu pri vytváraní servera.

V našej bakalárskej práci sa budeme zaoberať sieťovými protokolmi TCP/IP, UDP/IP a DIM, poukážeme na ich výhody a nevýhody. Budeme merať ich priepustnosť a rýchlosť odozvy s rôznym množstvom odoslaných dát. Vzájomne ich porovnáme a zo získaných výsledkov vyvodíme závery.

Na úvod si rozoberieme a stručne popíšeme všetky dôležité informácie o týchto protokoloch. Popíšeme ich nuansy a podrobne analyzujeme, ako organizujú komunikáciu (ako sa spájajú, ako kódujú informácie atď.). K tomu vytvoríme niekoľko programov na meranie priepustnosti a rýchlosti odozvy. Potom výsledky týchto experimentov navzájom porovnáme, aby sme zistili, ktorý protokol je najvhodnejší na použitie v komunikácii klient-server. V tejto práci budeme používať typ pripojenia RPC. Potom pomocou grafov vizuálne ukážeme rozdiel v ich priepustnosti / rýchlosti odozvy a vyvodíme záver.

1. Úlohy a cieľ práce

Aby sme pochopili, čo sú tieto protokoly a ako organizujú komunikáciu medzi klientskymi a serverovými programami, prvým krokom je analyzovať dostupné informácie o nich a naučiť sa potrebné informácie, ktoré potom možno použiť:

- Naštudovať sieťové komunikačné protokoly TCP/IP, UDP/IP a DIM.

Keď sme sa už oboznámili so spôsobmi fungovania týchto protokolov, môžeme už začať vytvárať programy na určenie ich priepustnosti a rýchlosti odozvy, čo je hlavná úloha. Na **meranie priepustnosti** bol vytvorený program, ktorý meria počet odoslaných paketov za určité časové obdobie, po ktorom sa zobrazí celkový počet odoslaných dát lomeno za uplynutý čas. Výsledkom bude množstvo dát odoslaných za sekundu (v MB/s). Na **meranie odozvy** bol vytvorený program, ktorý meria čas, ktorý uplynul medzi odoslaním požiadavky a prijatím odpovede zo servera. Výsledkom tejto skúsenosti bude minimálne, maximálne a priemerné oneskorenie pre danú komunikáciu. Nižšie stručne zhrnieme vyššie uvedené:

- Programovo realizovať aplikáciu pre meranie priepustnosti uvedených protokolov z hľadiska množstva prenesených dát za zadefinované časové okno.
- Programovo realizovať aplikáciu pre meranie odozvy uvedených protokolov z hľadiska doby čakania medzi odoslaním žiadosti a prijatím odpovede.

Po vytvorení pracovných programov prejdeme k testovaniu a zaznamenávaniu výsledkov. Pakety budú mať nasledujúce veľkosti – 64, 128, 256, 1024 a 2048 bajtov. Na meranie priepustnosti sme zvolili 60 sekundové časové okno. Na meranie odozvy bol zvolený milión opakovaní. Stručne:

- Vykonať merania s oboma typmi aplikácií s rôznymi veľkosťami posielaných správ.

Aby bolo jasné, umiestnili sme ich výsledky do grafov, z ktorých teraz môžeme jednoducho zvýrazniť, ktorý protokol spracováva, odosiela a prijíma údaje lepšie ako ostatné:

- Vyhodnotiť vykonané merania, porovnať priepustnosti a odozvy uvedených protokolov.

Po porovnaní je v grafe znázornený rozdiel v rýchlosti jednotlivých protokolov, z čoho je možné vyvodiť záver.

2. Sieťové komunikačné protokoly

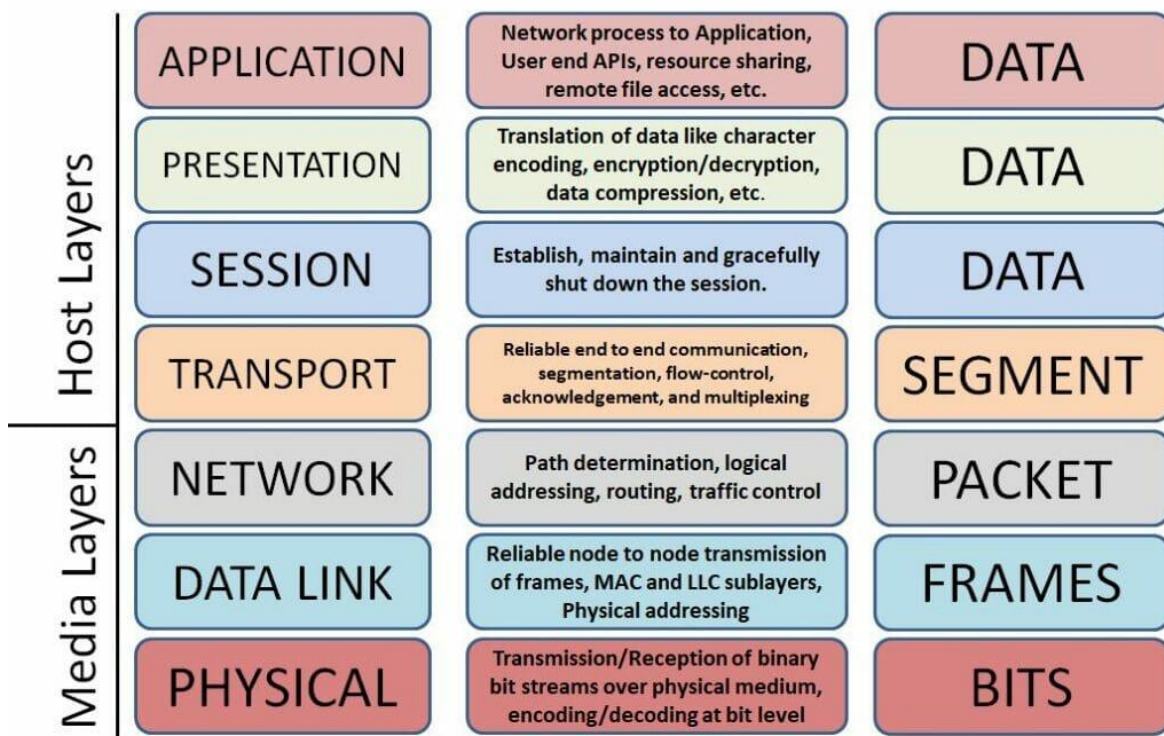
Protokoly sú súbory pravidiel pre formáty správ a procedúry, ktoré umožňujú strojom a aplikačným programom vymieňať si informácie. Tieto pravidlá musí dodržiavať každý stroj zapojený do komunikácie, aby prijímajúci klient bol schopný porozumieť správe.

V súčasnosti existuje pomerne rozsiahly zoznam rôznych protokolov, ktoré zabezpečujú prenos údajov medzi zariadeniami. Zo všetkých vyberáme široko používané TCP a UDP, ale DIM nie je zatiaľ populárnym.

Dnes existuje štandardizovaný model **OSI/ISO**, ktorý sa široko používa na opis systémov dátovej komunikácie. Ak hovoríme o protokoloch ako TCP a UDP, tento referenčný model je vhodne spomenúť.

OSI Model

OSI je 7-vrstvová architektúra, pričom každá vrstva má špecifické funkcie [1]. Všetkých týchto 7 vrstiev spolupracuje na prenose údajov. Obrázok 1 stručne popisuje tieto vrstvy.



Obrázok 1 OSI/ISO referenčný model

Najnižšou vrstvou referenčného modelu OSI je **fyzická vrstva**. Fyzická vrstva obsahuje informácie vo forme bitov. Je zodpovedná za skutočné fyzické spojenie medzi zariadeniami a za prenos jednotlivých bitov z jedného uzla do druhého.

Vrstva dátového spojenia je zodpovedná za doručenie správy medzi uzlom. Hlavnou funkciou tejto vrstvy je zabezpečiť bezchybný prenos údajov z jedného uzla do druhého cez fyzickú vrstvu. Keď paket dorazí do siete, je zodpovednosťou **Data Link Layer (DLL)** preniesť ho do hostiteľa pomocou jeho **MAC adresy**.

Sieťová vrstva slúži na prenos údajov z jedného hostiteľa na druhý umiestnený v rôznych sieťach. Tiež sa stará o smerovanie paketov, t.j. výber najkratšej cesty na prenos paketu z počtu dostupných ciest. IP adresy odosielateľa a príjemcu sú umiestnené v hlavičke sieťovej vrstvy.

Transportná vrstva poskytuje služby aplikačnej vrstve a preberá služby zo sieťovej vrstvy. Je zodpovedná za doručenie kompletnej správy z jedného konca na druhý. Transportná vrstva tiež poskytuje potvrdenie o úspešnom prenose dát a opätovne prenáša dáta, ak sa nájde chyba.

Vrstva relácie je zodpovedná za nadviazanie spojenia, údržbu relácií, autentifikáciu a zaisťuje bezpečnosť.

Prezentačná vrstva (alebo **prekladová vrstva**) extrahuje a manipuluje údaje z aplikačnej vrstvy podľa požadovaného formátu na prenos cez sieť.

Na samom vrchole zásobníka vrstiev referenčného modelu OSI je **aplikačná vrstva**, ktorá implementuje sieťové aplikácie. Tieto aplikácie produkujú dáta, ktoré musia prenášať cez sieť. Táto vrstva slúži aj ako okno pre aplikačné služby na prístup k sieti a na zobrazovanie prijatých informácií užívateľovi.

Internet Protocol (IP)

Internetový protokol (IP) slúži na odosielanie údajov z jedného zariadenia do druhého cez internet. Každé zariadenie má IP adresu, ktorá ho jednoznačne identifikuje a umožňuje mu komunikovať a vymieňať si údaje s inými zariadeniami pripojenými na internet [2].

IP je zodpovedný za definovanie spôsobu, akým si aplikácie a zariadenia vymieňajú dátové pakety. Je to hlavný komunikačný protokol zodpovedný za formáty a pravidlá na výmenu údajov a správ medzi počítačmi v jednej sieti alebo niekoľkých sieťach pripojených na Internet. Robí to cez Internet Protocol Suite (bežne známy ako TCP/IP), skupinu komunikačných protokolov, ktoré sú rozdelené do štyroch abstraktných vrstiev.

IP je hlavný protokol v rámci internetovej vrstvy TCP/IP aj UDP/IP. Jeho hlavným účelom je doručovať dátové pakety medzi zdrojovou aplikáciou alebo zariadením a cieľom pomocou metód a štruktúr, ktoré do dátových paketov umiestňujú značky, ako napríklad informácie o adrese [3].

IP a jeho pridružené smerovacie protokoly sú možno najvýznamnejšie z celej sady TCP/IP. IP je zodpovedný za:

- **IP adresovanie** - Konvencie IP adresovania sú súčasťou protokolu IP.
- **Komunikácia medzi hostiteľmi** - IP určuje cestu, ktorou musí paket prejsť, na základe IP adresy prijímajúceho hostiteľa.
- **Formátovanie paketov** – IP zhromažďuje pakety do jednotiek známych ako IP datagramy.
- **Fragmentácia** – ak je paket príliš veľký na prenos cez sieťové médium, IP na odosielaťcom hostiteľovi rozdelí paket na menšie časti. IP na prijímajúcom hostiteľovi potom rekonštruje fragmenty na pôvodný paket.

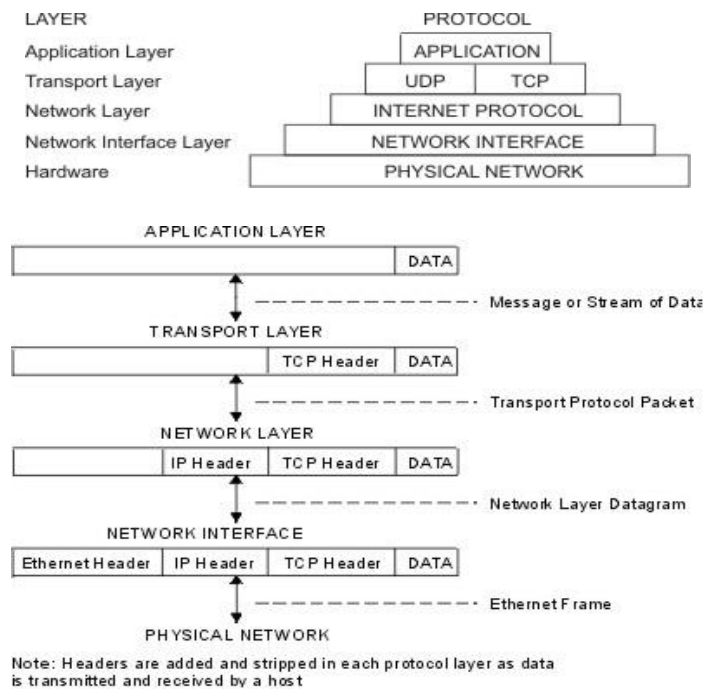
IPv4 (Internet Protocol version 4) je štvrtá verzia internetového protokolu (IP). Prvá široko používaná verzia. Kvôli rýchlemu rastu internetu však bolo potrebné vytvoriť nový internetový protokol s vylepšenými schopnosťami, ako je napríklad väčší adresný priestor. Táto nová verzia, známa ako verzia 6, je označená ako **IPv6**. Aby sa predišlo nejasnostiam pri adresovaní internetového protokolu, používa sa nasledujúca konvencia:

- Keď sa v popise používa výraz IP, popis sa vzťahuje na IPv4 aj IPv6.
- Keď sa v popise používa výraz IPv4, popis sa vzťahuje iba na IPv4.
- Keď sa v popise používa výraz IPv6, popis sa vzťahuje iba na IPv6.

IP pripája Header IP k hlavičke segmentu alebo paketu okrem informácií pridaných protokolom TCP alebo UDP. Informácie v hlavičke IP zahŕňajú IP adresy odosielaťceho a prijímajúceho hostiteľa, dĺžku a poradie datagramov. Tieto informácie sa poskytujú v prípade, že datagram presahuje povolenú veľkosť bajtov pre sieťové pakety a musí byť fragmentovaný.

2.1. Transmission Control Protocol

Transmission Control Protocol (TCP) je určený na použitie ako vysoko spoľahlivý protokol medzi zariadeniami v počítačových komunikačných sieťach s prepínaním paketov a vo vzájomne prepojených systémoch takýchto sietí [3]. Obrázok 2 nižšie popisuje vrstvomý model protokolu TCP/IP:



Obrázok 2 Referenčný model TCP/IP

Konektivita

TCP/IP starostlivo definuje, ako sa informácie presúvajú od odosielateľa k príjemcovi. Po prvé, aplikačné programy odosiľajú správy alebo toky údajov pomocou protokolov Internet Transport Layer Protocol [4]. Protokol prijme údaje z aplikácie, rozdelí ich na menšie časti nazývané pakety, pridá cieľovú adresu, a potom pakety odovzdá ďalšej vrstve protokolu, vrstve internetovej siete (IP). Primárnym účelom TCP je teda poskytovať spoľahlivý, zabezpečený a logický okruh alebo službu pripojenia medzi procesmi.

TCP sa musí obnoviť z údajov, ktoré sú poškodené, stratené, duplikované alebo doručené v ľubovoľnom poradí internetovým komunikačným systémom. To sa dosiahne priradením poradového čísla každému vysielanému oktetu a vyžadovaním pozitívneho potvrdenia (**ACK**) od prijímajúceho TCP. Ak **ACK** nie je prijaté v rámci časového limitu, dáta sa prenesú znova. V prijímači sa poradové čísla používajú na správne zoradenie segmentov, ktoré môžu byť prijaté mimo poradia, a na odstránenie duplikátov. Poškodenie sa rieši pridaním kontrolného súčtu ku každému vysielanému segmentu a jeho kontrolou v prijímači a vyradením poškodených segmentov.

Pokiaľ budú TCP naďalej správne fungovať a internetový systém nebude úplne rozdelený, žiadne chyby prenosu neovplyvnia správne doručenie údajov. TCP sa zotavuje z chýb systému internetovej komunikácie [4].

Spracovanie dátovej komunikácie

Keď používateľ vydá príkaz, ktorý používa protokol aplikačnej vrstvy TCP/IP, spustí sa reťazec udalostí. Príkaz alebo správa používateľa prechádza cez zásobník protokolov TCP/IP na lokálnom počítači a potom cez sieťové médium protokolov na príjemcovi. Protokoly na každej vrstve odosielaťujúceho hostiteľa pridávajú informácie k pôvodným údajom [5].

Balík je základná jednotka informácií prenášaných cez sieť, ktorá pozostáva aspoň z hlavičky s adresami odosielaťujúceho a prijímajúceho hostiteľa, tela s údajmi, ktoré sa majú preniesť. Keď paket prechádza protokolom TCP/IP, protokoly na každej vrstve buď pridávajú alebo odoberajú polia zo základnej hlavičky. Keď protokol na odosielaťujúcom hostiteľovi pridáva údaje do hlavičky paketu, proces sa nazýva „zapuzdrenie údajov“ (data encapsulation).

História paketu začína, keď používateľ na jednom hostiteľovi odošle správu alebo vydá príkaz, ktorý musí získať prístup k vzdialenému hostiteľovi. Aplikačný protokol spojený s príkazom alebo správou formátuje paket tak, aby ho mohol spracovať príslušný protokol transportnej vrstvy, TCP alebo UDP.

Formát hlavičky (Header format)

TCP segmenty sa odosielaťujú ako internetové datagramy. Hlavička internetového protokolu obsahuje niekoľko informačných polí vrátane zdrojovej a cieľovej adresy hostiteľa. Hlavička TCP nasleduje po hlavičke **IP** a poskytuje informácie špecifické pre protokol TCP. Toto rozdelenie umožňuje existenciu protokolov na hostiteľskej úrovni iných ako TCP [6].

Source Port		Destination Port	
Sequence Number			
Acknowledgment Number			
Data Offset	Reserved	Control Bits	Window
Checksum		Urgent Pointer	
Options		Padding	
Data			

Tabuľka 1 TCP Formát hlavičky

Popis jednotlivých pojmov, ktoré sú v tabuľke 1:

- **Source Port:** 16 bitov. Číslo zdrojového portu.
- **Destination Port:** 16 bitov. Číslo cieľového portu.

- **Sequence Number:** 32 bitov. Poradové číslo prvého dátového oktetu v tomto segmente (okrem prípadu, keď je prítomný SYN). Ak je prítomný SYN, poradové číslo je počiatočné poradové číslo (ISN) a prvý dátový oktet je ISN+1.
- **Acknowledgment Number:** 32 bitov. Ak je nastavený riadiaci bit ACK, toto pole obsahuje hodnotu nasledujúceho poradového čísla, ktoré odosielateľ segmentu očakáva, že prijme. Po vytvorení spojenia sa vždy odošle.
- **Data Offset:** 4 bity. Počet 32-bitových slov v hlavičke TCP. Označuje, kde začínajú dáta. TCP hlavička (dokonca aj jedna vrátane možností) je celé číslo s dĺžkou 32 bitov.
- **Reserved:** 6 bitov. Vyhradené pre budúce použitie. Musí byť nula.
- **Control Bits:** 6 bitov:
 - **URG:** Urgentné pole ukazovateľa je významné
 - **ACK:** Pole potvrdenia je významné
 - **PSH:** Funkcia Push
 - **RST:** Obnoviť pripojenie
 - **SYN:** Synchronizácia sekvenčných čísel
 - **FIN:** Žiadne ďalšie údaje od odosielateľa
- **Window:** 16 bitov. Počet dátových oktetov začínajúci tým, ktorý je uvedený v poli potvrdenia, ktorý je odosielateľ tohto segmentu ochotný akceptovať. Štandardne sa veľkosť okna meria v bajtoch, takže je obmedzená na 2^{16} (65535) bajtov. Vďaka možnosti mierky okna TCP je však možné túto veľkosť zväčšiť až na 1 GB. Ak chceme povoliť túto možnosť, obe strany sa na tom musia dohodnúť vo svojich segmentoch SYN.
- **Checksum:** 16 bitov. Pole 16-bitového kontrolného súčtu sa používa na kontrolu chýb hlavičky TCP, užitočného zaťaženia a pseudohlavičky IP. Ak segment obsahuje nepárny počet hlavičiek a textových oktetov, ktoré sa majú sčítať, posledný oktet sa doplní vpravo nulami, aby sa vytvorilo 16-bitové slovo na účely kontrolného súčtu. Pri výpočte kontrolného súčtu je samotné pole kontrolného súčtu nahradené nulami. Kontrolný súčet pokrýva aj 96-bitovú pseudo hlavičku koncepčne predponovanú hlavičke TCP. Táto pseudo hlavička obsahuje zdrojovú adresu (**Source Address**), cieľovú adresu (**Destination Address**), protokol a dĺžku TCP (**TCP length**). To poskytuje TCP ochranu proti chybné smerovaným segmentom. Tieto informácie sú prenášané v internetovom protokole a sú prenášané cez TCP/sieťové rozhranie v argumentoch alebo výsledkoch volaní TCP na IP.
- **Urgent Pointer:** 16 bitov. Toto pole oznamuje aktuálnu hodnotu urgentného ukazovateľa ako kladnú odchýlku od poradového čísla v tomto segmente. Ukazovateľ naliehavosti

ukazuje na poradové číslo oktetu za naliehavými údajmi. Toto pole sa interpretuje iba v segmentoch s nastaveným riadiacim bitom **URG (Urgent)**.

- **Options:** variabilné. V niektorých prípadoch sa môže použiť na rozšírenie protokolu. Niekedy sa používa na testovanie.
- **Padding:** variabilné. Výplň hlavičky TCP sa používa na zabezpečenie toho, že hlavička TCP končí a údaje začínajú na 32 bitovej hranici. Výplň je zložená z núl.
- **Data:** variabilné. Údaje.

Protokolový mechanizmus

Na rozdiel od tradičnej alternatívy UDP, ktorá môže začať vysielat' pakety okamžite, TCP vytvára spojenia, ktoré musia byť vytvorené pred prenosom údajov. TCP spojenie možno rozdeliť do 3 fáz [7]:

- Nadviazanie spojenia
- Prenos dát
- Ukončenie spojenia

Pripojenie TCP je riadené operačným systémom prostredníctvom prostriedku, ktorý predstavuje lokálny koncový bod pre komunikáciu, internetový soket. Počas životnosti pripojenia TCP prejde lokálny koncový bod sériou zmien stavu:

Stav	Konečný bod	Popis
LISTEN	Server	Čaká sa na požiadavku na pripojenie z ľubovoľného vzdialeného koncového bodu TCP
SYN-SENT	Klient	Čaká sa na zodpovedajúcu žiadosť o pripojenie po odoslaní žiadosti o pripojenie
SYN-RECEIVED	Server	Čaká sa na potvrdenie žiadosti o pripojenie po prijatí a odoslaní žiadosti o pripojenie
ESTABLISHED	Server a Klient	Otvorené pripojenie, prijaté údaje môžu byť doručené používateľovi. Normálny stav pre fázu prenosu dát pripojenia

FIN-WAIT-1	Server a Klient	Čaká sa na žiadosť o ukončenie pripojenia zo vzdialeného protokolu TCP alebo na potvrdenie predtým odoslanej žiadosti o ukončenie pripojenia
FIN-WAIT-2	Server a Klient	Čaká sa na žiadosť o ukončenie pripojenia zo vzdialeného TCP
CLOSE-WAIT	Server a Klient	Čaká sa na žiadosť o ukončenie pripojenia od miestneho používateľa
CLOSING	Server a Klient	Čaká sa na potvrdenie žiadosti o ukončenie pripojenia zo vzdialeného TCP
LAST-ACK	Server a Klient	Čaká sa na potvrdenie žiadosti o ukončenie pripojenia, ktorá bola predtým odoslaná vzdialenému TCP (čo zahŕňa potvrdenie jeho žiadosti o ukončenie pripojenia)
TIME-WAIT	Server alebo Klient	Čaká sa, kým uplynie dostatok času, aby ste sa uistili, že platnosť všetkých zostávajúcich balíkov na pripojení vypršala
CLOSED	Server a Klient	Žiadny stav pripojenia.

