

DEBRECENI EGYETEM

TDK DOLGOZAT

DEBRECEN

2021

DEBRECENI EGYETEM
INFORMATIKAI KAR
KOMPUTERGRAFIKA ÉS KÉPFELDOLGOZÁS TANSZÉK

ADATÁTVITELI TECHNIKÁK AZ FMFT SZOFTVERBEN

BÍRÁLATRA VALÓ BEKÜLDÉSÉHEZ HOZZÁJÁRULOK:

DR. TORNAI RÓBERT
TÉMAVEZETŐ

KISS-IMRE DALMA
PROGRAMTERVEZŐ INFORMATIKUS

DEBRECEN

2021

Tartalomjegyzék

BEVEZETÉS	1
1. A FELHASZNÁLT TECHNIKÁK	3
1.1. Adatátviteli technikák	3
1.1.1. TCP csatornán történő átvitel	3
1.1.2. UDP csatornán történő átvitel	4
1.1.3. UDP átviteli csatorna kontroll csatornával	4
1.2. Fejlesztőkörnyezet	4
1.2.1. Qt keretrendszer	5
1.2.2. C++ programozási nyelv	5
1.2.3. Boost C++ könyvtár	5
2. MEGVALÓSÍTOTT MEGOLDÁSOK	7
2.1. SCTP	7
2.2. FMFT kliens	8
2.3. FMFT szerver	12
2.4. Párhuzamosítás	14
2.5. Ütemezés	15
2.6. Titkosítás	16
2.7. CRC vizsgálat	17
2.8. Logolás	18
2.9. Paraméterezés	19
3. ÁTVITELI TESZTEK	23
3.1. Csomagméret tesztek	23

3.2. Csomagméret tesztek CRC vizsgálattal	24
3.3. Csomagméret tesztek titkosítással	25
3.4. Átlagsebesség a különböző technikák esetén	26
EREDMÉNYEK	29
TOVÁBBFEJLESZTÉSI LEHETŐSÉGEK	31
IRODALOMJEGYZÉK	33
KÓDJEGYZÉK	37
ÁBRÁK JEGYZÉKE	39
SAJÁT MUNKA LEÍRÁSA	41
KÖSZÖNETNYILVÁNÍTÁS	43

Bevezetés

A dolgozatom során célom egy fájlátviteli rendszer készítése. Manapság gyakori, hogy adathordozók helyett hálózaton továbbítunk adatokat illetve fájlokat, legfőképp amikor nagy mennyiségű adatról beszélünk illetve nagy távolságokról. Létező megoldás erre a File Transfer Protocol (FTP), illetve a UDP-based Data Transfer Protocol (UDT) [1]. Egy ezektől különböző, gyors és hatékony megoldást készítettem és fogok leírni.

A munka során feltérképeztem a különböző fájlátviteli protokollok sajátosságait [2]. Előnyeik kihasználásával tervezem megtalálni a legjobb stratégiát a fájlok átviteléhez [3]. Ehhez készítettem egy szerver-kliens programpárt, melyben a kliens képes lesz fájlokat továbbítani a szerver részére és állományokat lekérni onnan. Leteszteltem az FTP, SFTP, SCTP, UDP és RBUDP protokollokat, ezek alapján választottam ki a legmegfelelőbbet [4]. Qt használatával elérhetővé teszem a program Linux és Windows operációs rendszereken való használatát. Ezen felül célom, hogy WebAssembly segítségével a kliens akár böngészőből is futhasson. Miután ezeket megvalósítottam, továbbfejlesztettem a programomat CRC vizsgálattal a csomagok sérülésmentességének teszteléséhez. Titkosítást adtam hozzá a szoftveremhez a biztonság növelése érdekében. Ezentúl a hatékonyságot párhuzamosítással növelem, a csomagütemezés segítségével pedig felülről korlátos sebességű küldést teszek lehetővé.