Tabuľka 2 Stavy pripojenia TCP

Nadviazanie spojenia

Predtým, ako sa klient pokúsi pripojiť k serveru, server sa musí najprv pripojiť k portu a počúvať ho, aby ho otvoril pre pripojenia: toto sa nazýva „pasívne otvorené“ („passive open“). Po vytvorení pasívneho otvorenia môže klient vytvoriť spojenie iniciovaním aktívneho otvorenia pomocou trojcestného (alebo 3-krokového) „**handshake**“:

1. **SYN:** Aktívne otvorenie sa vykoná tak, že klient odošle SYN serveru. Klient nastaví poradové číslo segmentu na náhodnú hodnotu A.
2. **SYN-ACK:** Ako odpoveď server odpovie SYN-ACK. Číslo potvrdenia je nastavené o jedno viac ako prijaté poradové číslo, t.j. A+1, a poradové číslo, ktoré server vyberie pre paket, je iné náhodné číslo, B.

3. **ACK:** Nakoniec klient pošle ACK späť na server. Poradové číslo je nastavené na prijatú hodnotu potvrdenia, t.j. A+1, a číslo potvrdenia je nastavené o jedno viac ako prijaté poradové číslo, t.j. B+1.

Kroky **1** a **2** stanovujú a potvrdzujú poradové číslo pre smer klient-server. Kroky **2** a **3** stanovujú a potvrdzujú poradové číslo pre smer server-klient. Po dokončení týchto krokov klient aj server dostali potvrdenia a nadviazala sa plne duplexná komunikácia.

Zraniteľnosť/Nevýhoda

Odmietnutie služby (Denial of service). Použitím sfalšovanej IP adresy a opakovaným odosielaním účelovo zostavených paketov SYN, po ktorých nasleduje veľa paketov ACK, môžu útočníci spôsobiť, že server spotrebuje veľké množstvo zdrojov pri sledovaní falošných spojení. Toto je známe ako povodňový útok SYN. Navrhované riešenia tohto problému zahŕňajú súbory cookie SYN a kryptografické hádanky, hoci súbory cookie SYN majú svoj vlastný súbor zraniteľností. Sockstress je podobný útok, ktorý možno zmierniť správou systémových zdrojov.

Únos pripojenia (Connection hijacking). Útočník, ktorý je schopný odpočúvať TCP reláciu a presmerovať pakety, môže uniesť TCP spojenie. Za týmto účelom sa útočník dozvie poradové číslo z prebiehajúcej komunikácie a vytvorí falošný segment, ktorý vyzerá ako ďalší segment v streame. Takýto jednoduchý únos môže viesť k tomu, že jeden paket bude chybné prijatý na jednom konci. Keď prijímajúci hostiteľ potvrdí ďalší segment na druhej strane pripojenia, synchronizácia sa stratí.

2.2. User Datagram Protocol

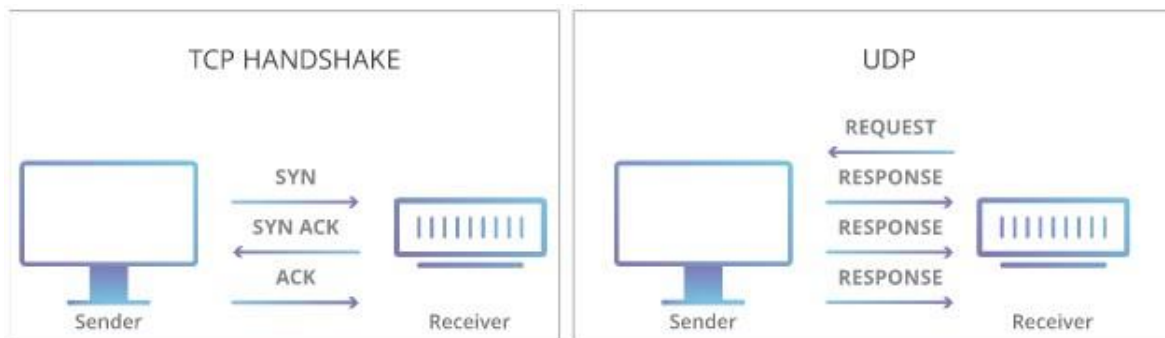
Tento User Datagram Protocol (UDP) je definovaný tak, aby sprístupnil datagramový režim počítačovej komunikácie s prepínaním paketov v prostredí vzájomne prepojenej množiny počítačových sietí. Tento protokol predpokladá, že ako základný protokol sa používa internetový protokol (IP) [8].

Tento protokol poskytuje aplikačným programom procedúru na odosielanie správ iným programom s minimálnym protokolovým mechanizmom. Protokol je orientovaný na transakciu a nie je zaručené doručenie a ochrana proti duplicite [9]. Aplikácie vyžadujúce spoľahlivé doručovanie tokov údajov by mali používať protokol TCP (Transmission Control Protocol).

Konektivita v porovnaní s TCP

UDP je rýchlejší, ale menej spoľahlivý ako TCP. Pri TCP komunikácii začínajú dva počítače vytvorením spojenia prostredníctvom automatizovaného procesu nazývaného „handshake“. Až po dokončení tohto handshake jeden počítač skutočne prenesie dátové pakety do druhého.

Komunikácia UDP týmto procesom neprechádza [9]. Namiesto toho môže jeden počítač jednoducho začať odosielať údaje do druhého:



Obrázok 3 TCP vs UDP konektivita

Tieto rozdiely vytvárajú určité výhody. Pretože protokol UDP nevyžaduje „handshake“ ani kontrolu, či údaje prichádzajú správne, dokáže prenášať údaje oveľa rýchlejšie ako TCP.

Táto rýchlosť však vytvára kompromisy. Ak sa UDP datagram stratí počas prenosu, nebude znova odoslaný. V dôsledku toho musia byť aplikácie, ktoré používajú UDP, schopné tolerovať chyby, straty a duplikáty.

Technicky je takáto strata paketov menšou chybou v UDP ako dôsledkom toho, ako je vybudovaný internet. Väčšina sieťových smerovačov nevykonáva usporiadanie balíkov a potvrdenie príchodu podľa návrhu, pretože by to vyžadovalo nerealizovateľné množstvo dodatočnej pamäte. TCP je spôsob, ako vyplniť túto medzeru, keď si to aplikácia vyžaduje.

Spracovanie komunikácie

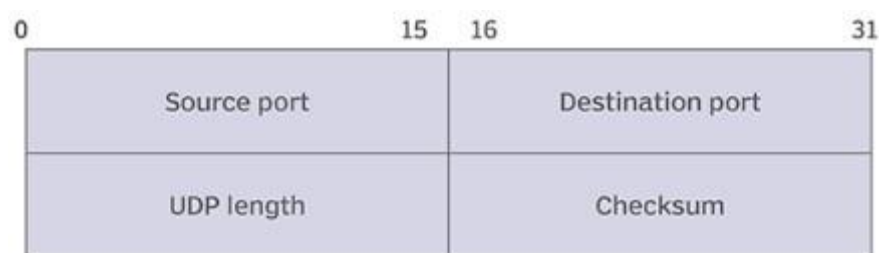
UDP používa IP na získanie datagramu z jedného počítača do druhého. UDP funguje tak, že zhromažďuje údaje v pakete UDP a pridáva do paketu svoju vlastnú hlavičku. Tieto údaje pozostávajú zo zdrojových a cieľových portov, na ktorých sa má komunikovať, dĺžky paketu a kontrolného súčtu. Potom, čo sú pakety UDP zapuzdrené do paketu IP, sú odoslané na miesto určenia.

Na rozdiel od TCP, UDP nezaručuje, že sa pakety dostanú do správnych cieľov. To znamená, že UDP sa nepripája priamo k prijímaciemu počítaču. Namiesto toho odosiela údaje a

spolieha sa na zariadenia medzi odosielajúcim a prijímajúcim počítačom, aby správne dostali údaje tam, kde majú ísť. Ak aplikácia nedostane odpoveď v určitom časovom rámci, odošle paket znova alebo sa prestane pokúšať.

Formát hlavičky (Header format)

UDP používa hlavičky pri balení údajov správ na prenos cez sieťové pripojenia. Hlavičky UDP obsahujú súbor parametrov nazývaných „polia“ definované technickými špecifikáciami protokolu [10].



Obrázok 4 UDP hlavička [10]

Na Obrázok 4 sú:

- **Source Port** je voliteľné pole, ak má význam, označuje port procesu odosielania a možno ho považovať za port, na ktorý by mala byť adresovaná odpoveď, ak neexistujú žiadne ďalšie informácie. Ak sa nepoužije, vloží sa hodnota nula.
- **Destination Port** má význam v kontexte konkrétnej internetovej cieľovej adresy.
- **UDP Length** je dĺžka v oktetoch tohto užívateľského datagramu vrátane tejto hlavičky a údajov. To znamená, že minimálna hodnota dĺžky je osem.
- **Checksum** je 16-bitový doplnok k súčtu pseudohlavičky informácií z hlavičky IP, hlavičky UDP a údajov, na konci doplnený nulovými oktetmi (ak je to potrebné), aby sa vytvoril násobok dvoch oktetov.

Nevýhody UDP

„Riziká“ UDP, ako je **strata paketov**, nie sú vo väčšine prípadov vážnym problémom. UDP však možno zneužiť na škodlivé účely. Keďže protokol UDP nevyžaduje podanie ruky, útočníci môžu „zaplaviť“ cieľový server prevádzkou UDP bez toho, aby najprv získali povolenie tohto servera na začatie komunikácie.

Typický UDP záplavový útok posiela veľké množstvo UDP datagramov na náhodné porty na cieľovom počítači. To núti cieľový počítač odpovedať rovnako veľkým počtom paketov, čo naznačuje, že tieto porty boli nedostupné. Výpočtové zdroje potrebné na reakciu na každý

podvodný datagram môžu vyčerpať cieľový počítač, čo vedie k odmietnutiu služby legítimnej prevádzke [11].

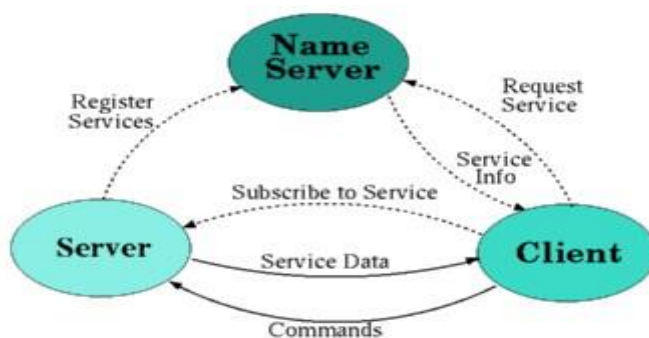
- UDP nemá žiadne okno a žiadnu funkciu na zabezpečenie prijímania údajov v rovnakom poradí, v akom boli odoslané
- UDP nepoužíva žiadnu kontrolu chýb, ale deteguje chybu v prijatom pakete
- Smerovač môže byť pri UDP neopatrný. Po kolízii znovu neposielajú UDP datagram a často zahodia UDP pakety pred TCP paketmi
- Neexistuje žiadna kontrola toku a žiadne potvrdenie prijatých údajov
- Opravou chýb sa zaoberá iba aplikačná vrstva. Aplikácie sa teda môžu jednoducho obrátiť na používateľa, aby správu odoslal znova
- UDP môže mať veľký počet používateľov, ale prenos veľkého množstva údajov cez UDP môže spôsobiť preťaženie a nikto s tým nič neurobí, takže nemá žiadnu kontrolu preťaženia
- UDP nemá riadenie toku, riadenie preťaženia. Implementácia je povinnosťou užívateľského programu

2.3. Distributed Information Management

V mnohých aplikáciách je dôležitá potreba transparentným spôsobom sprístupniť informácie veľkému množstvu iných procesov. DIM je veľmi výkonná alternatíva k týmto systémom. Poskytuje pomenovaný priestor pre procesy na zverejňovanie informácií (**Vydavatelía**) a veľmi jednoduché API pre procesy, ktoré sú ochotné tieto informácie využívať (**Odberatelía**). Plne zvláda obnovu chýb na úrovni Vydavatelía a Odberatelía (**Publishers and Subscribers**), bez dodatočného softvéru v aplikácii.

DIM je prenosný balík s malým množstvom odoslaných dát na publikovanie informácií, prenos dát a medziprocesovú komunikáciu. Ako väčšina komunikačných systémov je založený na paradigme klient/server [12].

Základným pojmom v prístupe DIM je pojem „služba“. Servery poskytujú služby klientom. Služba je zvyčajne súbor údajov (akéhokoľvek typu alebo veľkosti) a je rozpoznaná podľa názvu - "pomenované služby". Menný priestor pre služby je voľný.

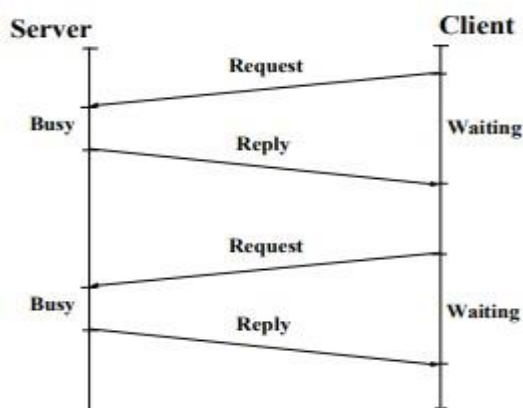


Obrázok 5 DIM diagram [12]

Mechanizmus komunikácie RPC

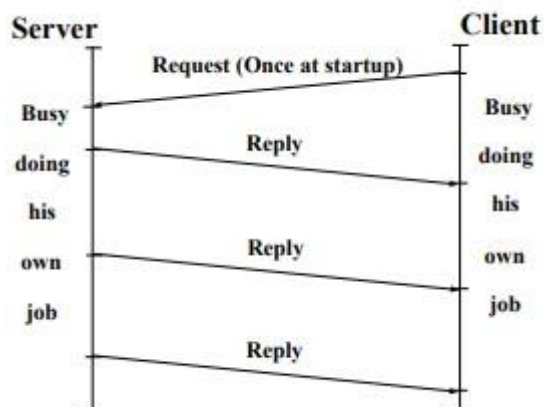
Pri navrhovaní komunikačného systému je dôležitou otázkou výber komunikačného mechanizmu, ktorý sa má použiť. Distribuované aplikácie sú často založené na vzdialených volaniach procedúr (RPC). V mechanizme RPC klient odošle správu obsahujúcu názov rutiny, ktorá sa má vykonať, a jej parametre na server, server vykoná rutinu a odošle správu obsahujúcu výsledok. To znamená, že komunikácia je medzi dvoma bodmi asynchrónna, pretože klient vždy čaká, kým sa rutina neskončí.

RPC mechanizmus veľmi ťažký a nevhodný pre online aplikácie v reálnom čase. Vo väčšine týchto aplikácií sú dôležitými úlohami monitorovanie parametrov (v pravidelných časových intervaloch) a asynchrónna reakcia na meniace sa podmienky; okrem toho sa tieto úlohy bežne opakujú a musia sa vykonávať na neznáme časové obdobie.



Obrázok 6 RCP mechanizmus

Ako najlepšie sa v tomto prípade javilo, že klienti deklarujú záujem o službu poskytovanú serverom iba raz (pri spustení) a následne získavajú aktualizácie v pravidelných časových intervaloch alebo pri zmene podmienok. T.j. protokol umožňujúci asynchrónnu komunikáciu typu jeden k mnohým.



Obrázok 7 DIM mechanizmus

Ale v tejto práci použijem mechanizmus DIM RPC na vytvorenie všetkých protokolov v rovnakých situáciách. Na to použijem oficiálne príručky pre Server [13] a pre Klient [14].

Implementácia DIM Server

Servery DIM sú procesy, ktoré majú poskytovať informácie [15]. Aby sa proces stal serverom, musí deklarovať všetky služby, ktoré môže poskytnúť, a všetky príkazy, ktoré je ochotný prijať a potom odoslať tieto informácie na menný server.

Služba sa deklaruje uvedením názvu služby, adresy a veľkosti údajov, ktoré tvoria túto službu, a formátu údajov (s cieľom konvertovať medzi rôznymi platformami).

V C++ trieda DimService implementuje obsluhu služieb. Formát a veľkosť sú implicitné pre základné typy údajov (celé čísla, čísla s pohyblivou desatinnou čiarkou, reťazce atď.), pre zložité typy údajov, ako sú štruktúry, musí používateľ poskytnúť popis formátu a veľkosti.

Príkazy sú deklarované zadaním názvu príkazu, formátu príkazu a rutiny spätného volania, ktorá sa má zavolať pri prijatí príkazu. V C++ trieda DimCommand implementuje spracovanie príkazov pomocou virtuálnej metódy commandHandler (rpcHandler v našej situácii).

Po deklarovaní všetkých poskytnutých služieb a príkazov je potrebné zavolať inú metódu, aby bolo možné začať obsluhovať požiadavky klientov. Táto statická metóda triedy DimServer odošle úplný zoznam služieb a príkazov na menný server a nastaví všetky potrebné mechanizmy, aby boli požiadavky klientov vybavované transparentne. Odvtedy sa server môže vrátiť k svojej normálnej úlohe, všetky požiadavky klientov a aktualizácie služieb budú automaticky spracované systémom DIM [16].

Implementácia DIM Client

Klienti DIM sú procesy, ktoré potrebujú dostupné informácie na splnenie svojich úloh, ktorými sú zobrazovanie, monitorovanie alebo spracovanie. Proces sa stáva klientom zadáním názvu služby, o ktorú má záujem, a voliteľného intervalu aktualizácie. Klient môže tiež špecifikovať konštantu, ktorá sa má použiť ako údaje služby vždy, keď je služba dostupná alebo nedostupná.

Klient môže pokračovať vo svojej práci, všetky servisné dáta sa automaticky uložia a/alebo sa vykoná spätné volanie pri každom prijatí služby. Proces môže kedykoľvek odoslať príkaz na server zadáním názvu príkazu a údajov príkazu.

Interné kroky potrebné na to, aby ste sa stali klientom, buď vyžiadáním služby alebo odoslaním príkazu, sú nasledovné: najprv sa klient spýta **menného servera (Name Server)**, kde je služba dostupná, Menný Server odpovie adresu servera, klient kontaktuje potom priamo server a sformuluje svoju požiadavku (vrátane typu aktualizácie). Server sa potom v prípade potreby postará o aktualizáciu služby. Ak služba nie je dostupná, Menný Server odpovie záporne, konštantné údaje sa vložia do cesty údajov služby a spustí sa aktualizácia, ale požiadavka zostane vo fronte na mennom serveri, kedykoľvek bude služba dostupná, klient bude upozornený a spojenie nadviazané.

DNS

Ciele návrhu DNS ovplyvňujú jeho štruktúru [17]:

- Primárnym cieľom je konzistentný menný priestor, ktorý sa bude používať na odkazovanie na zdroje. Aby sa predišlo problémom spôsobeným ad hoc kódovaním, nemalo by sa vyžadovať, aby názvy obsahovali sieťové identifikátory, adresy, trasy alebo podobné informácie ako súčasť názvu.
- Samotná veľkosť databázy a frekvencia aktualizácií naznačujú, že musí byť udržiavaná distribuovaným spôsobom s lokálnym ukladaním do vyrovnávacej pamäte na zlepšenie výkonu. Prístupy, ktoré sa pokúšajú zhromaždiť konzistentnú kópiu celej databázy, budú čoraz drahšie a ťažšie, a preto by sa im malo vyhýbať. Rovnaký princíp platí pre štruktúru menného priestoru a najmä mechanizmy vytvárania a odstraňovania mien; tieto by sa tiež mali distribuovať.
- Ak existujú kompromisy medzi nákladmi na získavanie údajov, rýchlosťou aktualizácií a presnosťou vyrovnávacích pamätí, zdroj údajov by mal kontrolovať kompromis.

- Náklady na implementáciu takéhoto zariadenia vyžadujú, aby bol všeobecne užitočný a neobmedzoval sa na jednu aplikáciu. Mali by sme byť schopní používať mená na získavanie adries hostiteľov, údajov poštových schránok a ďalších, zatiaľ neurčených informácií. Všetky údaje spojené s názvom sú označené typom a dotazy môžu byť obmedzené na jeden typ.
- Pretože chceme, aby bol priestor názvov užitočný v odlišných sieťach a aplikáciách, poskytujeme možnosť používať rovnaký priestor názvov s rôznymi rodinami protokolov alebo manažmentom. Napríklad formáty adresy hostiteľa sa medzi protokolmi líšia, hoci všetky protokoly majú pojem adresy. DNS označuje všetky dáta triedou aj typom, aby sme mohli umožniť paralelné používanie rôznych formátov pre dáta typu adresa.
- Chceme, aby transakcie menného servera boli nezávislé od komunikačného systému, ktorý ich prenáša. Niektoré systémy môžu chcieť použiť datagramy na dotazy a odpovede, vytvoriť virtuálne okruhy len pre transakcie, ktoré vyžadujú spoľahlivosť (napr. aktualizácie databázy, dlhé transakcie); ostatné systémy budú používať výlučne virtuálne okruhy.
- Systém by mal byť užitočný v širokom spektre hostiteľských schopností. Osobné počítače aj veľkí časovo zdieľaní hostelia by mali byť schopní používať systém, aj keď možno rôznymi spôsobmi.

Organizácia doménového systému vychádza z niektorých predpokladov o potrebách a vzorcoch používania jeho používateľskej komunity, je navrhnutá tak, aby sa vyhlo mnohým komplikovaným problémom, ktoré sa vyskytujú vo všeobecných databázových systémoch.

Veľkosť celkovej databázy bude spočiatku úmerná počtu hostiteľov používajúcich systém, ale nakoniec sa rozrastie tak, aby bola úmerná počtu používateľov na týchto hostiteľoch, keď sa do systému domény pridávajú poštové schránky a ďalšie informácie.

Väčšina údajov v systéme sa bude meniť veľmi pomaly (napr. väzby poštových schránok, adresy hostiteľov), ale systém by mal byť schopný sa vysporiadať s údajmi, ktoré sa menia rýchlejšie (rádovo v sekundách alebo minútach).

Klienti doménového systému by mali byť schopní identifikovať dôveryhodné menné servery, ktoré uprednostňujú, pred prijatím odporúčaní na menné servery mimo tejto "dôveryhodné" množiny.

Prístup k informáciám je dôležitejší ako okamžité aktualizácie alebo záruky konzistencie. Proces aktualizácie teda umožňuje, aby sa aktualizácie šírili cez používateľov systému domény, namiesto toho, aby zaručovali, že všetky kópie budú aktualizované súčasne.

Menné Server (Name Server)

Menné Server vedie aktuálny zoznam všetkých serverov a služieb v systéme, prijíma registračné správy zo serverov a servisné požiadavky od klientov. Servery posielajú aj validačný balík v pravidelných intervaloch, takže Menné Server môže mať istotu, že fungujú. Ak server zlyhá pri odosielaní validačného balíka, Menné Server označí jeho služby ako nedostupné a všetky požiadavky klientov na ne budú zaradené do fronty, kým nebudú opäť v dostupné.

Menné Server pošle signál ukončenia každému procesu, ktorý sa pokúsi zaregistrovať službu, ktorá je už zaregistrovaná. Služby musia byť jedinečné.

Ak Menné Server zanikne, všetky predtým vytvorené spojenia medzi servermi a klientmi budú naďalej fungovať. Keď sa to splní, všetky servery znova zaregistrujú všetky svoje služby (boli skúšané v náhodných intervaloch) a všetci klienti znova požiadajú o služby, na ktoré čakajú, a potom sa nadviažu všetky spojenia.