A programom segítségével egyszerűen és megbízhatóan lehet fájlokat továbbítani a kívánt hálózatra csatlakozó gépre. Ez megkönnyíti a nagy mennyiségű adatok továbbítását. A való életben akár a gigabites kapcsolatokon is lelassulhat az átvitel egy pár megabites tartományra a forgalmas hálózatokon, valamint nagy mennyiségű elveszett csomagot tapasztalhatunk. Például orvosi kutatásokhoz nem ritka, hogy akár 60 TB adatra van szükség. A Gyires szuperszámítógép termében lévő virtuális szerveren tesztelve az ilyen nagy mennyiségű bemeneti adat több hetes átviteli időt is igényelhet, sokszor a vissza-

küldendő eredmény adat is ilyen méreteket vehet fel. Ezért fontos, hogy az átvitel minél nagyobb sebességen történjen, miközben kiszámítható, szabályozható és stabil marad.

1. fejezet

A felhasznált technikák

Ebben a fejezetben felsorolom a megvizsgált illetve felhasznált hálózati technikákat. Ismertetem a fejlesztéshez felhasznált fejlesztői környezetet, a választott programozási nyelvet és a hozzá tartozó Boost könyvtárat. Továbbá megindoklom, hogy miért ezeket választottam a szoftverem elkészítéséhez és miben lesznek nagy segítségemre.

1.1. Adatátviteli technikák

A fájlátviteli rendszer megírásához, illetve az adatátvitel megvalósításához a legelső döntés a felhasználandó technikák kiválasztása. Az átvitelhez a két alapvető megoldás az UDP és a TCP protokollok. Mindkettőnek vannak előnyei és hátrányai, ezekről részletesen írok ebben a fejezetben.

1.1.1. TCP csatornán történő átvitel

Első lehetséges megoldásként a Transmission Control Protocol került előtérbe, mely egy kapcsolat orientált adatátviteli protokoll. Előnye a megbízható és stabil kapcsolat, akár megbízhatatlan hálózati összeköttetés esetén is. Hiba után azonban lassan áll vissza magasabb értékre az átviteli ráta. Ezáltal a sebesség várhatóan nem lesz megközelítőleg sem egyenletes, sem maximális. Ezen alapul a File Transfer Protocol (FTP), mely talán az egyik legismertebb fájlátviteli protokoll. Tesztelés során látható, hogy a sebessége ingadozó, az átviteli rátájának átlaga alacsony. Másik ígéretes protokoll a Secure File Transfer Protocol (SFTP), melyben a titkosítás is tovább lassítja az átvitelt.

1.1.2. UDP csatornán történő átvitel

Második alapvető megoldás a User Datagram Protocol, mely csomag alapú küldést tesz lehetővé. Mivel nem vár választ a fogadótól, így ez a sebességet nem korlátozza. Hátránya, hogy az átvitel nem megbízható, mert nincs a megérkezett csomagokról visszajelzés [5]. Az UDP teljesítményét sokan vizsgálták már előttem [6]. Ezt a protokollt használja fel az UFTP (Encrypted UDP based FTP with multicast), mely a kutatómunkám kezdetén nem rendelkezett grafikus felhasználói felületet tartalmazó stabil verziójú klienssel.

Az UDP protokollt már 2002-ben is próbálták felhasználni nagy fájlok gyors átvitelére, így jött létre a Reliable Blast UDP [7]. Az RBUDP hátránya, hogy a küldendő állományt az `mmap` segítségével olvassa be a memóriába kezdő lépésként. Ez nagy memória terheléshez és hosszú előolvasási időhöz vezet. Szeletenként olvasva és írva a fájlt, a művelet átlapolódik a hálózati átvittel a megoldásomban. A chunk méretet 512 byte többszörösével teszteltem, hogy igazodjon a tároló egységek alapértékeihez. Így SSD-k esetén minimalizálni lehet a túlírást (write amplification). A tesztek alapján két optimális értéket találtam: 4096 byte és 8192 byte. Kisebb értékek esetén túl sok a küldendő csomag (általában az operációs rendszer rendelkezik egy felső korláttal), nagyobb esetén egyetlen CRC vagy átviteli hiba miatt túl sok az újraküldendő adat.