Menné Server sa môže spustiť na inom počítači, servery a klienti sa ho snažia nájsť v preddefinovanom zozname možných počítačov (zvyčajne premenná prostredia). Adresár služieb na mennom serveri je implementovaný ako tabuľka kľúč-hodnota (Hash Table).

Menné Server je tiež server DIM v tom zmysle, že poskytuje služby DIM. Zverejňuje ako služby DIM informácie týkajúce sa serverov a služieb dostupných v systéme, k týmto informáciám je možné pristupovať akýmkoľvek procesom. V C++ je možné na prístup k týmto informáciám použiť triedu DimBrowser.

3. Spojenie klient-server

Windows Sockets (Winsock) umožňuje programátorom vytvárať pokročilé internetové, intranetové a iné sieťové aplikácie na prenos aplikačných dát cez kábel, nezávisle od používaného sieťového protokolu.

Winsock definuje štandardné rozhranie poskytovateľa služieb (SPI) medzi aplikačným programovacím rozhraním (API), s jeho funkciami exportovanými z WS2_32.dll a zásobníkmi protokolov.

Windows Sockets považuje každú sadu sieťových protokolov za jedinečnú rodinu adries. Takže protokol IPv6 sa považuje za rodinu adries AF_INET6 a protokol IPv4 sa považuje za rodinu adries AF_INET. Protokoly IPv6 a IPv4 podporujú použitie rôznych vrstvených protokolov IP, ako sú TCP a UDP [18].

Winsock slúži ako tlmočník pre základné sieťové služby, ako sú požiadavky poslať alebo prijať. Tieto požiadavky sú veľmi všeobecné a **Winsock** funguje tak, že ich prekladá do požiadaviek špecifických pre aplikačný protokol na vykonanie požadovaných úloh.

V tejto práci používam knižnicu "**Winsock2**" na vytvorenie spojenia klient-server na protokoloch TCP a UDP, keďže je vhodný na prácu s Internetovým protokolom.

Pre protokol DIM som použil architektúru **RPC**, takže komunikácia prebieha rovnako ako pri TCP a UDP. Vytvorenie spojenia tohto typu umožňuje vstavaná knižnica "**dic.hxx**", ktorá má triedu **DimRpc**, ktorú použijeme na implementáciu spojenia klient-server.

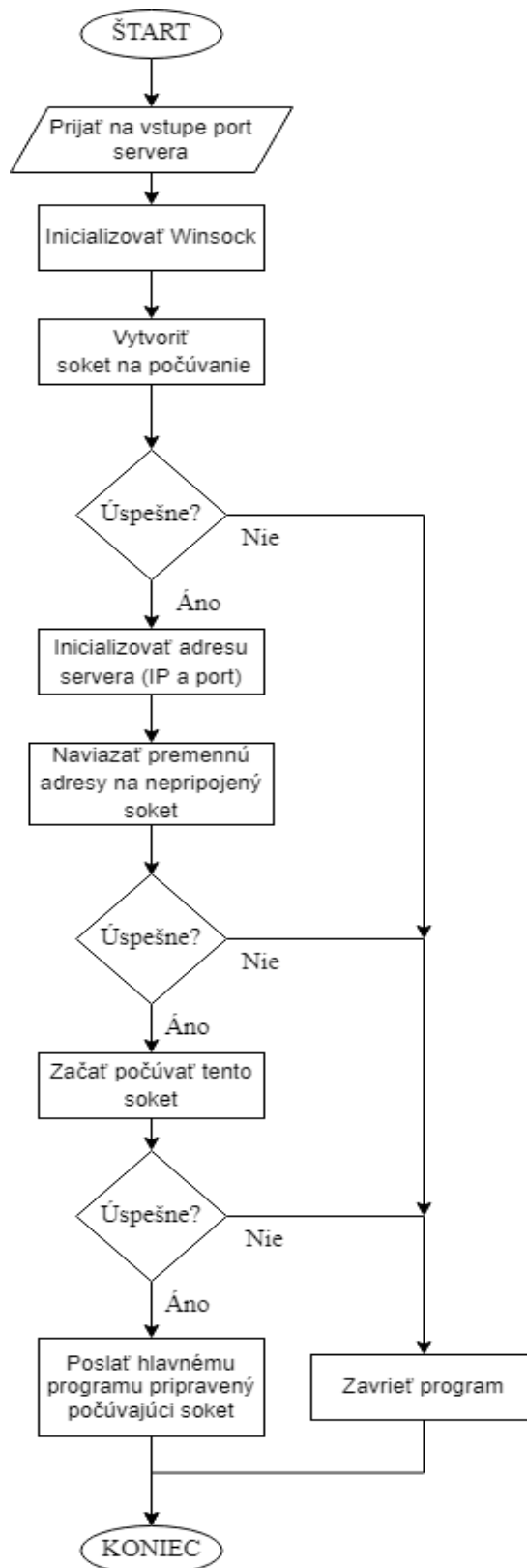
ServerLib a **ClientLib** sú knižnice, ktoré sme vytvorili, aby sme zmenšili množstvo hlavného kódu a zlepšili navigáciu. Budú obsahovať funkcie ako: "createListenSocket", "acceptClient" a "createClientSocket", ktoré sú potrebné na vytvorenie spojenia medzi klientom a serverom. V tejto kapitole ich popíšeme pomocou diagramov. A tiež "sendBuffSize", "show_info" a "receiveBuffSize", ktoré budú potrebné pre hlavnú slučku programu (popíšeme ich v kapitole 4. Softvérová implementácia). Tieto knižnice **neboli** použité pri implementácii protokolu DIM.

3.1. TCP/IP

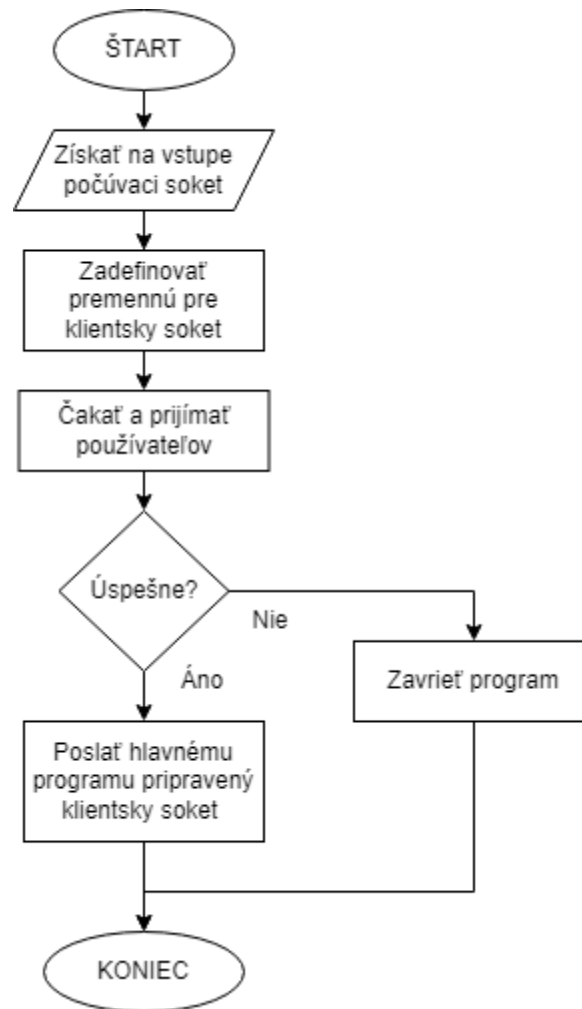
Štruktúra servera

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 pripojenia na

adresu servera, ktorá sa naň následne naviaže. Preto je tiež potrebné vytvoriť premennú, ktorá bude uchovávať údaje o adrese servera a porte. Potom pripojíme *listenSocket* k týmto hodnotám a začneme počúvať prichádzajúce požiadavky.

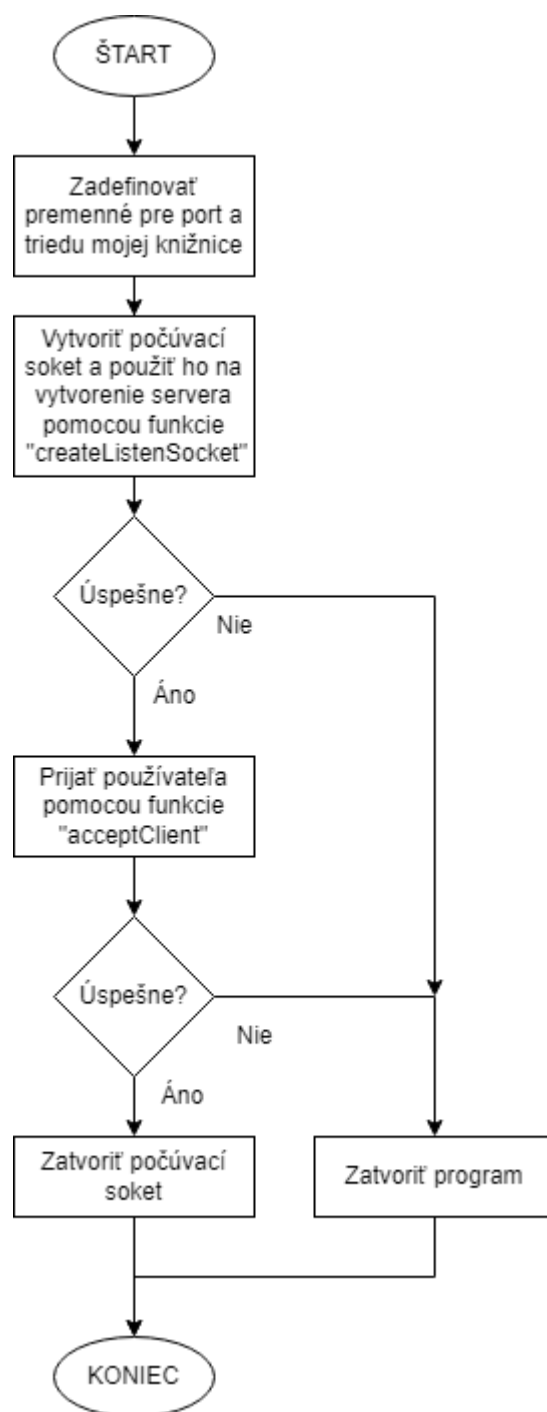


Obrázok 8 Vývojový diagram funkcie „createListenSocket“ TCP



Obrázok 9 Vývojový diagram funkcie „acceptClient“ TCP

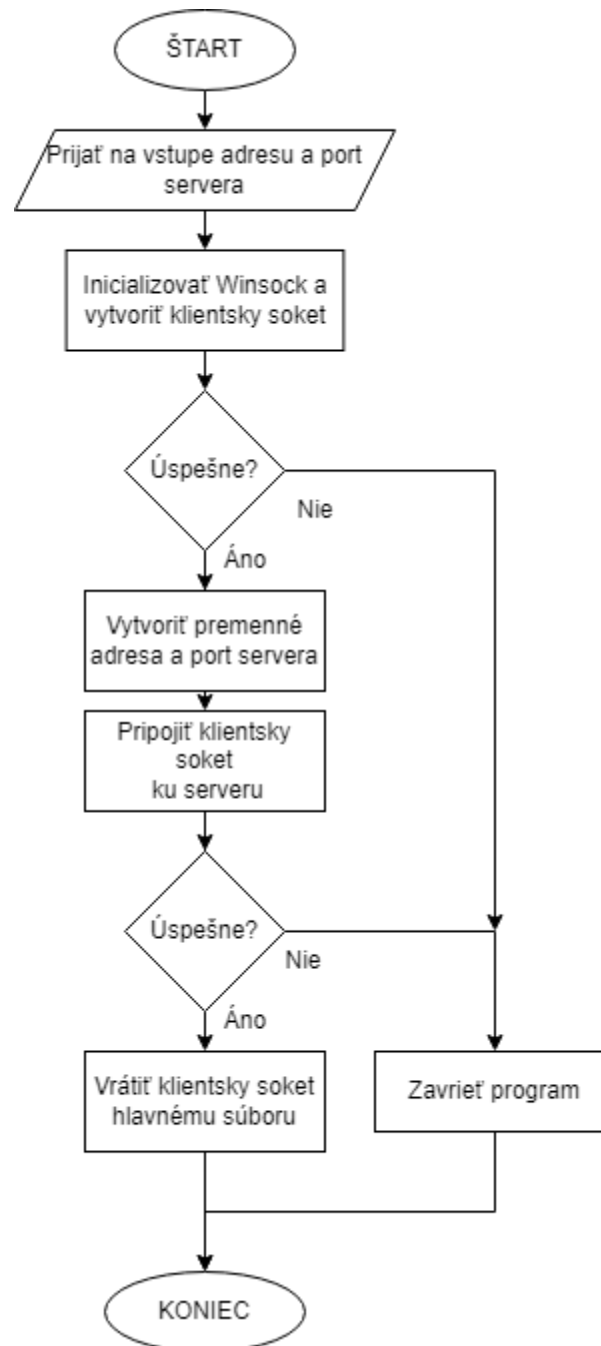
Funkcie uvedené na obrázkoch 8 a 9 sú v našej vlastnej knižnici. Sú zodpovedne za vytvorenie soketu pre klienta, počúvanie na tomto sokete a prijatie klienta. Musíme však tiež správne spustiť tieto funkcie v hlavnom programe:



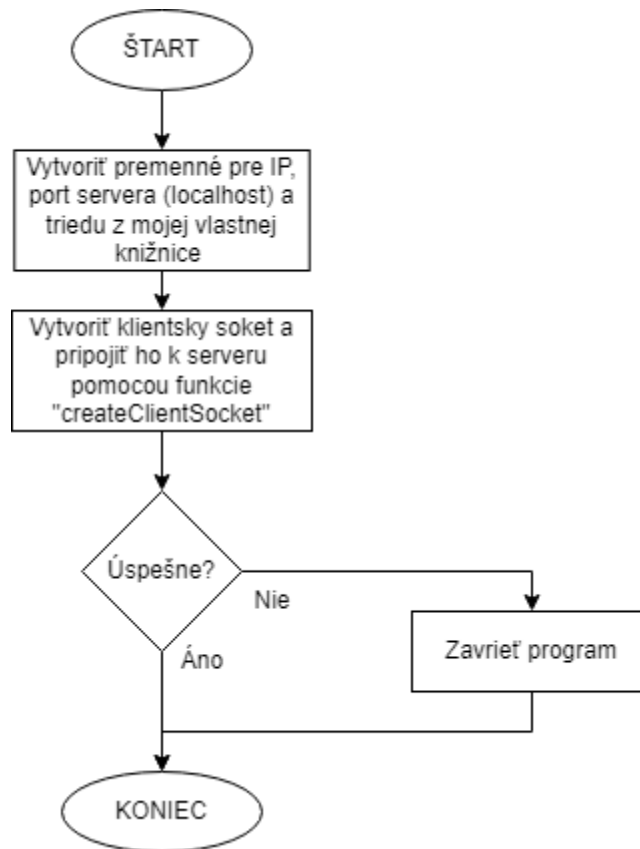
Obrázok 10 Vytvorenie servera a pripojenie klienta k nemu v hlavnom programe pomocou protokolu TCP

Štruktúra klienta

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



Obrázok 11 Vývojový diagram funkcie „createClientSocket“ TCP

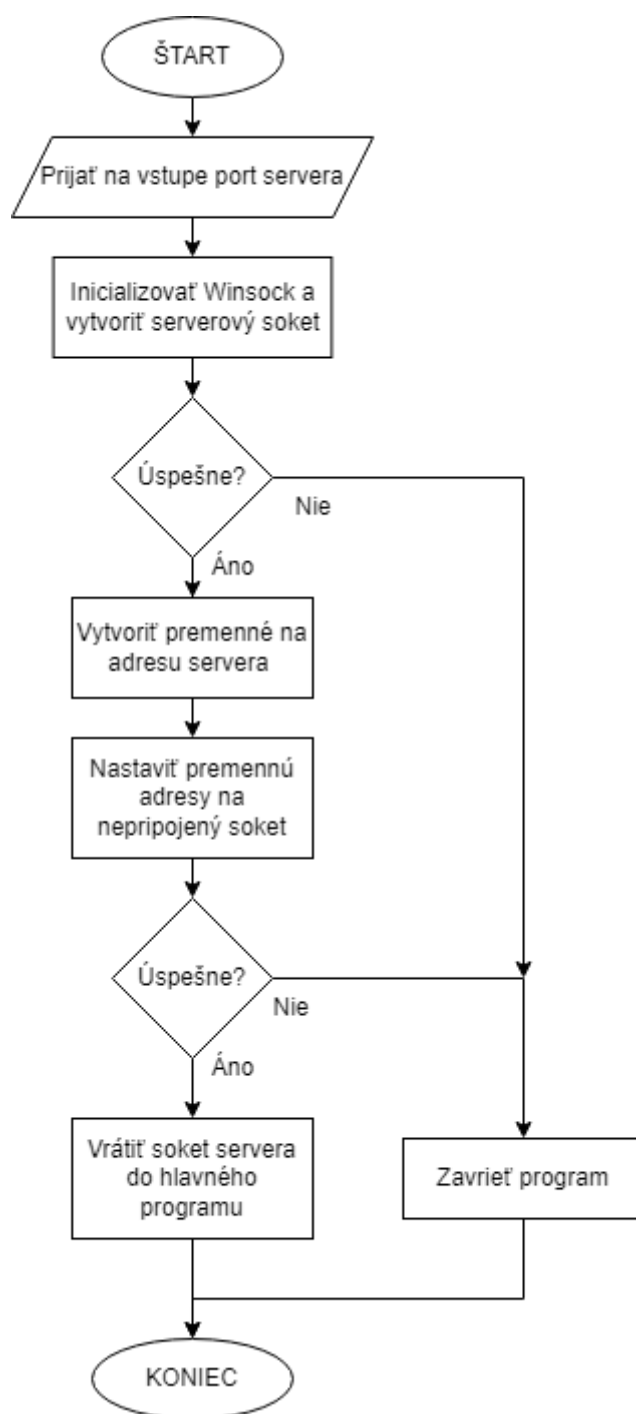


Obrázok 12 Vytvorenie klientskeho soketa a pripojenie ho ku serveru v hlavnom súbore pomocou protokolu TCP

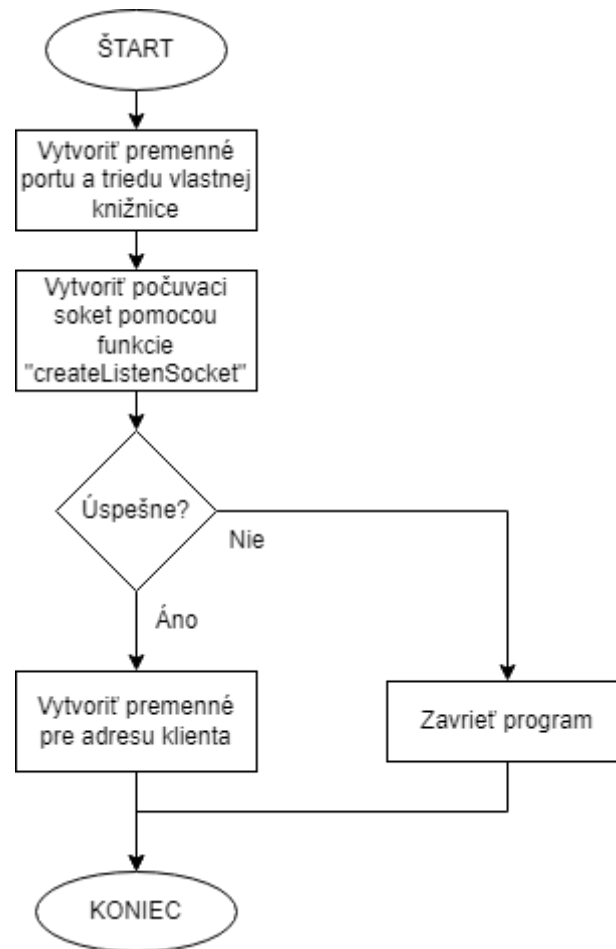
3.2. UDP/IP

Štruktúra servera

Spojenie server-klient s protokolom UDP sa robí podobným spôsobom ako TCP. WinsockAPI sa inicializuje na serveri a na klientovi, vytvorí sa serverový soket pre server a klientsky soket pre klienta, zapíše sa adresa servera a port, potom server pripojí vytvorenú adresu k svojmu soketu a klient sa potom pripojí. Na rozdiel od TCP, server nemusí dostať pokyn, aby počúval, UDP ho má v režime vždy zapnutý.



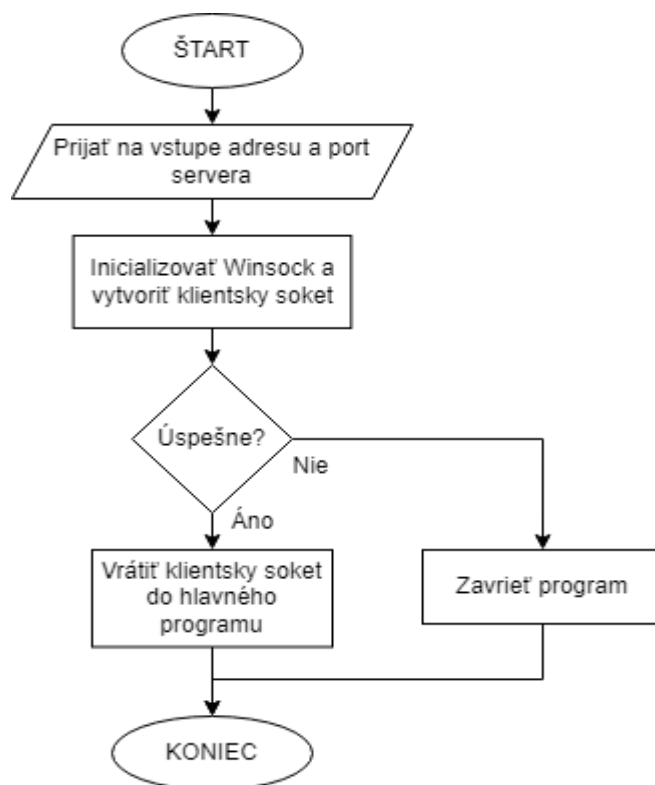
Obrázok 13 Vývojový diagram funkcie „createListenSocket“ UDP



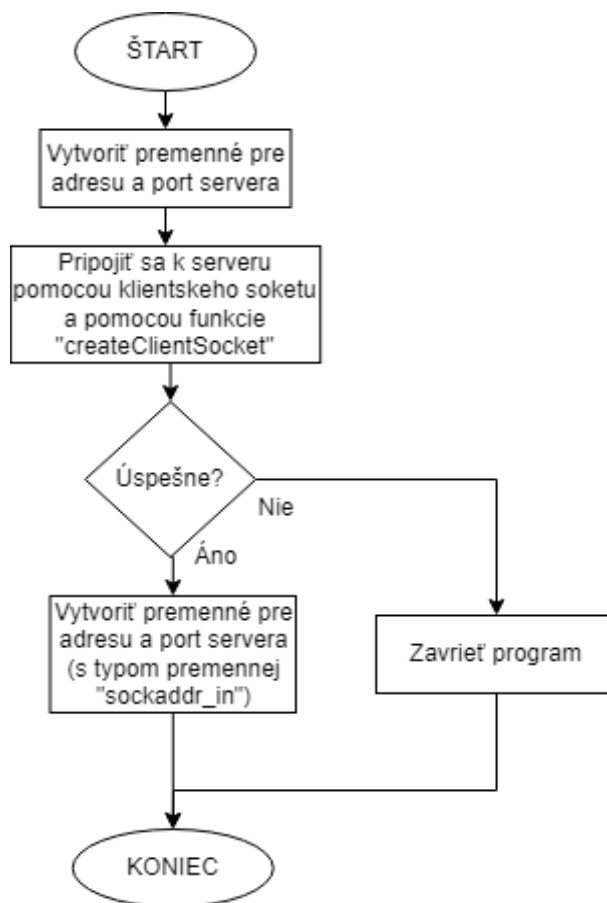
Obrázok 14 Vytvorenie servera v hlavnom programe pomocou protokolu UDP

Štruktúra klienta

Pri spojení klient-server v protokole UDP sa na rozdiel od TCP klient pripojí k serveru ihneď po vytvorení klientskeho soketu (na pripojenie nie je potrebná žiadna samostatná funkcia).