1.1.3. UDP átviteli csatorna kontroll csatornával

Az UDP hiányosságait pótolva, megvizsgáltam egy kevert megoldást melyben az adatátvitel egy UDP csatornán valósul meg, illetve egy TCP csatornán vele párhuzamosan érkezik a válasz az érkezett csomagokról. Bár így megfelelően megbízható az átvitel a TCP kontroll csatorna miatt lassú a visszajelzés. Végül eljutottam a végleges megoldásunkhoz, melyben az adatátvitel és a visszajelzés egy-egy külön UDP csatornán zajlik, mellyel stabilabb és gyorsabb adatátvitel valósítható meg.

1.2. Fejlesztőkörnyezet

A fejlesztés megkezdésekor kitűzött cél volt, hogy a program több operációs rendszeren is használható legyen, illetve az adat szállítás lehetséges legyen különböző rendszerek között

is. Ennek okán a választásom a Qt keretrendszerre esett, így lehetőségem van a platform-független fejlesztésre, a szoftver elérhető lesz Windowson, Linuxon és macOS-en, emellett mobil operációs rendszereken, mint az Android és iOS [8]. Továbbá a sok lehetőséget rejtő WebAssemblyt is támogatja a Qt [9], mellyel futtatható lesz webböngészőből is a szoftver [10].

1.2.1. Qt keretrendszer

A Qt rendszer mellett szólt az is, hogy a Qt-hez elérhetőek a Qt Creator és a Qt Designer kiegészítők. Ezek több platformon elérhető integrált fejlesztő környezetek, melyek használhatók Windows, Linux és macOS operációs rendszereken is. A Qt Creator a fejlesztést nagyban segíti például az automatikus kiegészítési javaslatokkal, a szintaktikai hibák detektálásával, valamint érzékeli a hiányzó, szükséges osztályokat és segítséget nyújt az importálásban is. Ezeken túl külön fordítási lépés nélkül azonnal jelzi az éppen bevitt kóddal kapcsolatos **warning** és **error** üzeneteket. Mindemellett az integrált nyomkövető rendszer is nagyon hatékony. A Qt Designer segítségével egyszerűen hozhatunk létre szép és könnyen használható funkcionális grafikus felületi elemeket és teljes felületet is [11]. Továbbá a Qt Style Sheet alkalmazásával, melyet a HTML CSS inspirált és ahhoz nagyon hasonlóan működik, a kinézet teljes mértékben testreszabható.

1.2.2. C++ programozási nyelv

Programozási nyelvnek a C++-t választottam, ami egy általános programozási nyelv. A C nyelv kiterjesztett változata, lehetőség van benne osztályok létrehozására, ezáltal objektum orientált módon használható. Jellemző rá a hordozhatóság, illetve alacsony és magas szintű programozásra is alkalmas, ami nélkülözhetetlen a legjobb teljesítmény elérése érdekében. A Qt-hoz az alapvető választás a QML lehetett volna, ám mivel ez egy szkriptnyelv, így nem elég hatékony a nagy sebességű átvitelhez [12].

1.2.3. Boost C++ könyvtár

A fejlesztés során nagy segítségemre lesz a Boost könyvtárrendszer, mely sok problémára hatékony és gyors megoldást biztosít [13]. Például a csomagokat, melyek már el lettek

küldve és érkezett róluk visszajelzés – miszerint megérkeztek –, el kell tárolni, hogy ne legyenek többször elküldve. Így számomra a tároláshoz használt adatszerkezet nagyon fontos, ugyanis nagy fájlok esetén sok a velük kapcsolatosan tárolandó adat. Ehhez szükséges egy bit-vector, melyre a Boost rendelkezik egy megfelelően optimális implementációval [14].