Obrázok 15 Vývojový diagram funkcie „createClientSocket“ UDP



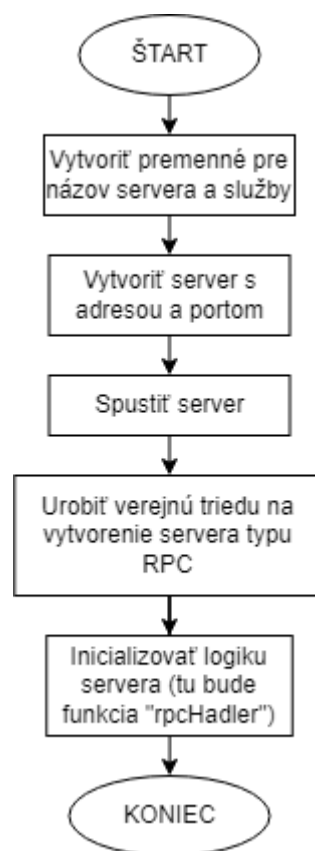
Obrázok 16 Pripojenie klienta k serveru v hlavnom súbore pomocou protokolu UDP

3.3. DIM

DIM je konceptom veľmi odlišný od dvoch predchádzajúcich protokolov. Aby však boli všetci na rovnakej úrovni, vytvorili sme klientske a serverové programy s typom komunikácie RPC (tento typ používajú TCP a UDP). V knižnici protokolu DIM sú špeciálne triedy na vytvorenie spojenia RPC modelu, ktorý používame v tejto práci.

Štruktúra servera

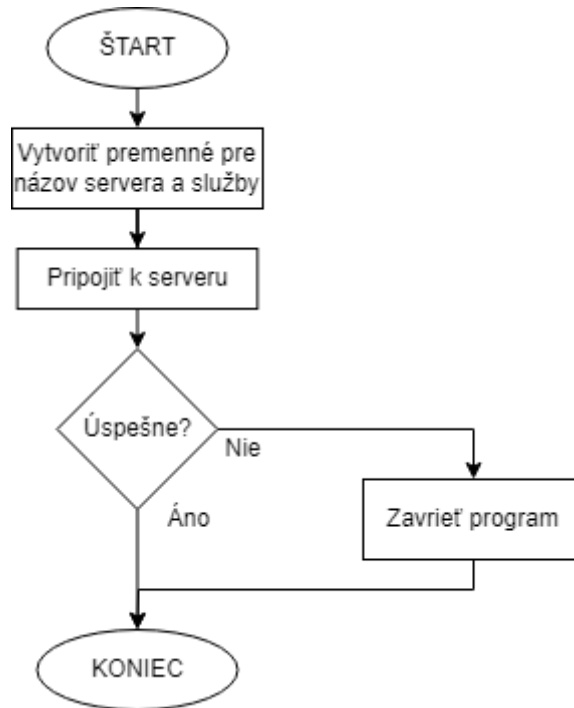
Najprv musíme zadať názov servera a služby, pomocou ktorej sa môže klient pripojiť. Potom vytvoríme požadované triedy a zapneme tento server. Funkcia „**void rpcHandler**“ je hlavná serverová slučka.



Obrázok 17 Vytvorenie servera s typom komunikácie RPC pomocou protokolu DIM

Štruktúra klienta

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.



Obrázok 18 Vytvorenie klienta s typom komunikácie RPC pomocou protokolu DIM

4. Softvérová implementácia

Metodika testovania je pre všetky protokoly rovnaká. V kóde je rozdiel medzi TCP a UDP väčšinou len vo funkcii odosielania / prijímania dátových paketov (rôzne názvy funkcií a rôzne vstupné parametre), ale DIM sa od nich veľmi líši.

Pri popise logiky na meranie priepustnosti a rýchlosti odozvy pre TCP a UDP bude rovnaký popis a stručne naznačíme, ako sme implementovali pomocou protokolu DIM.

Na softvérovej úrovni majú protokoly TCP a UDP logický koniec (napríklad: ak nebolo možné odoslať údaje, nebolo možné vytvoriť server atď.), protokol DIM to však nemá. Ale protokoly TCP a DIM prestanú fungovať, ak sa komunikácia preruší (napríklad: vypne sa klientsky alebo serverový program). Pre DIM to bude jeho logický koniec.

4.1. Meranie priepustnosti

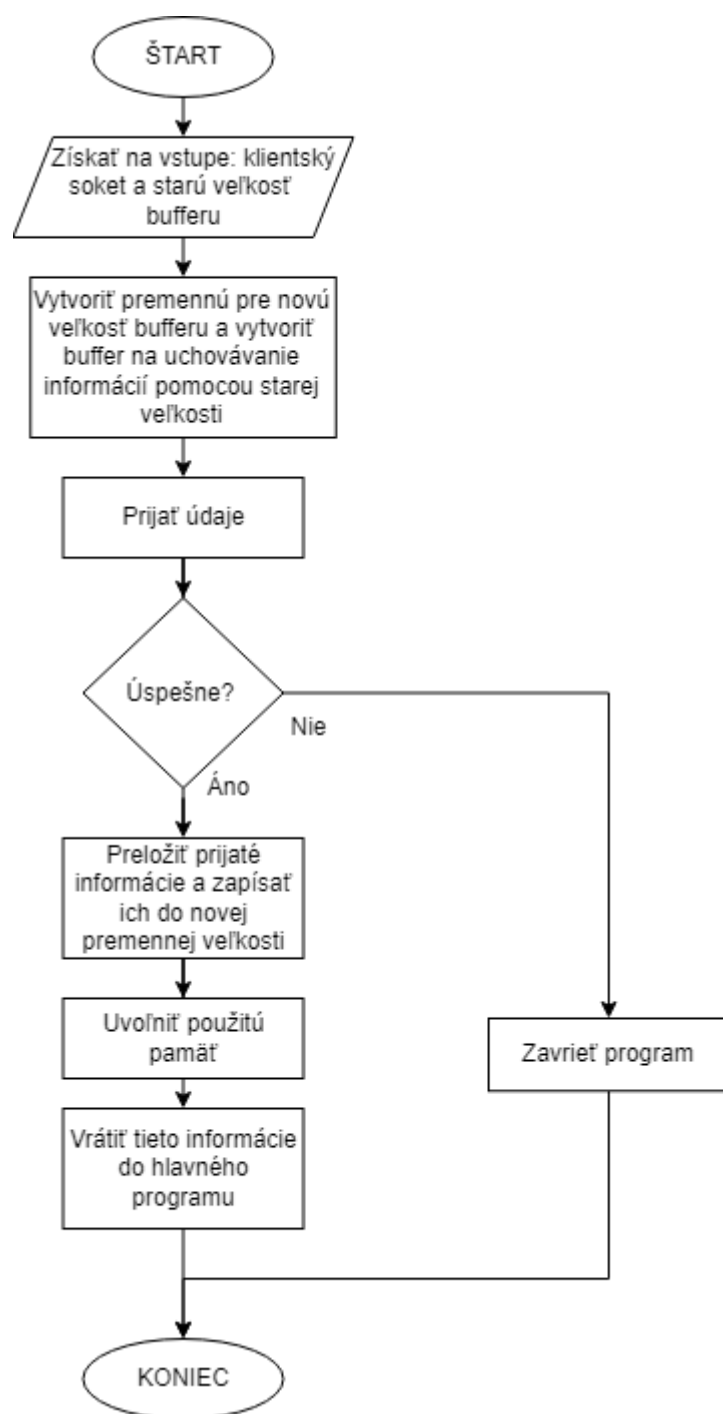
Server TCP/UDP

Server po pripojení používateľa najskôr zadefinuje potrebné premenné a počká na veľkosť vyrovnávacích pamätí, naplní ich a spustí nekonečnú slučku.

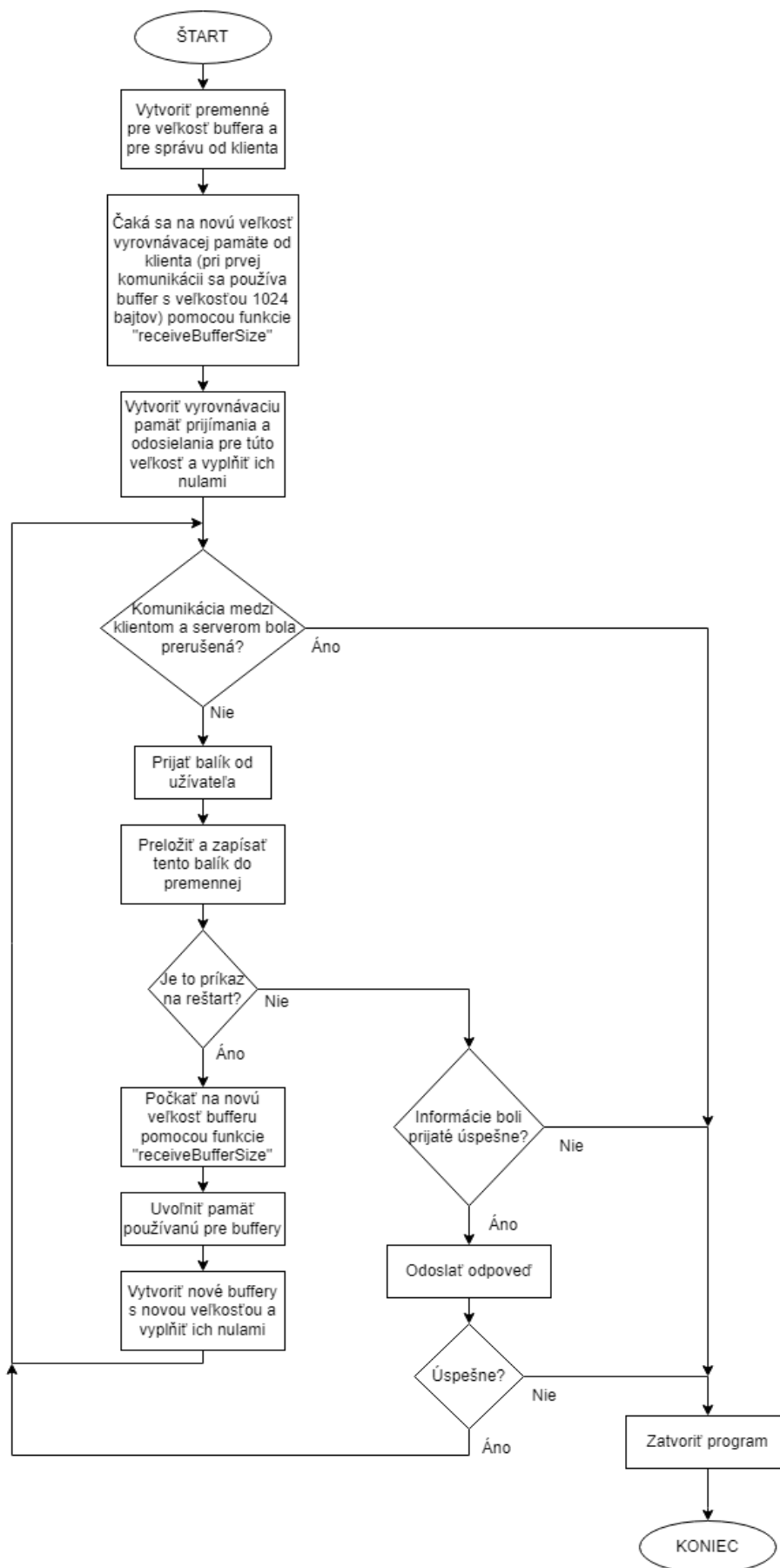
V samotnej slučke čaká na dátový paket, skontroluje ho na príkaz reštartu a hotový paket jednoducho odošle klientovi, aby sme zistili, že dáta boli úspešne prijaté serverom.

Keď sa server reštuje, stále pokračuje v komunikácii s klientom pomocou starej veľkosti paketu a čaká na prijatie novej veľkosti. Potom uvoľní pamäť a vytvorí vyrovnávacie pamäte (bufery) s novou veľkosťou.

V časti 3. Spojenie klient-server sme už spomenuli funkciu "receiveBufferSize". Vytvorená za účelom prijatia údajov o novej veľkosti vyrovnávacej pamäte od klienta pri reštarte (alebo pri prvej komunikácii), ich dešifrovaní a odoslaní informácie späť do hlavného súboru. Pri implementácii v TCP a UDP vyzerá táto funkcia rovnako, s výnimkou samotnej funkcie odosielania údajov a jej vstupných premenných (ako sme už spomenuli v tejto kapitole).



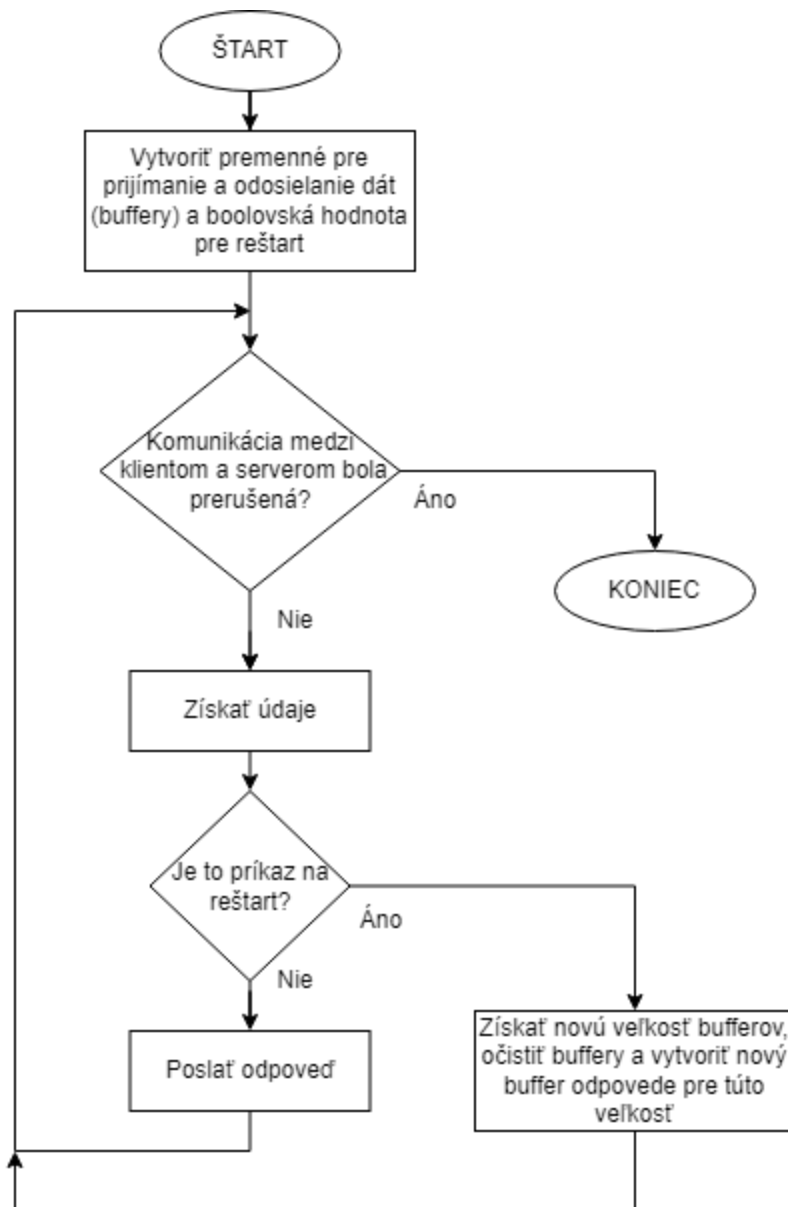
Obrázok 19 Vývojový diagram funkcie „receiveBufferSize“ TCP a UDP



Obrázok 20 Logika serverového programu pri práci s protokolmi TCP a UDP

DIM

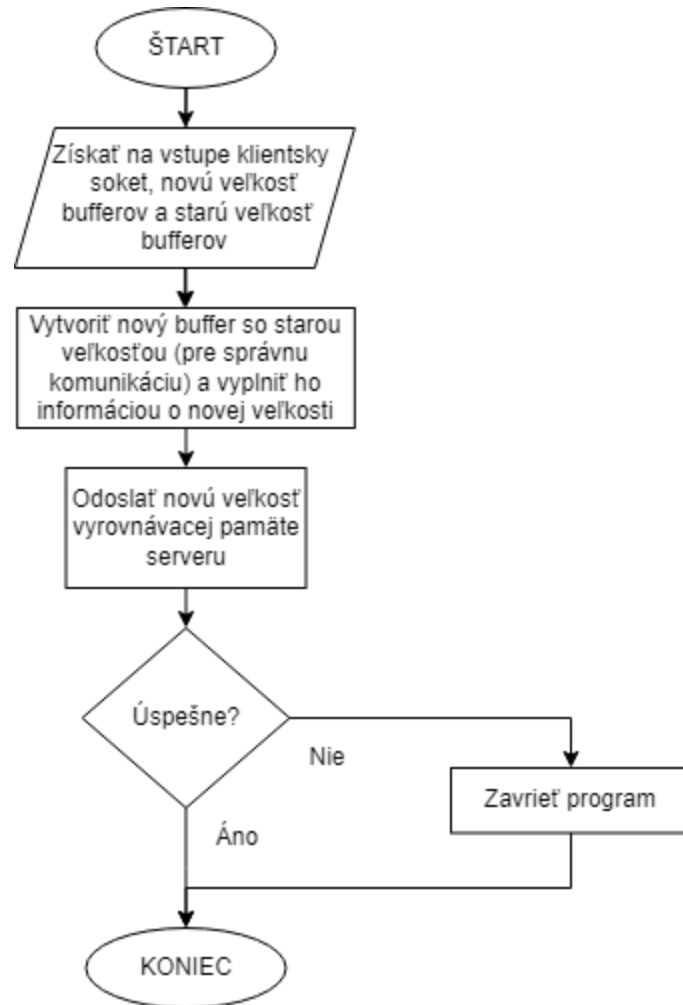
Zo strany servera sa najskôr vytvoria všetky potrebné premenné a začne sa hlavná vnútrotriedová funkcia „**rpcHandler()**“, čo je slučka. Údaje sú vždy kontrolované na prítomnosť príkazu na reštart, ak nie je, potom sa jednoducho odošle odpoveď. A ak potrebujeme reštart, potom server čaká na informáciu o novej veľkosti paketu, zapíše toto číslo do premennej typu „int“, vymaže vyrovnávacie pamäte a naplní buffer odpovede novým množstvom dát.



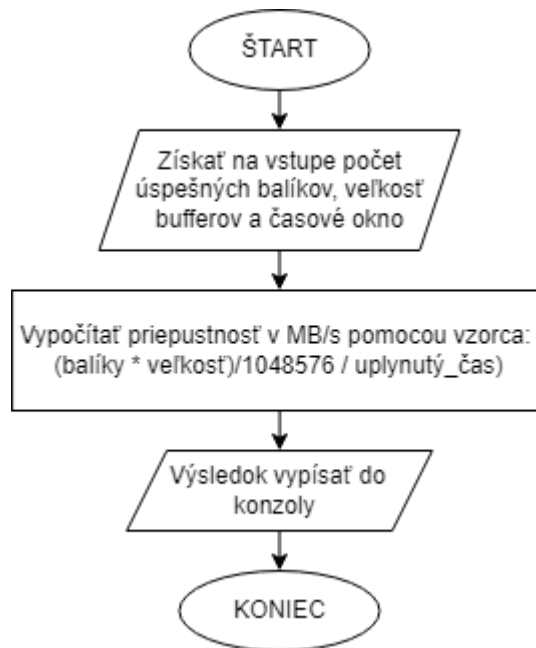
Obrázok 21 Logika serverového programu pri práci s protokolom DIM

Klient TCP/UDP

Aby server vedel o novom množstve odosielaných dát, vytvorili sme funkcie „sendBuffSize“ na vypísanie výsledkov funkcie „show_info“, o ktorých sme písali skôr.



Obrázok 22 Vývojový diagram funkcie „sendBufferSize“ TCP a UDP



Obrázok 23 Vývojový diagram funkcie pri meraní priepustnosti „show_info“ TCP a UDP

Po pripojení klienta k serveru môžeme definovať potrebné premenné, ako je množstvo odoslaných dátových paketov, časové okno (ako dlho budú tieto pakety neustále odosielané), veľkosť vyrovnávacej pamäte a počítadlo odoslaných paketov.

Potom začne hlavná slučka, v ktorej musíme zadať počet bajtov, ktoré klient a server odošle / prijme pre správnu komunikáciu. Na reštartovanie slučky sa táto veľkosť zapíše do dvoch premenných. Pri reštarte môže klient správne odoslať novú veľkosť paketu na server pomocou starej vyrovnávacej pamäte, po čom sa okamžite vytvoria nové buffer s novou dĺžkou.

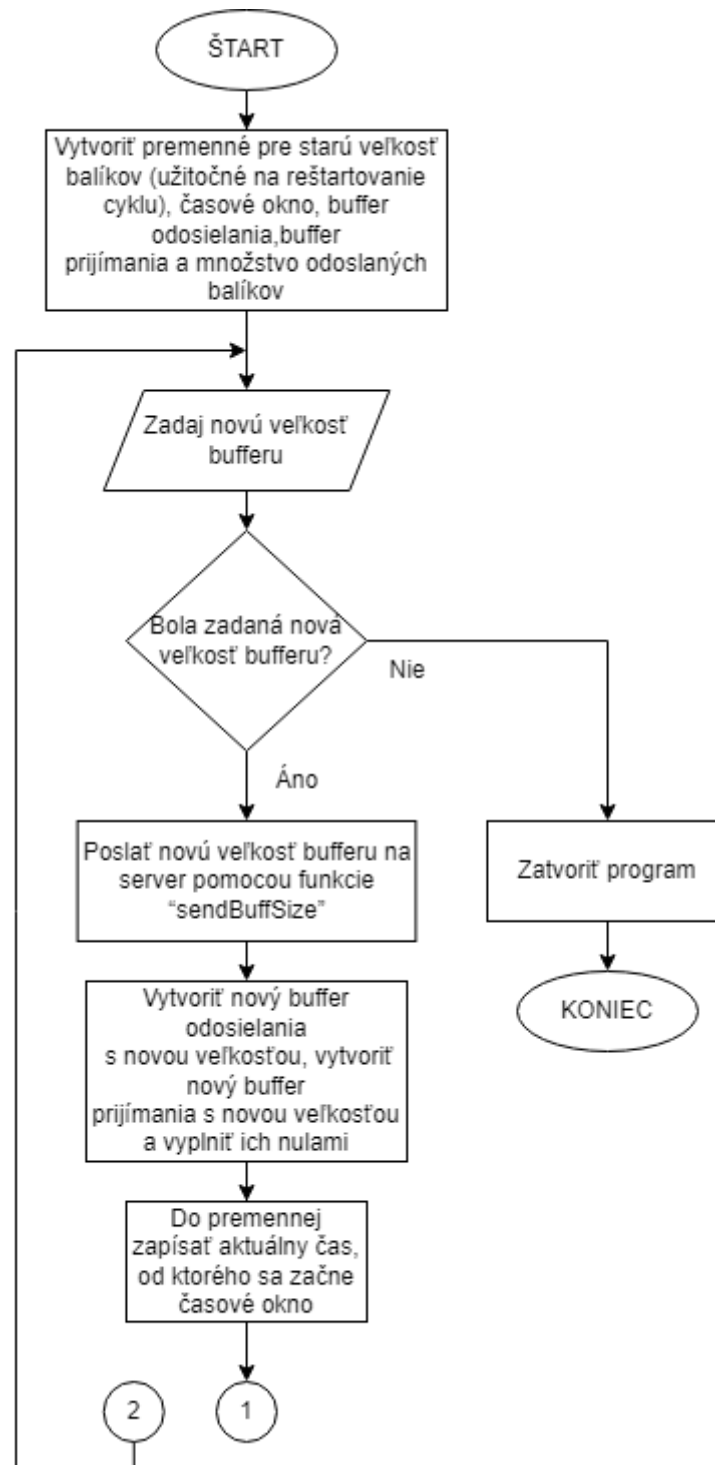
Buffery sú vyplnené nulami, pre dátový typ „char“ 1 znak = 1 bajt.

Aby program určil, koľko času uplynulo, použil som knižnicu „chrono“, pomocou ktorej môžeme preniesť aktuálny čas v počítačovom systéme do premennej. A tiež vypočítať uplynutý čas pomocou interných premenných knižnice.

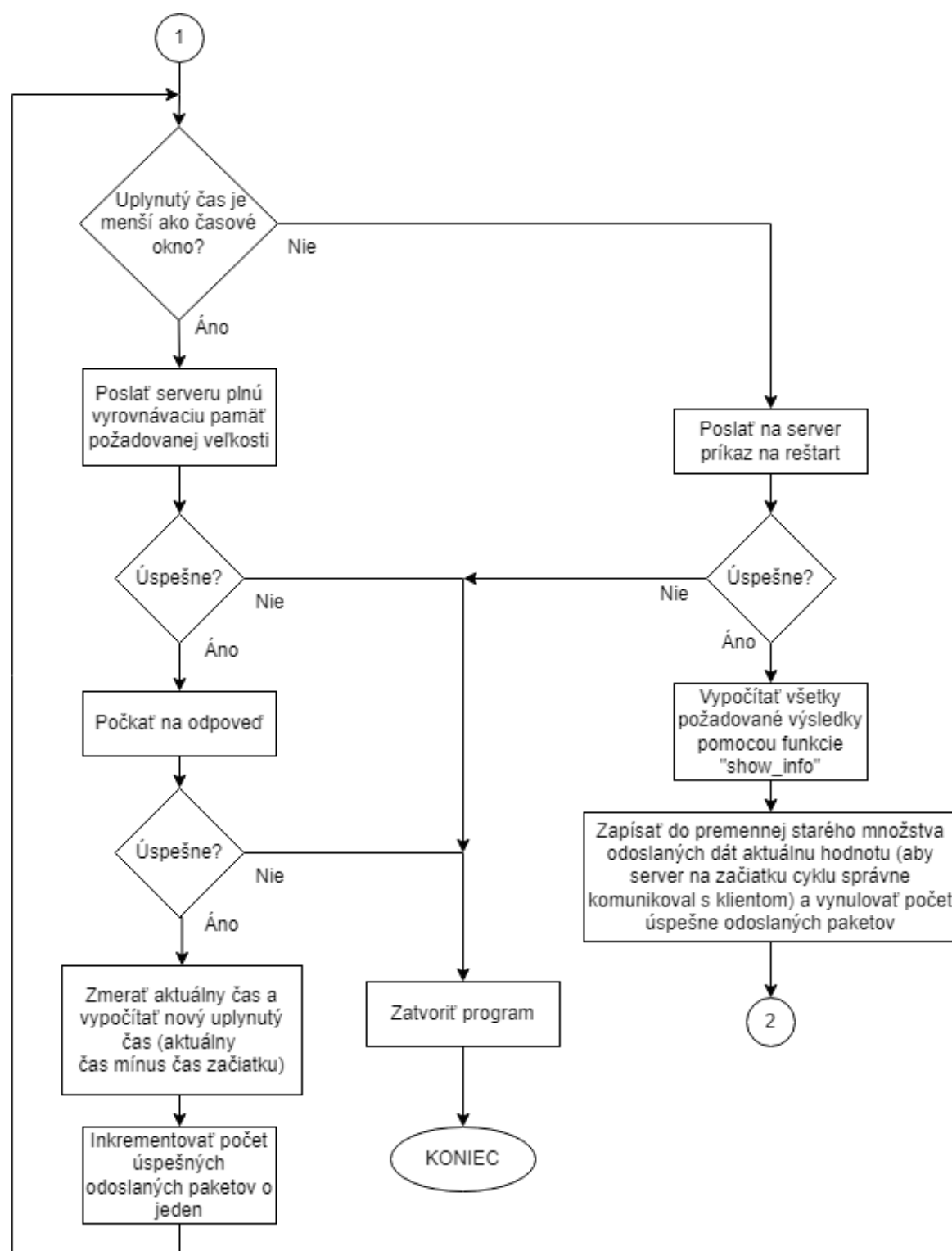
V cykle merania priepustnosti protokolu stačí odosielať a prijímať dáta, merať, koľko času uplynulo od začiatku cyklu a zvýšiť počet úspešných komunikácií o jeden.

Aby sa program mohol úspešne reštartovať bez toho, aby ho bolo potrebné zatvárať a znova otvárať, v serverovom programe sme skontrolovali obsah prichádzajúcich paketov. Ak je číslo „666“, server tiež chápe, že je potrebné ho reštartovať, a klient následne odošle tento príkaz na server, zapíše výsledky testu, zmení potrebné premenné a spustí cyklus od začiatku.

Grafický popis všetkého vyššie uvedeného je znázornený na obrázkoch 24 a 25.



Obrázok 24 Klientsky program pre meranie priepustnosti pomocou protokolov TCP a UDP. Časť 1



Obrázok 25 Klientsky program pre meranie priepustnosti pomocou protokolov TCP a UDP. Časť 2

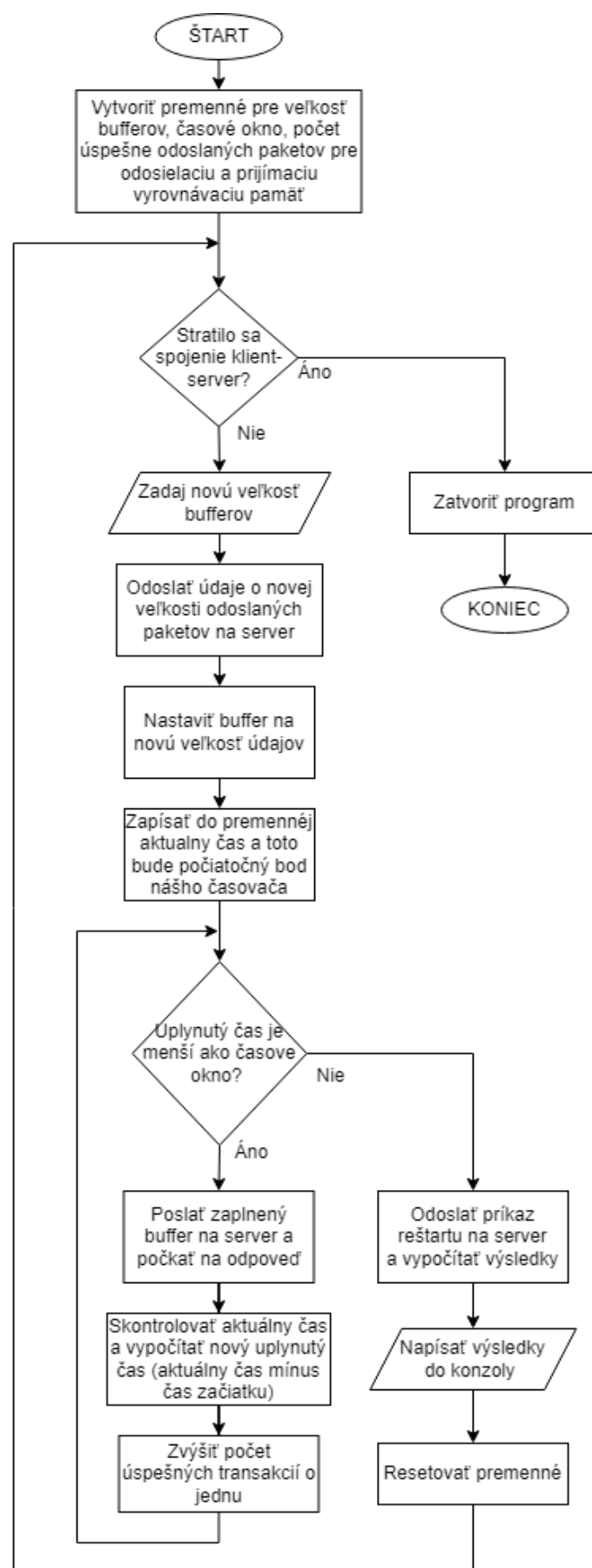
DIM

Tu je tiež potrebné najskôr definovať potrebné premenné, len teraz sme namiesto typu premennej „**char**“ pre buffery použili typ „**string**“ (typ „**string**“ používa 8 bajtov na 1 znak, takže tento buffer sa pri odoslaní sa zmení na typ "**char**").

Potom je potrebné spustiť cyklus, v ktorom klient najskôr zadá novú hodnotu buffera, odošle tieto údaje na server a do premennej naplní čísla 0 požadovaného množstva.

Ďalej zmeriame aktuálny čas, zdefinujeme na to potrebné premenné a spustíme merací cyklus. Funkcia `“service.setData((char*)text.c_str());”` odošle daný buffer na server, zmenený na dátový typ "**char**".

Nakoniec pošleme príkaz reštartu na server, vypočítame počet bajtov odoslaných za sekundu pomocou vzorca a resetujeme premenné.



Obrázok 26 Klientsky program pre meranie priepustnosti pomocou protokolu DIM

4.2. Meranie odozvy

Teraz, na rozdiel od prvého merania (meranie priepustnosti založené na množstve údajov odoslaných za určitý čas), musíme tiež merať čas strávený odosielaním klientom a prijímaním serverom, ako aj spätočnú cestu týchto údajov. Tu sme namerali minimálne, maximálne a priemerné oneskorenie. Množstvo uplynutého času sa tiež meria pomocou knižnice "**chrono**". Namiesto časového okna sme použili priamu hodnotu počtu opakovaní daného cyklu.

Server TCP/UDP/DIM

Program servera je úplne rovnaký ako v odseku 4.1. Meranie priepustnosti.

Klient TCP/UDP

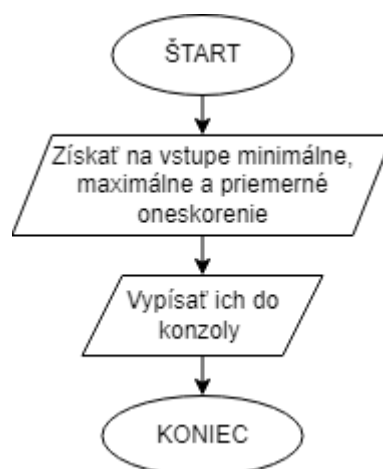
Keďže sú tieto dva programy podobné, zameriam sa na rozdiely nižšie.

Prvým krokom je tiež definovanie potrebných premenných pre hlavnú slučku a výsledné premenné. Existujú už známe premenné, ktoré sú zodpovedné za rovnaké funkcie, ktoré už sme opísali. Premenná "**times**" je počet iterácií výpočtu, "**ping**" je čas strávený jedným opakovaním, "**min**" je minimálne oneskorenie, "**max**" a "**avg**" sú maximálne a priemerné oneskorenie.

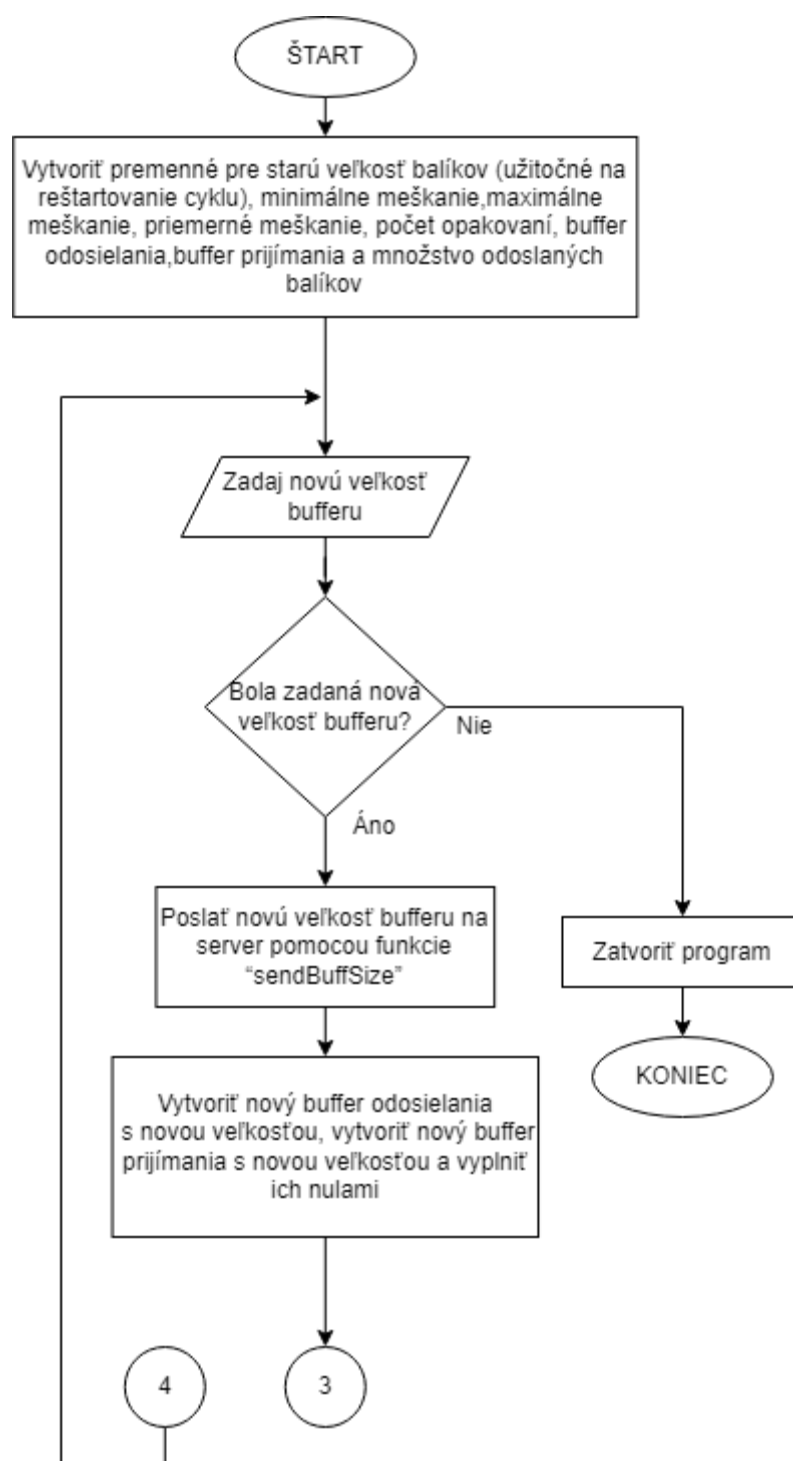
Ďalej je potrebné vytvoriť cyklus opakovania, v ktorom sa zaznamenáva čas pred odoslaním a po prijatí odpovede. Potom sa nové hodnoty zapíšu do pridelených premenných.

Príkaz reštartu je rovnaký ako v prichádzajúcom odseku, ale musíme tiež resetovať nové použité premenné.

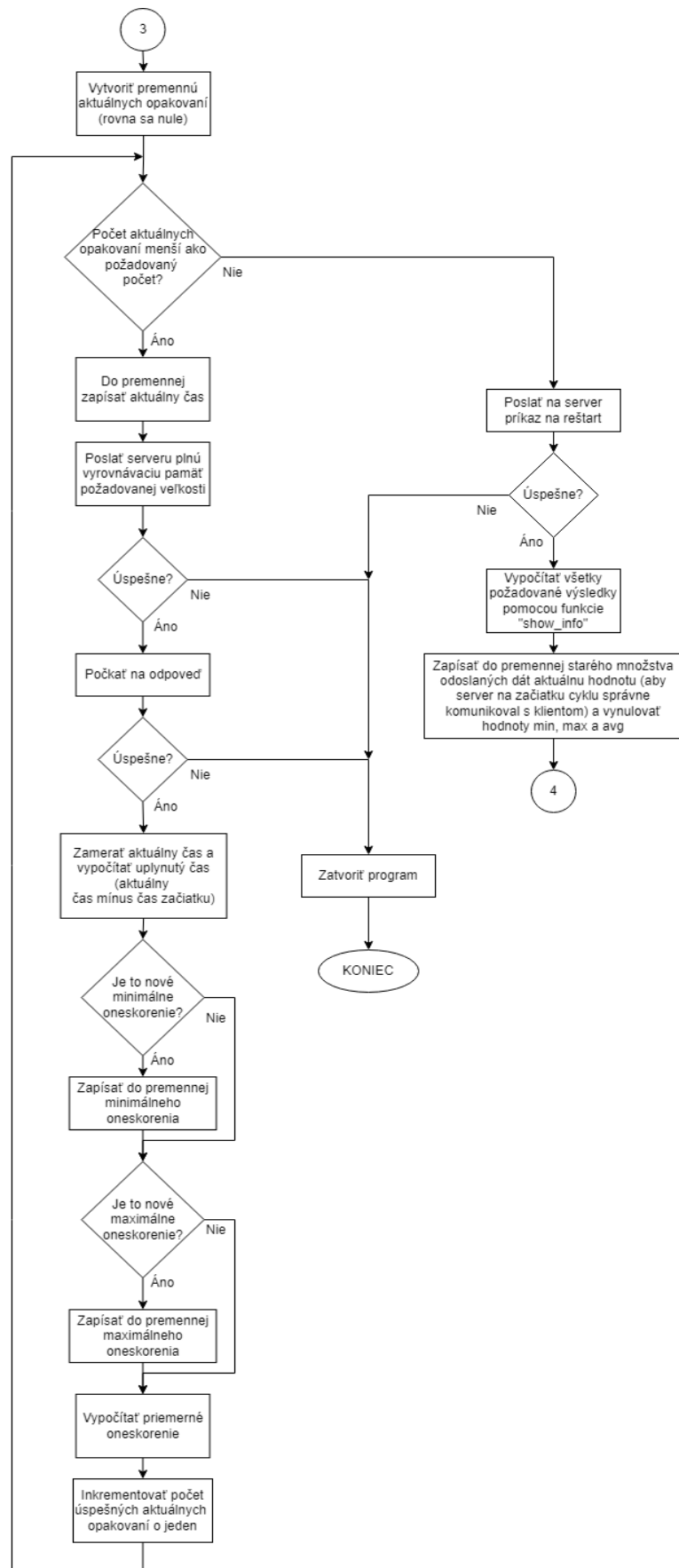
Keďže meriame iné veličiny, musíme funkciu mierne prerobiť na vypísanie výsledkov "**show_info**".



Obrázok 27 Vývojový diagram funkcie pri meraní odozvy „show_info“ TCP a UDP

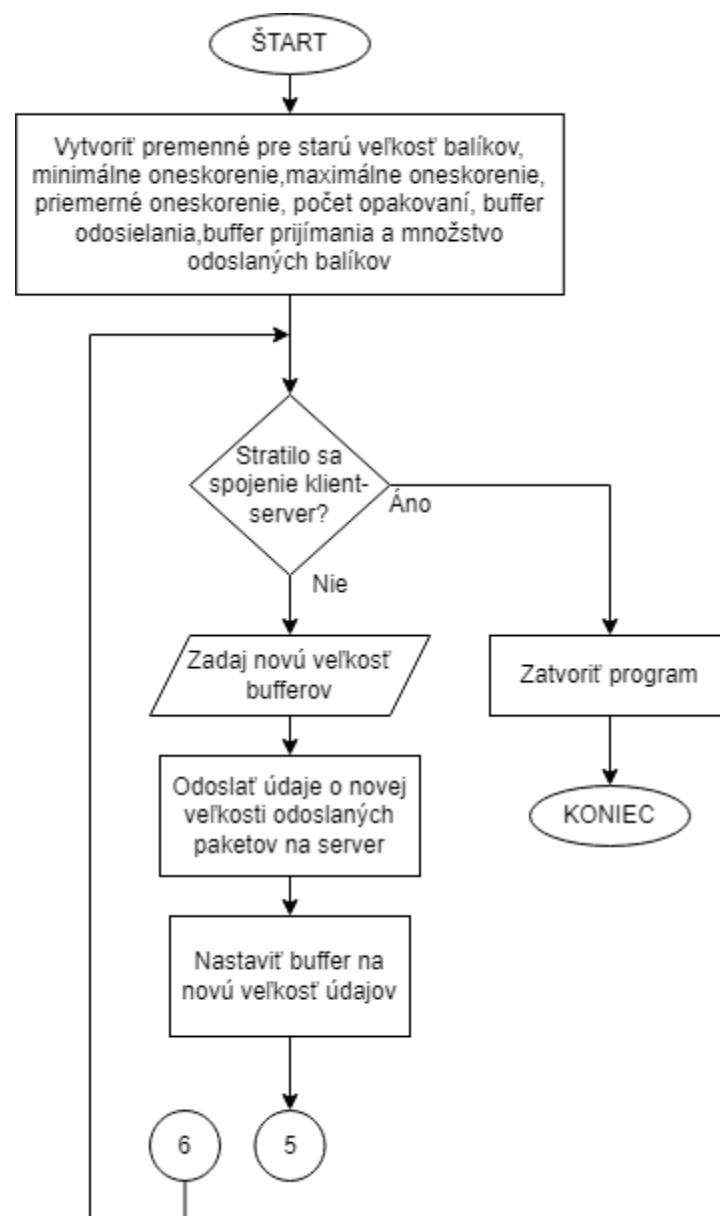


Obrázok 28 Klientsky program pre meranie odozvy pomocou protokolov TCP a UDP. Časť 1

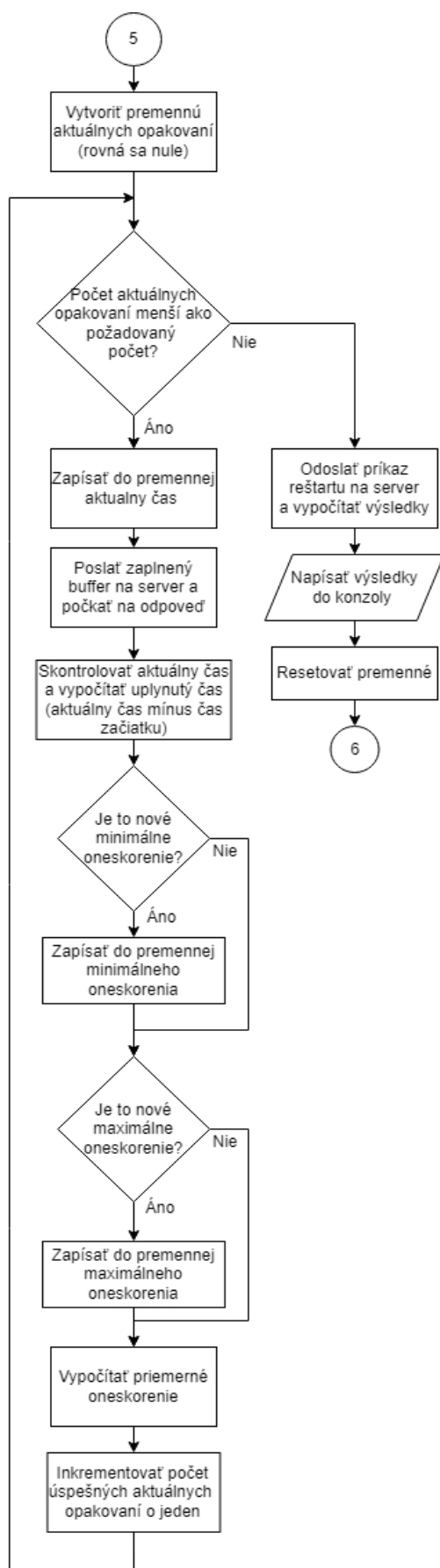


Obrázok 29 Klientsky program pre meranie odozvy pomocou protokolov TCP a UDP. Časť 2

DIM



Obrázok 30 Klientsky program pre meranie odozvy pomocou protokolu DIM. Časť 1



Obrázok 31 Klientsky program pre meranie odozvy pomocou protokolu DIM. Časť 2

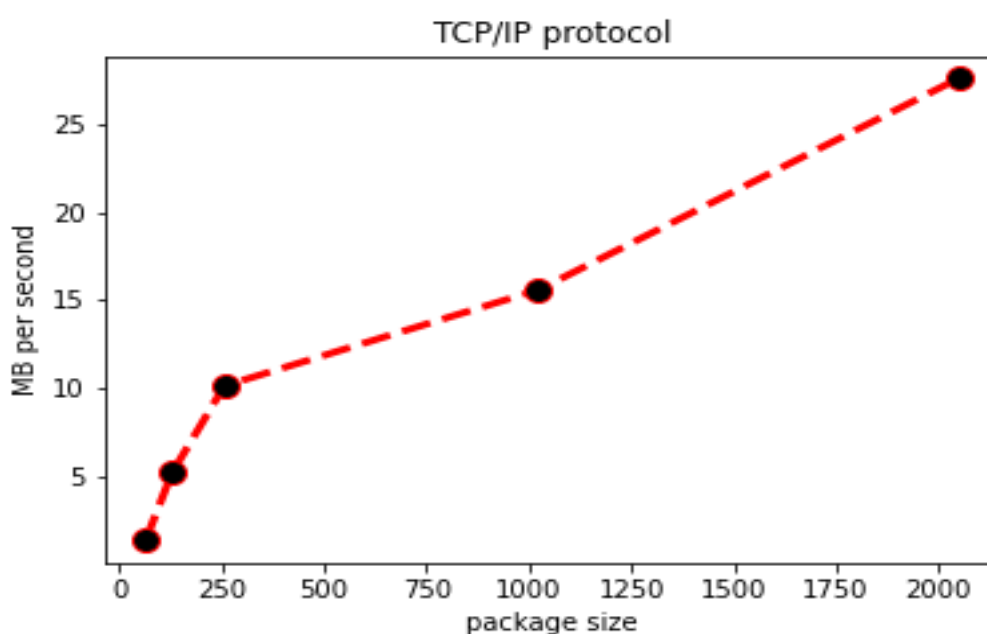
5. Porovnanie internetových protokolov

Pre testovanie priepustnosti a odozvy zvolili sme nasledujúce podmienky:

- Veľkosť balíkov - **64, 128, 256, 1024, 2048 bajtov**.
- Pri **meraní priepustnosti** sme použili časové okno **60 sekúnd**.
- Keď sme zmerali **rýchlosť odozvy**, nastavil som počet opakovaní na **1000000-krát**.
- Keďže sa výsledky počas testovania často mierne líšili, vykonali sme každý typ testovania **10-krát** a vypočítali sme od nich **aritmetický priemer**.