2. fejezet

Megvalósított megoldások

Ebben a fejezetben a létrehozott Fast Manager of File Transfer (FMFT) program pár legfontosabb jellemzőit ismertetem. A fejlesztés során teszteltem az SCTP protokollt [15]. A párhuzamosítással a többmagos processzorokat tudom hatékonyabban terhelni. Az ütemezés finomításával el lehet kerülni a bufferek túlterhelését. A titkosítással növelem a küldés biztonságosságát, valamint a CRC vizsgálattal az átvitel megbízhatósága javítható. A logolás segítségével vizsgálhatóvá tettem a program által elért teljesítményt. Végül a parancssori paraméterek segítségével a tesztelést tettem hatékonyabbá.

2.1. SCTP

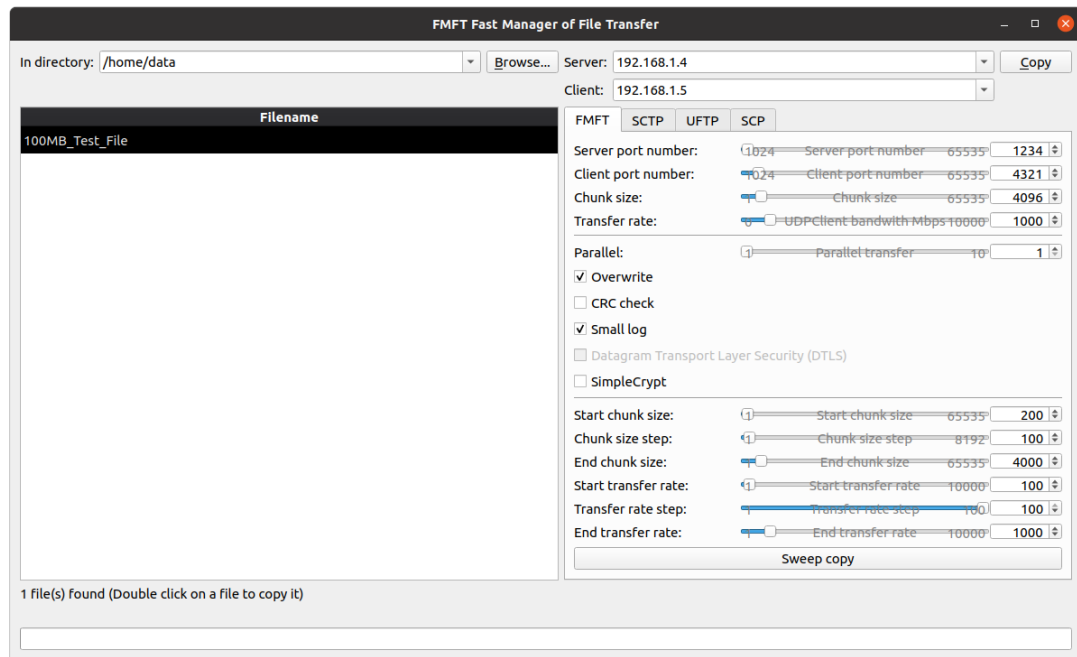
A kutatási munkát a Qt hálózati mintaprogramjainak tesztelésével kezdtem. Több jó kiindulási lehetőséget találtam köztük, de legígéretesebbnek a MultiStreamServer és a MultiStreamClient pár tűnt, mely SCTP (Stream Control Transmission Protocol) átvitelt használ [16]. Az első fordításkor hibát kaptam miszerint a QSctpSocket osztály nem található. Ez a probléma megoldódott a Qt rendszer forrásból való fordításával az SCTP csomag engedélyezésével Linux operációs rendszeren. Utánajárva a problémának, általában a statikus buildet javasolják, bár ez a fejlesztési mód lassúnak minősült a projekt fordításakor. A futtatható állományok 400 MB körüli méretűek voltak, melyek túl nagyoknak bizonyultak a tesztelőkkel történő kommunikáció során. Statikus fordításra egy program kiadásakor van szükség, így a szoftver nem tartalmaz függőségeket a Qt felé. Viszont a generált bináris még így is rendelkezett függőséggel az operációs rendszer felé. A

későbbiekben további kódoptimalizálással kapcsolatos argumentumok módosítására volt szükség a static flag elhagyása mellett, így a Qt illetve a projekt fordítása is gyorsabbá vált az SCTP támogatás mellett.