5.1. Meranie priepustnosti

TCP/IP



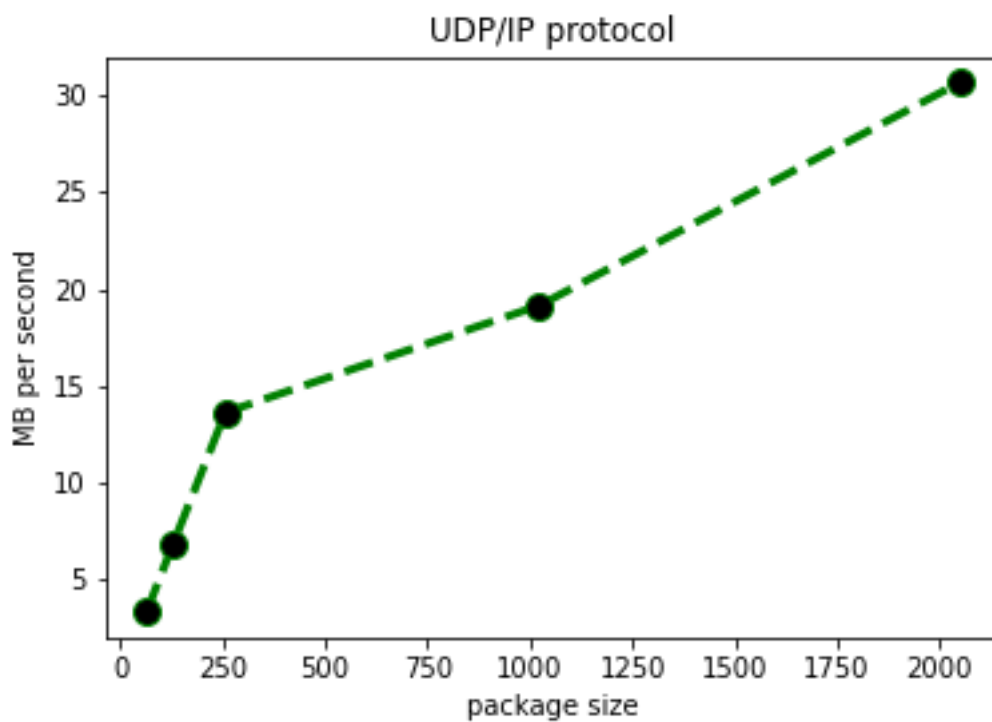
Obrázok 32 Priepustnosť TCP/IP

Po testovaní priepustnosti protokolu TCP sa v priemere získali tieto výsledky:

Počet bajtov	Rýchlosť (v MB/s)
64	1.47
128	5.25
256	10.17
1024	15.60
2048	27.52

Tabuľka 3 Porovnanie množstva a rýchlosti TCP/IP

UDP/IP



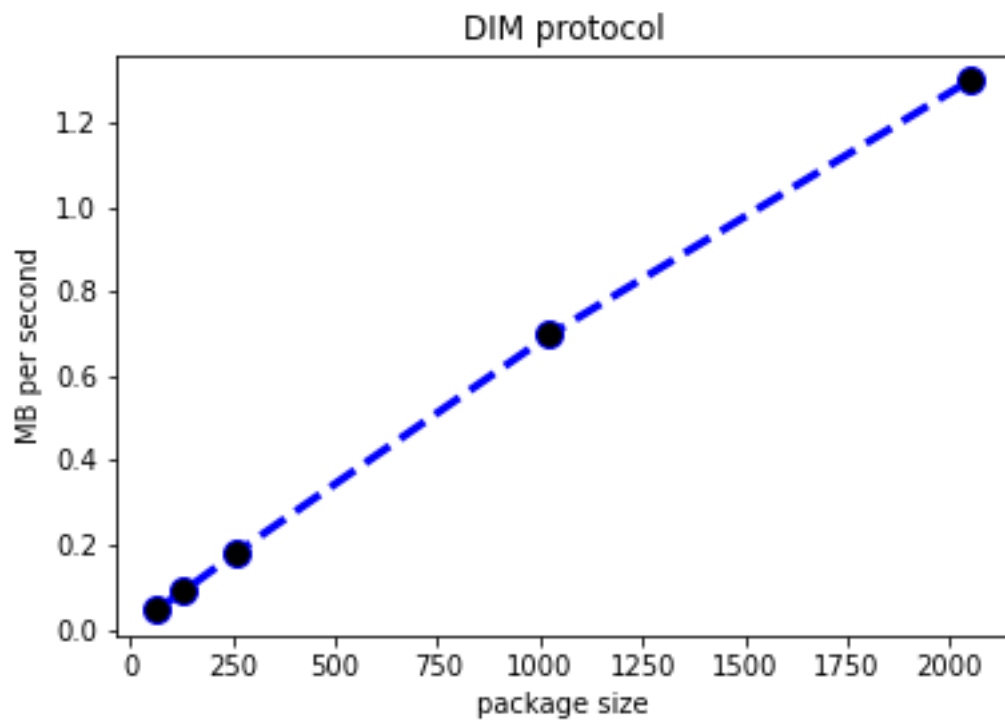
Obrázok 33 Priepustnosť UDP/IP

Po testovaní priepustnosti protokolu UDP sa v priemere získali tieto výsledky:

Počet bajtov	Rýchlosť (v MB/s)
64	3.32
128	6.78
256	13.61
1024	19.15
2048	30.66

Tabuľka 4 Porovnanie množstva a rýchlosti UDP/IP

DIM



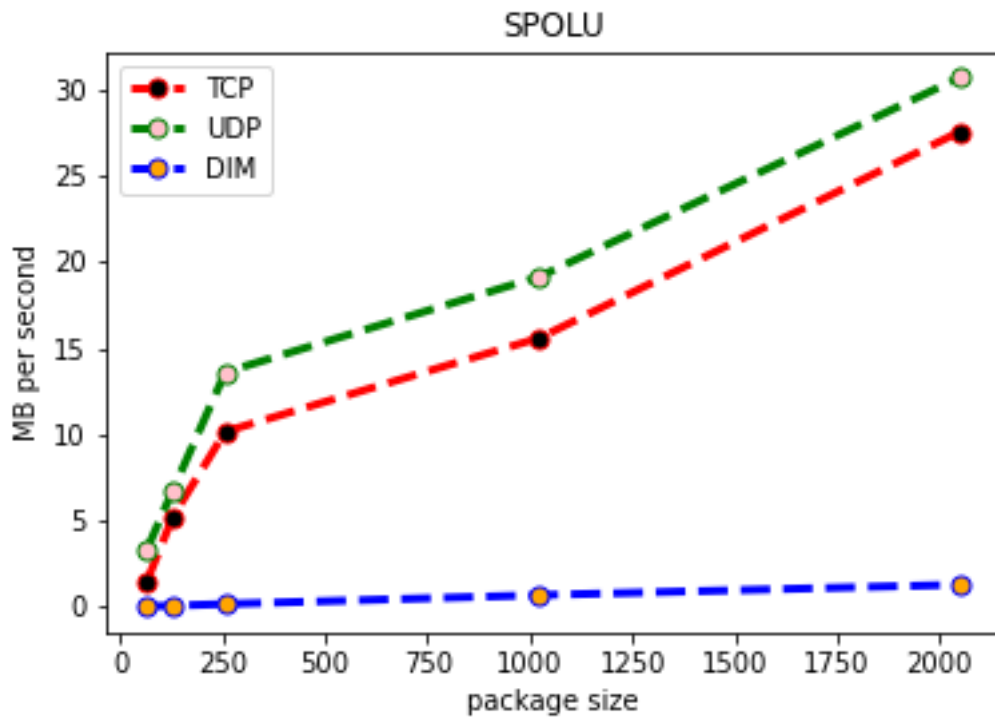
Obrázok 34 Priepustnosť DIM

Po testovaní priepustnosti protokolu DIM sa v priemere získali tieto výsledky:

Počet bajtov	Rýchlosť (v MB/s)
64	0.047
128	0.089
256	0.18
1024	0.698
2048	1.3

Tabuľka 5 Porovnanie množstva a rýchlosti DIM

Porovnanie priepustnosti

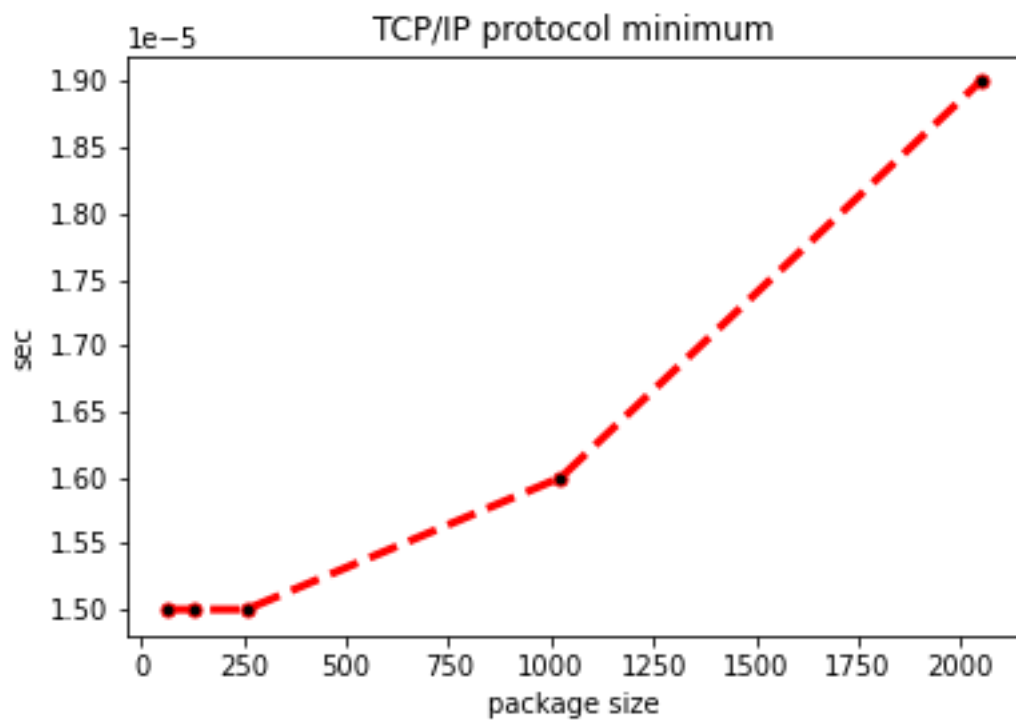


Obrázok 35 Porovnanie výsledkov protokolov pri meraní priepustnosti

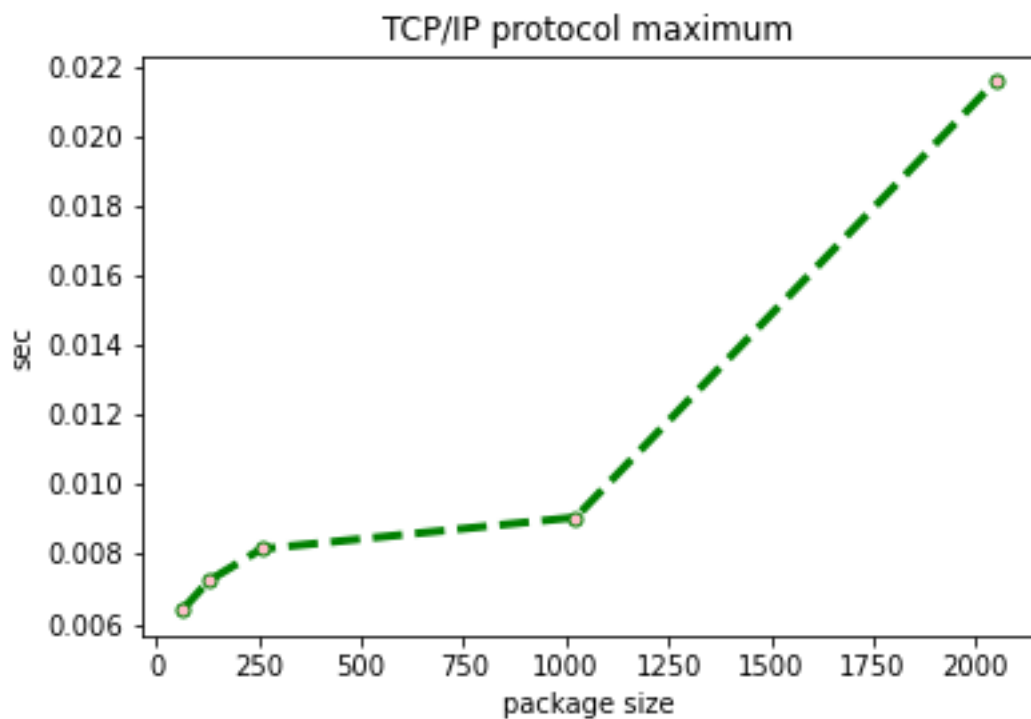
Graf jasne ukazuje rozdiel v rýchlostiach týchto protokolov. **UDP** je lepší ako jeho konkurenti, pretože prijíma a odosiela viac požiadaviek v rovnakom čase. Aj keď dokáže tolerovať stratu paketov.

5.2. Meranie odozvy

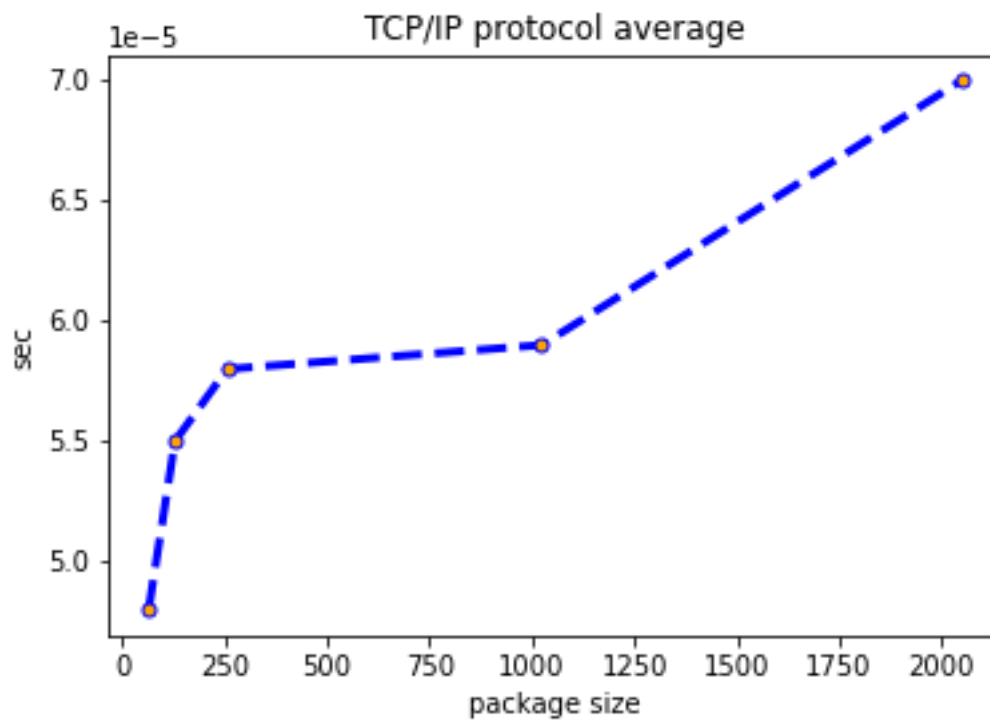
TCP/IP



Obrázok 36 Minimálne oneskorenie protokolu TCP/IP



Obrázok 37 Maximálne oneskorenie protokolu TCP/IP



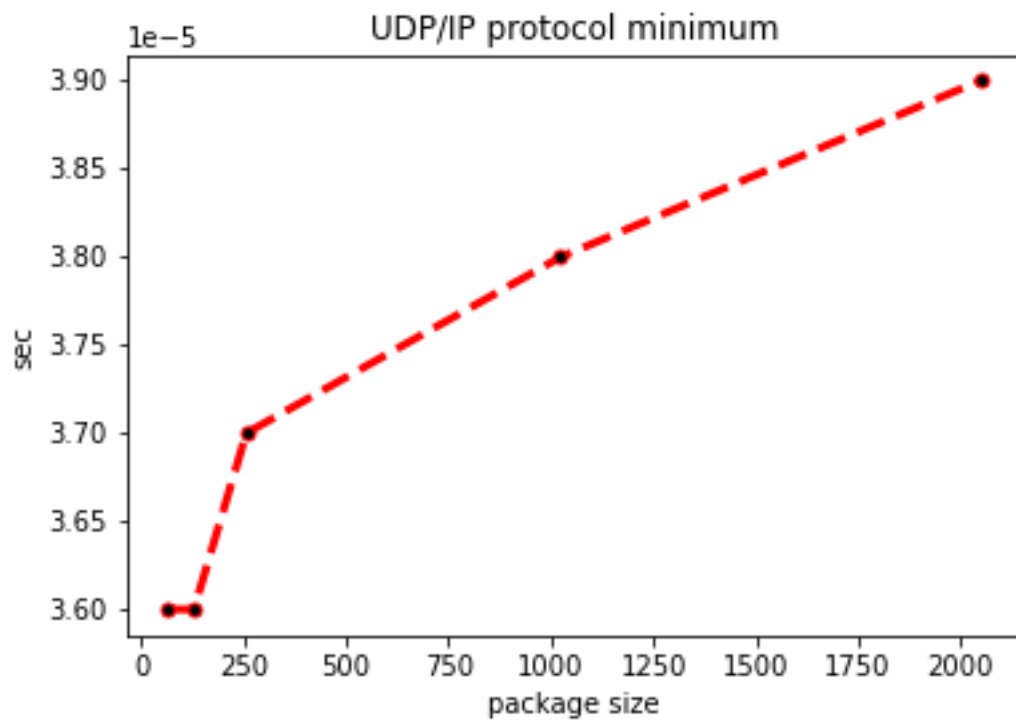
Obrázok 38 Priemerné oneskorenie protokolu TCP/IP

Po testovaní protokolu TCP sa v priemere získali tieto výsledky:

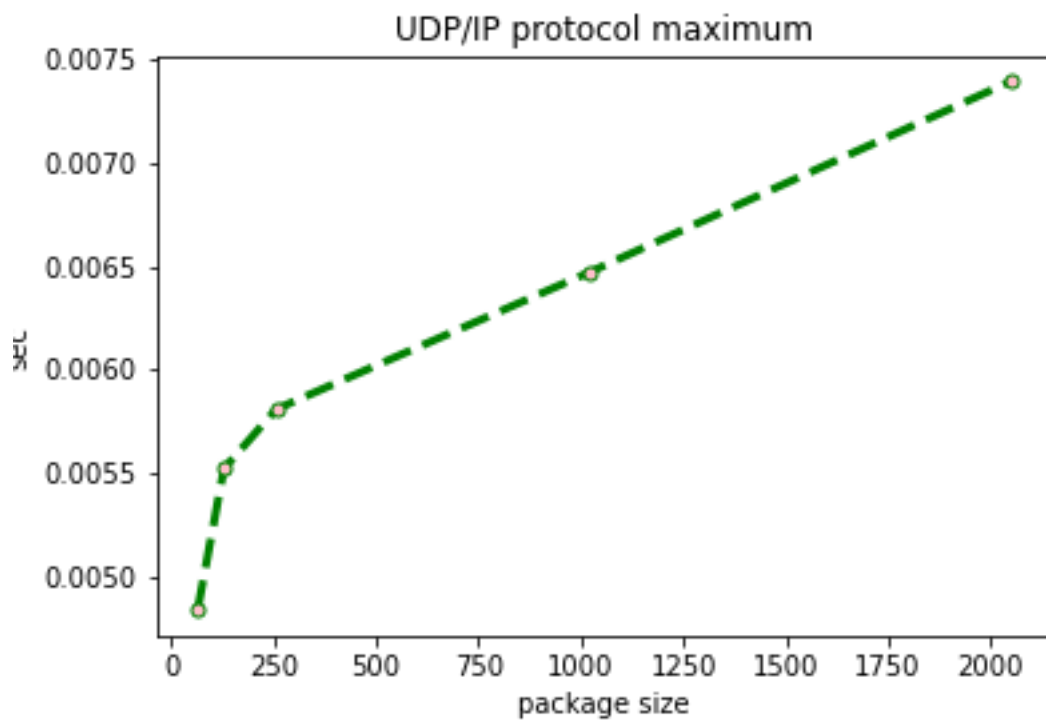
Počet bajtov	Minimálne oneskorenie	Maximálne oneskorenie	Priemerné oneskorenie
64	0.015 ms	6.417 ms	0.048 ms
128	0.015 ms	7.245 ms	0.055 ms
256	0.015 ms	8.156 ms	0.058 ms
1024	0.016 ms	9.067 ms	0.059 ms
2048	0.019 ms	11.593 ms	0.070 ms

Tabuľka 6 Výsledky oneskorenia odozvy protokolu TCP/IP

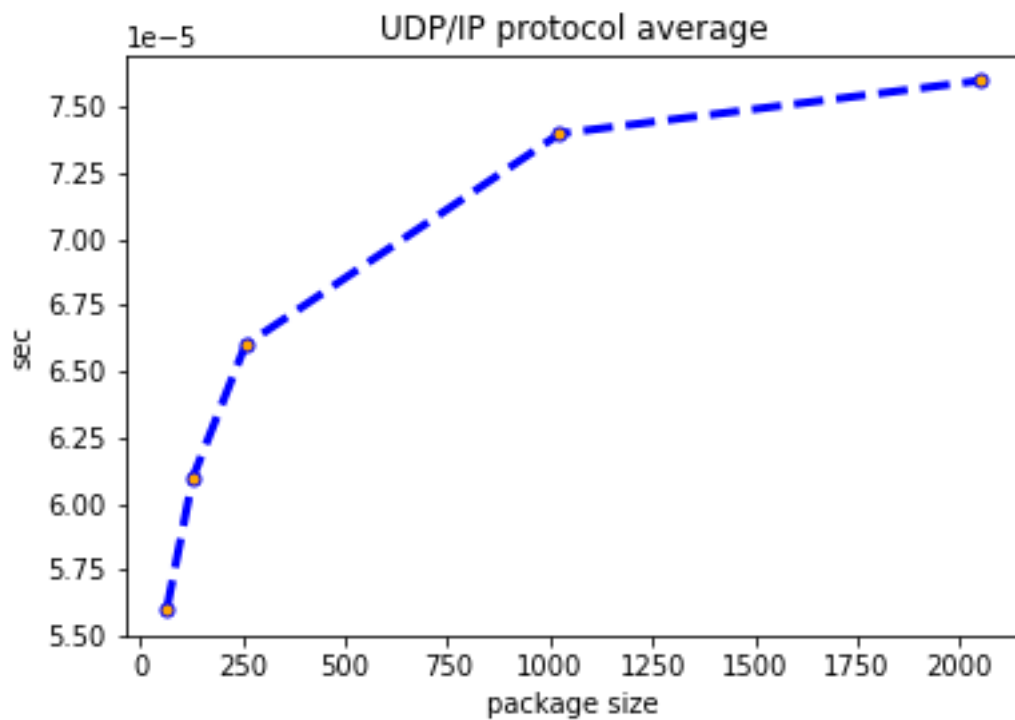
UDP/IP



Obrázok 39 Minimálne oneskorenie protokolu UDP/IP



Obrázok 40 Maximálne oneskorenie protokolu UDP/IP



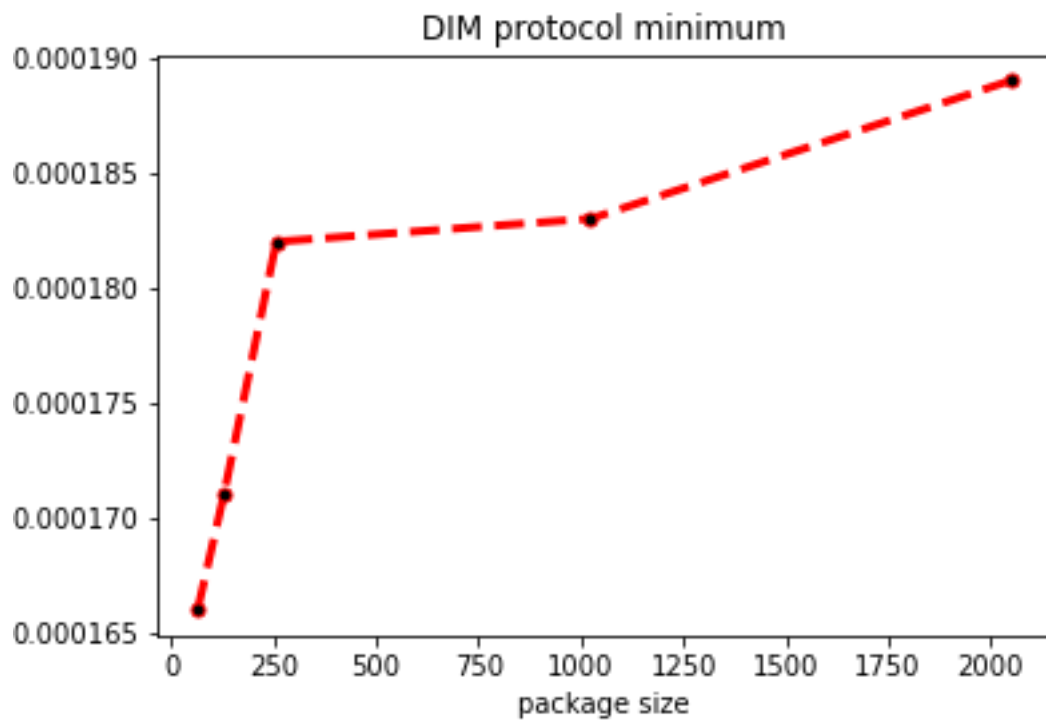
Obrázok 41 Priemerné oneskorenie protokolu UDP/IP

Po testovaní protokolu UDP sa v priemere získali tieto výsledky:

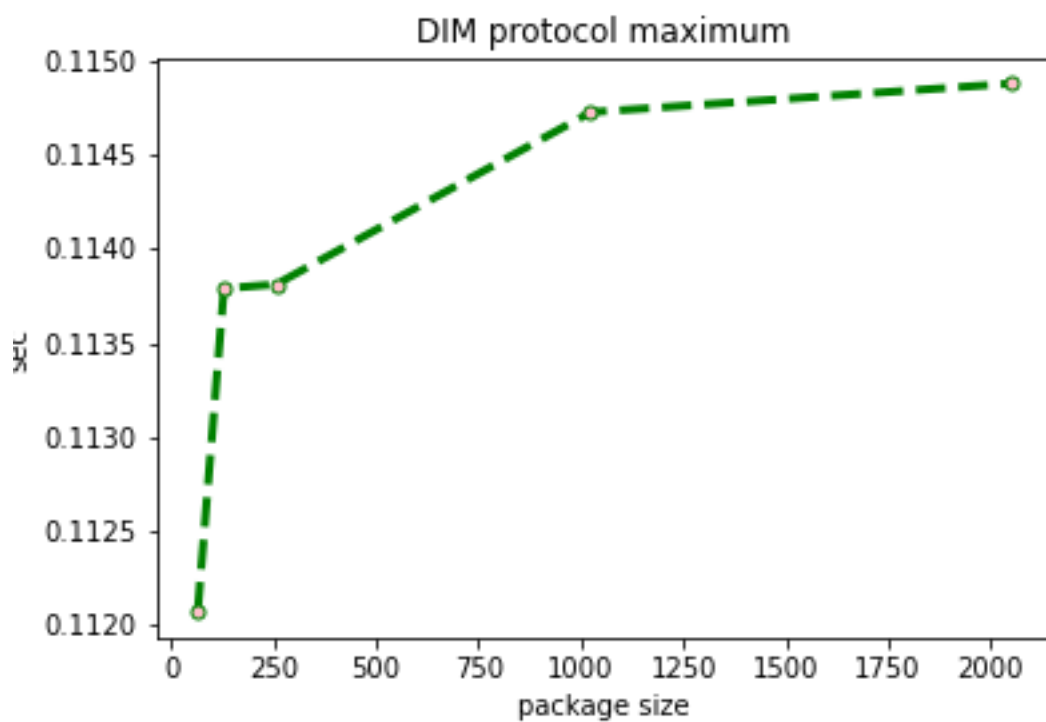
Počet bajtov	Minimálne oneskorenie	Maximálne oneskorenie	Priemerné oneskorenie
64	0.036 ms	4.841 ms	0.056 ms
128	0.036 ms	5.523 ms	0.061 ms
256	0.037 ms	5.811 ms	0.066 ms
1024	0.038 ms	6.477 ms	0.074 ms
2048	0.039 ms	7.397 ms	0.076 ms

Tabuľka 7 Výsledky oneskorenia odozvy protokolu UDP/IP

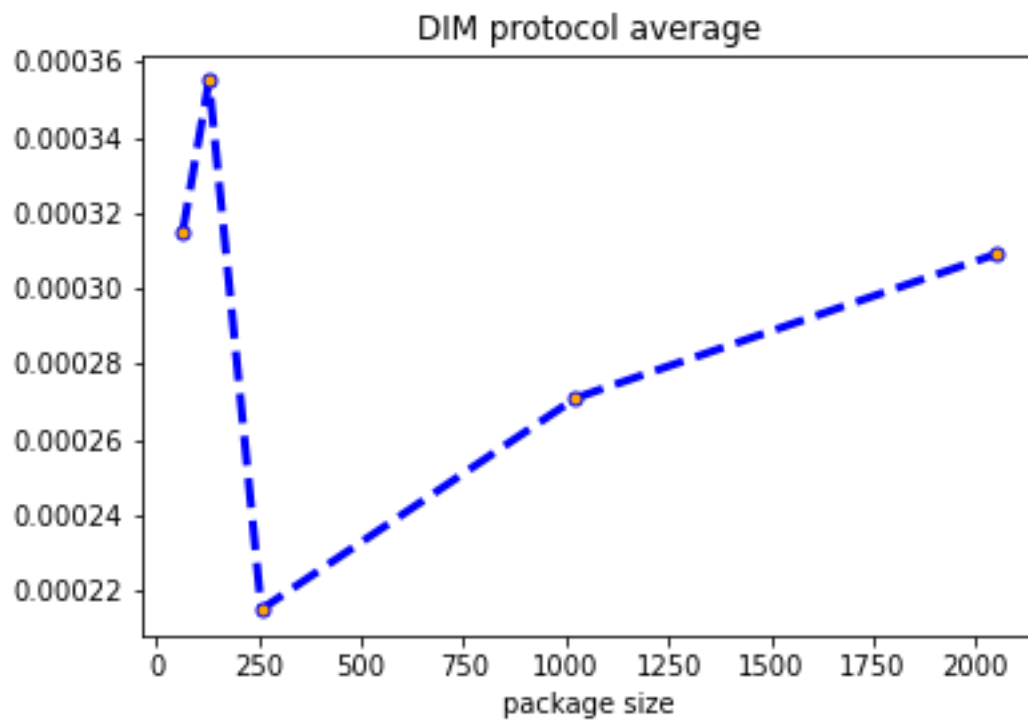
DIM



Obrázok 42 Minimálne oneskorenie protokolu DIM



Obrázok 43 Maximálne oneskorenie protokolu DIM



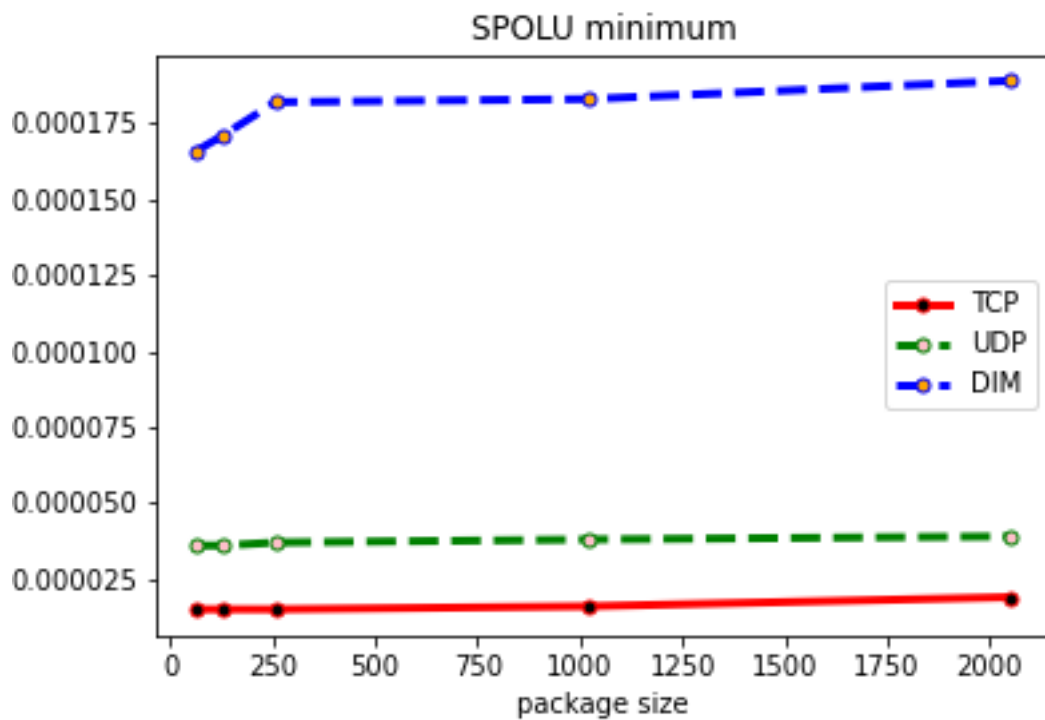
Obrázok 44 Priemerné oneskorenie protokolu DIM

Po testovaní protokolu DIM sa v priemere získali tieto výsledky:

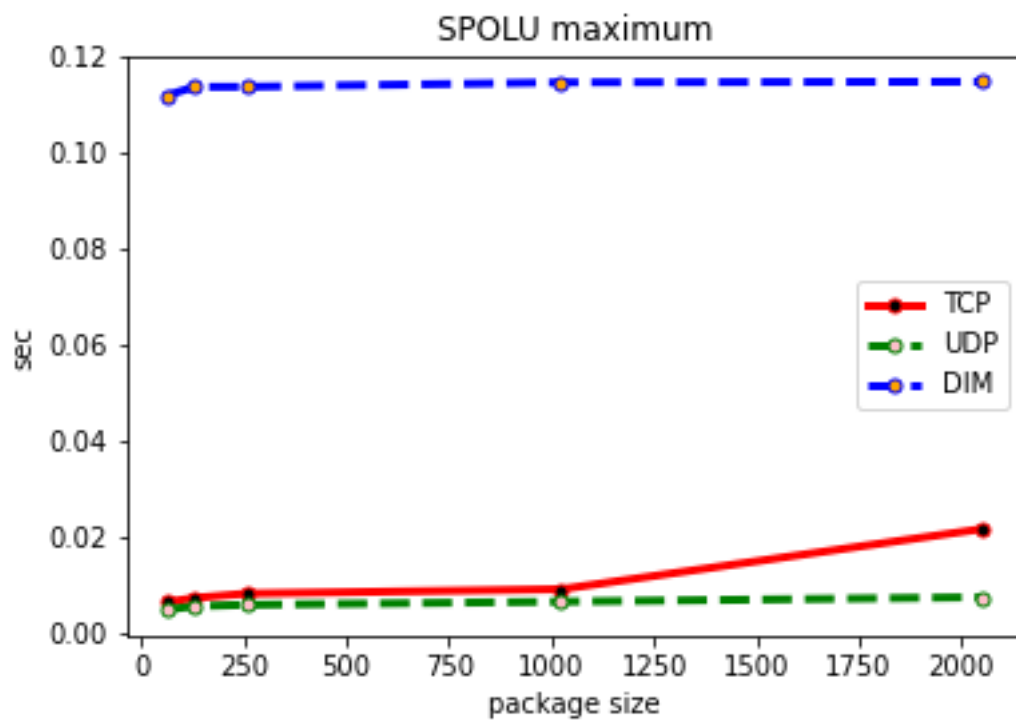
Počet bajtov	Minimálne oneskorenie	Maximálne oneskorenie	Priemerné oneskorenie
64	0.166 ms	112.068 ms	0.315 ms
128	0.171 ms	113.791 ms	0.355 ms
256	0.182 ms	113.813 ms	0.215 ms
1024	0.183 ms	114.731 ms	0.271 ms
2048	0.189 ms	114.883 ms	0.309 ms

Tabuľka 8 Výsledky oneskorenia odozvy protokolu DIM

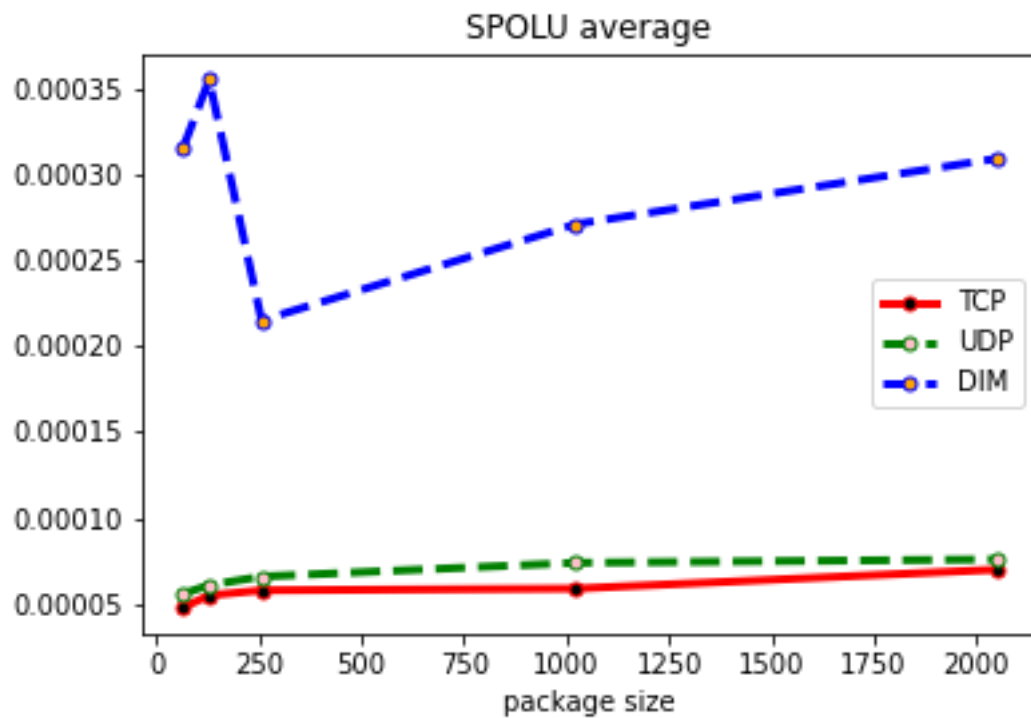
Porovnanie odozvy



Obrázok 45 Porovnanie minimálneho oneskorenia protokolov



Obrázok 46 Porovnanie maximálneho oneskorenia protokolov



Obrázok 47 Porovnanie priemerného oneskorenia protokolov

Tieto grafy ukazujú, že minimálne oneskorenie pre protokol TCP je menšie ako pre ostatné, ale maximálne oneskorenie je väčšie ako umožňuje UDP. Priemerné oneskorenie so zvýšením množstva odoslaných dát rastie rýchlejšie pre UDP, TCP má tento ukazovateľ mierne navýšený.

Záver

V tejto práci som prešiel dlhú cestu od pochopenia protokolov k štúdiu ich priepustnosti a rýchlosti odozvy. Keď som prešiel všetkými fázami vývoja týchto protokolov, všimol som si veľkú podobnosť medzi protokolmi TCP a UDP vo všetkých aspektoch. A po preštudovaní ich rýchlostných možností dospel k záveru, že sú veľmi dobré na ich výber ako prenosový protokol medzi programami klient-server.

TCP je rýchly a bezchybný protokol, to znamená, že vďaka známej metóde odosielenia dát pomocou „handshake“ sa súbory vždy dostanú k odosielaťovi, čím sa eliminuje možnosť straty paketov. A z mínusov môžem vypichnúť „fronty“ medzi požiadavkami používateľov, pretože nie je schopný spracovať informácie a okamžite ich odoslať veľkému počtu klientov. A napriek tomu je z hľadiska priepustnosti horší ako protokol UDP, takže podľa môjho názoru je zvýšenie rýchlosti odosielenia paketov rozumným vývojom protokolu TCP. Odporúčam ho použiť v akomkoľvek programe, kde veľa používateľov nebude posielať svoje požiadavky súčasne. Napríklad také programy ako: internetový obchod, messenger, webové stránky atď.

UDP je zase najlepší z nich. Svojou priepustnosťou necháva ostatných za sebou a navyše vďaka podpore veľkého počtu používateľov je takmer ideálny. Z mínusov je samozrejme možná strata paketov v dôsledku skutočnosti, že pakety sú jednoducho odosielené jeden po druhom bez toho, aby vedeli, pre ktoré zariadenie sú určené. Aj keď sme sa už naučili rôznymi spôsobmi obnoviť túto stratu paketov, ale stále, ako si myslím, rozumným riešením by bolo zabudovať kontrolu, či paket dosiahol svoj cieľ v samotnom protokole. Potom by sme si nemuseli vytvárať vlastne funkcie a celkovo sa s tým trápiť. S najväčšou pravdepodobnosťou to trochu spomalí rýchlosť samotných údajov, pretože server bude stále zaneprázdnený týmito kontrolami, ale pre mňa je to rozumná obeta. Odporučil by som ho použiť pre všetky možné programy, najmä ak máme znalosti a schopnosti prekonať stratu paketov.

Protokol DIM svojou konštrukciou nebol určený pre komunikáciu typu „RPC“, preto ho nemožno striktné posudzovať. Koniec koncov s jeho pomocou môžeme robiť dobrých messengerov, webové stránky alebo online boty. DIM je veľmi pomalý z hľadiska rýchlosti v porovnaní s TCP a UDP, ale je najbezpečnejší z hľadiska straty paketov. Preto ho odporúčam používať, ak je potrebné spoľahlivo a bezpečne prenášať dáta z jedného počítača do druhého.

Zoznam použitej literatúry

- [1] R. Callon, Digital Equipment Corporation. (December 1990) – RFC: 1195 Use of OSI IS-IS for Routing in TCP/IP and Dual Environments
- [2] Information Sciences Institute University of Southern California. (September 1981) – RFC: 791 INTERNET PROTOCOL, DARPA INTERNET PROGRAM, PROTOCOL SPECIFICATION
- [3] Information Sciences Institute University of Southern California. (September 1981) – RFC: 793 TRANSMISSION CONTROL PROTOCOL, DARPA INTERNET PROGRAM, PROTOCOL SPECIFICATION
- [4] Cerf, V. Kahn, R. Kahn. (May 1974) – A Protocol for Packet Network Intercommunication, IEEE Transactions on Communications
- [5] Dalal, Y. Sunshine, C. Sunshine. (December 1978) – Connection Management in Transport Protocols, Computer Networks
- [6] Oracle Corporation and/or its affiliates. (2010) – System Administration Guide, Volume 3 - TCP/IP Protocol Architecture Model – <https://docs.oracle.com/cd/E19455-01/806-0916/ipov-10/index.html>
- [7] Vinton Cerf, Yogen Dalal, Carl Sunshine. (December 1974) – RFC: 675 SPECIFICATION OF INTERNET TRANSMISSION CONTROL PROGRAM
- [8] Gary R. Wright, W. Richard Stevens. (January 1995) – TCP/IP Illustrated, ISBN: 020163354X, Volume 2, Chapter 23 – UDP: User Datagram Protocol
- [9] Linda Rosencrance, George Lawton, Chuck Moozakis. (October 2021) – DEFINITION User Datagram Protocol (UDP)
- [10] J. Postel, ISI. (28 August 1980) – RFC: 768 User Datagram Protocol
- [11] L. Eggert, NetApp, G. Fairhurst, University of Aberdeen, G. Shepherd, Cisco Systems. (March 2017) – RFC: 8085 UDP Usage Guidelines
- [12] DIM - Distributed Information Management System–https://dim.web.cern.ch/dim_intro.html
- [13] C. Gaspar. (February 11 1999) – Class: DimRpc
https://dim.web.cern.ch/cpp_doc/DimRpc.html
- [14] C. Gaspar. (February 11 1999) – Class: DimRpcInfo
https://dim.web.cern.ch/cpp_doc/DimRpcInfo.html

[15] C. Gaspar, M. Donszelmann, Ph. Charpentier, CERN, European Organisation for Nuclear Research. – DIM, a Portable, Light Weight Package for Information Publishing, Data Transfer and Inter-process Communication

[16] C. Gaspar, EP Department, With additional contributions. (April 2022) – Distributed Information Management System

[17] P. Mockapetris, ISI. (November 1987) - RFC: 1034 DOMAIN NAMES - CONCEPTS AND FACILITIES

[18] Windows Sockets 2 Architecture –

<https://docs.microsoft.com/en-us/windows/win32/winsock/windows-sockets-2-architecture-2>

Prílohy

Príloha A: CD médium – bakalárska práca v elektronickej podobe, prílohy v elektronickej podobe.

Príloha B: Používateľská príručka

Príloha C: Systémová príručka

TECHNICKÁ UNIVERZITA V KOŠICIACH
FAKULTA ELEKTROTECHNIKY A INFORMATIKY

**POROVNANIE PRIEPUSTNOSTI A ODOZVY SIEŤOVÝCH
KOMUNIKAČNÝCH PROTOKOLOV**
Príloha B – Používateľská 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

Obsah

Zoznam obrázkov	75
Úvod	76
1 Inštalácia a pochopenie programov	76
2 Spúšťanie programov	76
Záver	81

Zoznam obrázkov

Obrázok 1 Zadanie požadovaného priečinka	77
Obrázok 2 Spustenie servera.....	78
Obrázok 3 Spustenie klienta.....	78
Obrázok 4 Vypisovanie výsledkov testu	79
Obrázok 5 Zadanie požadovaného priečinka(DIM)	79
Obrázok 6 Spustenie DNS.....	80

Úvod

Predmetom tejto používateľskej príručky je oboznámiť koncového používateľa s aplikáciami, ich funkciami a návodom na ich používanie. Konkrétne ide o tieto tri aplikácie: Klientsky program merania priepustnosti, Klientsky program merania odozvy a serverový program pre nich. V tomto dodatku budú krok za krokom analyzované inštrukcie na spustenie programu pre protokoly TCP,UDP a DIM.

Inštalácia a pochopenie programov

Najprv je potrebné skopírovať súbory umiestnené v prílohe. Každý priečinok TCP a UDP protokolov obsahuje dva programy - jeden klient, druhý server. V prípade protokolu DIM sú to 3 - DNS, klient a server. Celkovo je teda 14 programov.

Priečinky TCP1, UDP1 a DIM1 ukladajú aplikácie na **meranie priepustnosti**. TCP2, UDP2 a DIM2 - **meranie odozvy**. Časové okno pre meranie priepustnosti je už definované v programe, je to 60 sekúnd. Počet opakovaní pre meranie odozvy tiež už definované a je to stotisíckrát(100 000).

Adresa servera v oboch typoch merania je *“localhost”*, konkrétne – *“127.0.0.1”*. **Port** je tiež preddefinovaný a je to *“2222”*.

Výsledky budú zapísané v **MB/s** (megabajtov za sekundu) pri meraní priepustnosti a v **ms** (milisekundách) pri meraní odozvy.

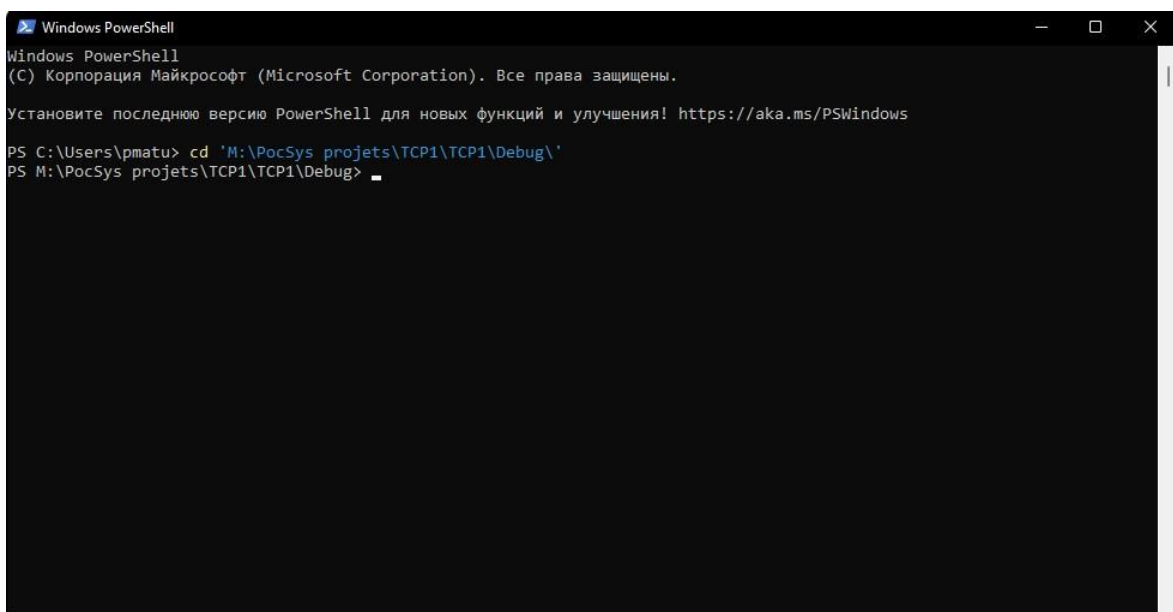
Spúšťanie programov

Programy už majú svoje parametre potrebné na testovanie, užívateľovi zostáva len ich správne spustiť a zadať do klientskeho programu veľkosť odosielaných paketov. Potom je potrebné počkať, kým sa skončí testovanie a výsledky sa zapíšu do klientskeho programu. A bude možné znova zadať nové množstvo odoslaných údajov.

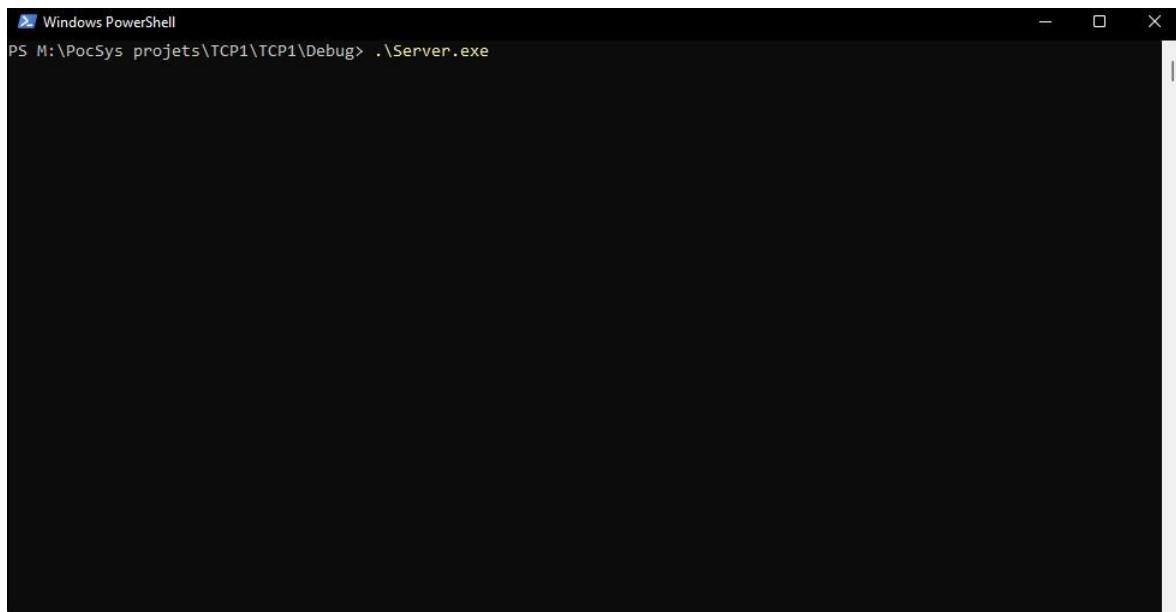
Aby správne spustil programy protokolov TCP a UDP je potrebné dodržať nasledujúce kroky:

1. Zapamätať cestu k súborom týchto programov
2. Otvoriť dvakrát(pre DIM tri krát) ľubovoľný terminál (napríklad: **“cmd”** alebo **“PowerShell”**). Postup na otvorenie príkazového riadku(**cmd**) :
 - Stlačením **“Win + R”** otvorí sa okno **“Spustiť”**
 - Napísať do riadku **“cmd”**

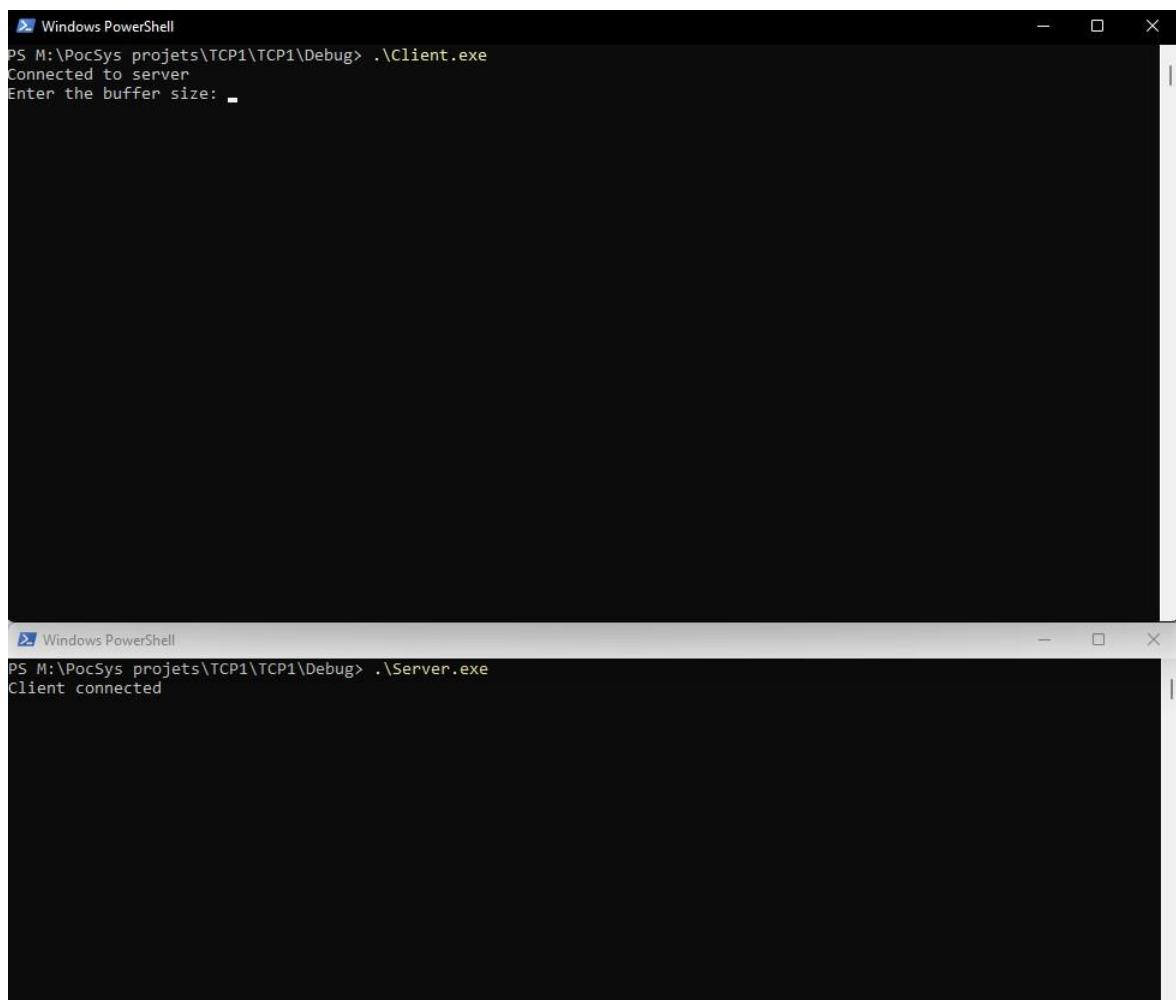
- Kliknúť na “OK”
3. Pomocou príkazu “cd” prejsť do priečinka “.\Debug”(pre DIM “.\bin32”) v oboch (pre DIM v troch) termináloch, ako je znázornené na obrázku 1 (pre DIM obrazok 5)
 4. **Tento krok je len pre protokol DIM!** Na jednom z terminálov spustiť program „dns” potrebný na vytvorenie spojenia, ako je znázornené na obrázku 6
 5. V jednom z terminálov spustiť serverovú aplikáciu pomocou príkazu “.\Server.exe”, ako je znázornené na obrázku 2.
Poznámka: terminály na spustenie programu môžu používať rôzne príkazy, napríklad: v príkazovom riadku (cmd) stačí zadať názov súboru “Server.exe”, ale v PowerShell treba zadať “.\Server.exe”
 6. Spustiť klientsky program v inom termináli.
Ak všetko prebehlo v poriadku, v termináli, na ktorom beží server, napíše „Klient je pripojený (Client connected)” (iba v TCP) a v termináli s klientským programom napíše: „Pripojený k serveru (Connected to server)“, „Zadajte veľkosť bufferu: (Enter the buffer size:)”. Zobrazené na obrázku 3
 7. Zadať veľkosť bufferu a počkať, kým sa test neskončí. Výsledky sa zapíšu do terminálu so spusteným klientským programom, ako je uvedené na obrázku 4
 8. Potom veľkosť bufferu je možné zadať znova
 9. Aby vypnúť program, treba zatvoriť terminál alebo stlačiť klávesy "Ctrl + C"



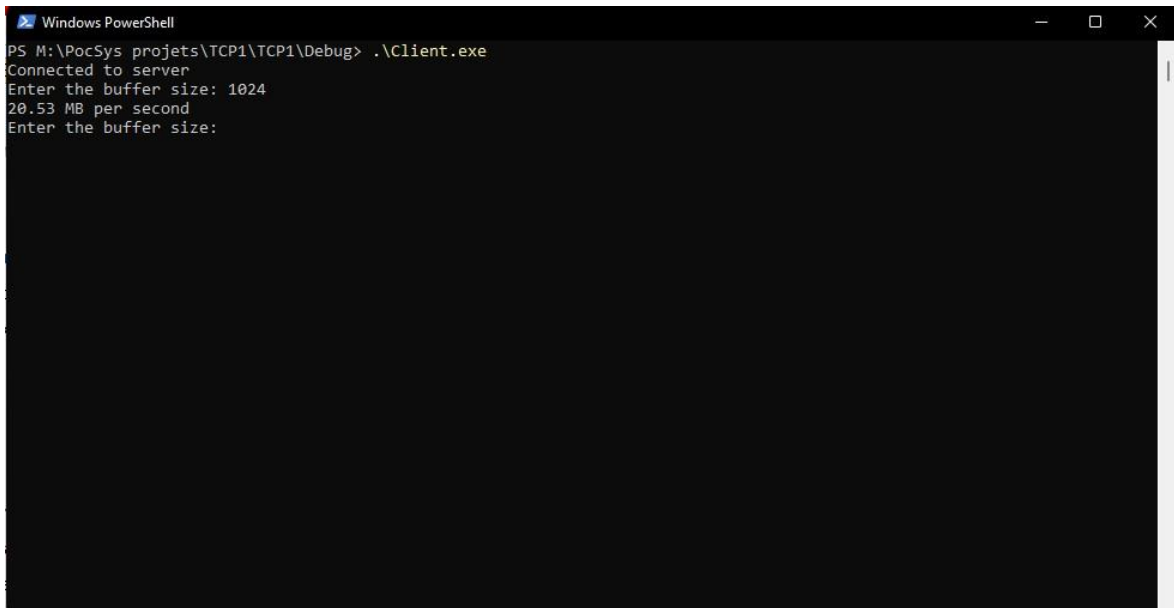
Obrázok 1 Zadanie požadovaného priečinka



Obrázok 2 Spustenie servera

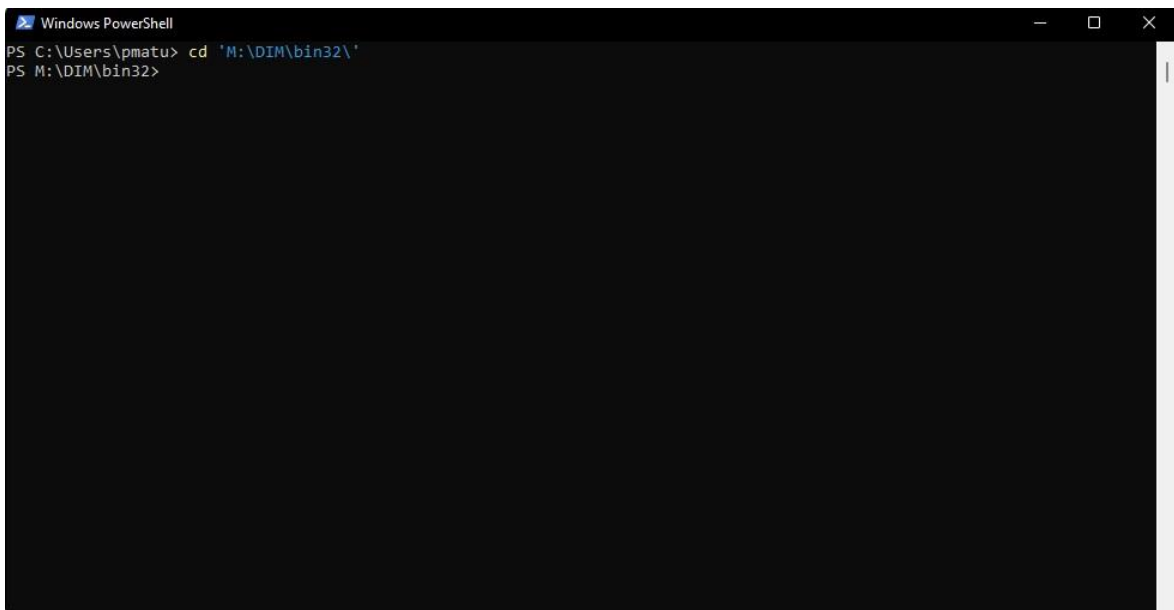


Obrázok 3 Spustenie klienta



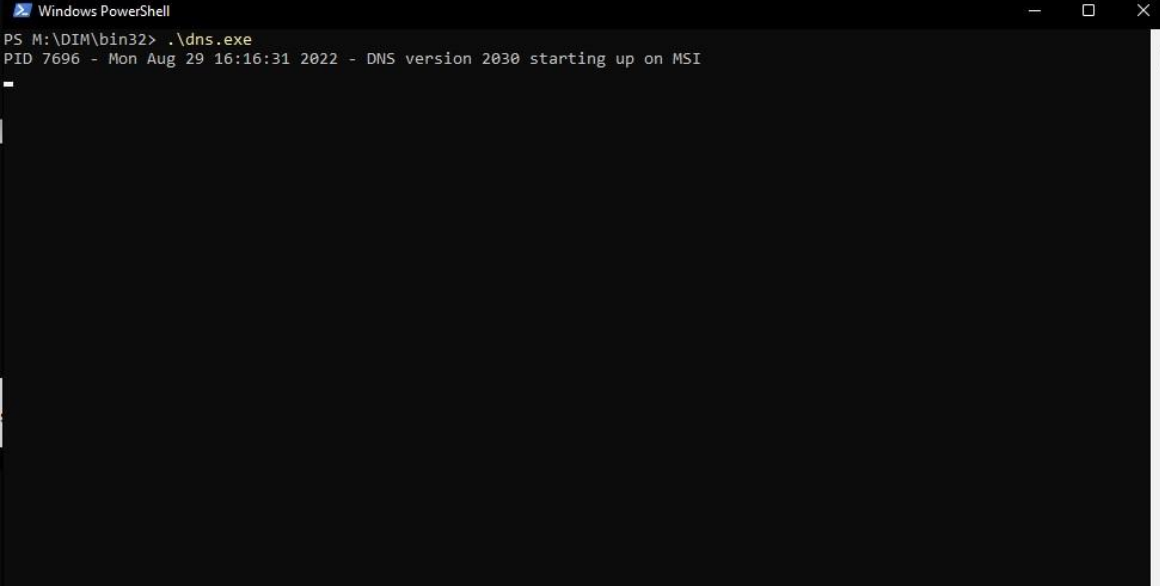
```
Windows PowerShell
PS M:\PocSys\projets\TCP1\TCP1\Debug> .\Client.exe
Connected to server
Enter the buffer size: 1024
20.53 MB per second
Enter the buffer size:
```

Obrázok 4 Vypisovanie výsledkov testu



```
Windows PowerShell
PS C:\Users\pmatu> cd 'M:\DIM\bin32\'
PS M:\DIM\bin32>
```

Obrázok 5 Zadanie požadovaného priečinka(DIM)



```
Windows PowerShell
PS M:\DIM\bin32> .\dns.exe
PID 7696 - Mon Aug 29 16:16:31 2022 - DNS version 2030 starting up on MSI
```

Obrázok 6 Spustenie DNS

Záver

Táto používateľská príručka informuje bežného používateľa o tom, ako používať aplikáciu na meranie priepustnosti a aplikáciu na meranie odozvy. Zároveň informuje o tom, ako programy správne spúšťať, ako ich používať a kde nájsť požadované výsledky.

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

Obsah

Úvod	85
1 Použité knižnice.....	85
2 Funkcie, triedy, premenné a iné.....	85
2.1. Potrebne funkcie, premenné a triedy knižnice <WinSock2.h>:	85
2.2. Potrebne funkcie knižnice <dis.hxx>:	87
2.3. Potrebne funkcie knižnice <dic.hxx>:	88
2.4. Potrebne funkcie knižnice <memory.h>:	88
2.5. Potrebne funkcie knižnice <chrono>:	88
2.6. Potrebne funkcie knižnice <stdio.h>:	88
2.7. Potrebne funkcie knižnice <string>:	88
2.8. Potrebne funkcie knižnice <iostream>:	89
2.9. Príkazy na deaktiváciu veľkého počtu upozornení a chýb počas kompilácie:	89
2.10. Potrebne funkcie knižnice <ServerLib.h>:	89
2.11. Potrebne funkcie knižnice <ClientLib.h>:	89
2.12. Iné:	90
3 Softvérové moduly	90
3.1. Vlastné knižnice	90
3.2. Spojenie klient-server	93
3.2.1. TCP	93
3.2.2. UDP	94
3.2.3. DIM	95
3.3. Modul servera	95
3.3.1. TCP Server	95
3.3.2. UDP Server	96
3.3.3. DIM Server	98
3.4. Modul klienta pri meraní priepustnosti	98

3.4.1.	TCP	98
3.4.2.	UDP.....	99
3.4.3.	DIM.....	101
3.5.	Modul klienta pri meraní odozvy	102
3.5.1.	TCP.....	102
3.5.2.	UDP.....	103
3.5.3.	DIM.....	105
Záver.....		107

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 prácu 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. Potrebne 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
- Funkcia **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 príjemcu 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. Potrebne 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 servera
- 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. Potrebne 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. Potrebne 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. Potrebne 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. Potrebne 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 bufferu do premennej vo zvolenom formáte

2.7. Potrebne 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()** - Vrátí ukazovateľ na pole, ktoré obsahuje sekvenciu znakov ukončenú nulou (t. j. C-reťazec) predstavujúcu aktuálnu hodnotu objektu reťazca.

2.8. Potrebne funkcie knižnice <iostream>:

- Objekt **cout** - objekt triedy ostream, s ním je možné odosielať údaje do konzoly
- Objekt **cin** - objekt triedy istream, 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. Potrebne 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. Potrebne 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 vyzeráť 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
    WSASStartup(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 začneme 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 zapnúť tento server. Funkcia "**void rpcHandler**" je hlavný cyklus 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 zeros
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.