Miután fordítottam egy alkalmas Qt rendszert, mellyel sikerült tesztelni a két példa programot, felfedeztem a bennük rejlő lehetőségeket. A MultiStreamServer és a MultiStreamClient három egymástól független csatornát tartalmaz, három különböző szolgáltatáshoz: ezek a Movie, a Time és a Chat. Ezt felhasználva hozzáadtam egy negyediket, mellyel már fájlok küldésére is van lehetőség. Teszteléskor kis fájlok átvitelénél a várt eredményeket kaptam, ám nagy fájlokkal való kísérlet során az eredmény erősen változó volt. Egy nanoszekundumos időmérő hozzáadásával vizsgálható lett az átviteli sebesség, melyet a szerver és a kliens oldalon egy-egy napló fájlba írtam ki, illetve a konzolról is leolvasható. Ezeket a napló fájlokat felhasználva lehet megtervezni a további fejlesztéseket, és optimalizálásokkal bővíteni a programokat.

2.2. FMFT kliens

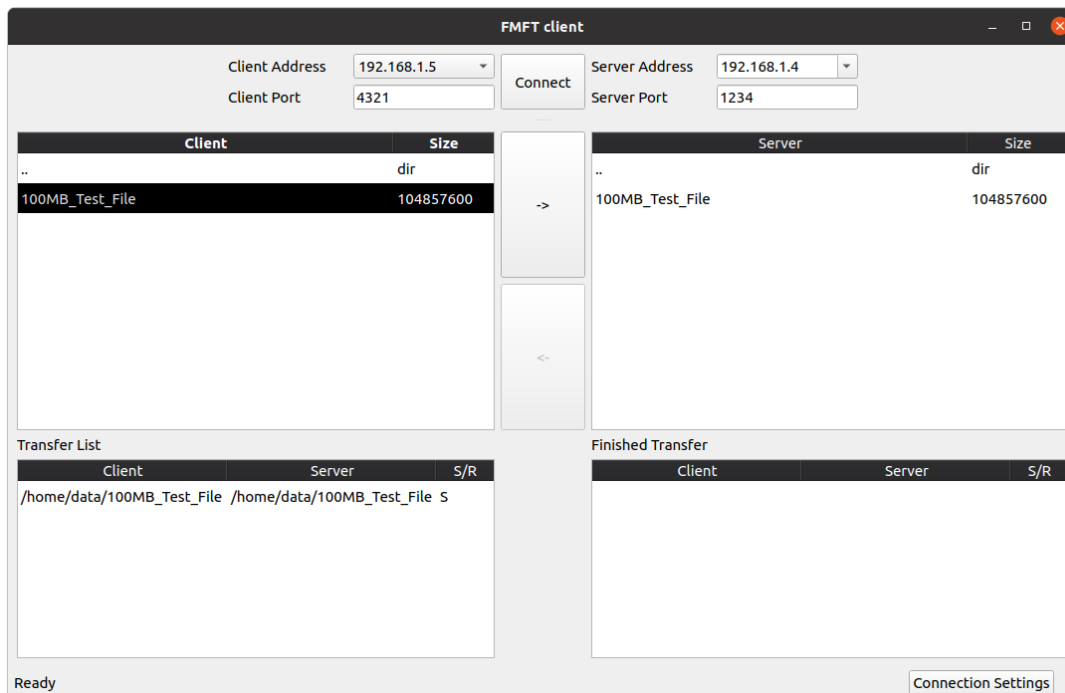
A 2.1-es ábrán látható az elkészült tesztfelület, mely segítségével egy vagy több küldést is végrehajthatunk. Lehetőség van megadni a szerver IP címét és portját valamint azt a hálózati lábat, melyen a csomagokat küldeni kívánjuk. Kiválaszthatjuk a mappát, melyből fájlt szeretnénk továbbítani, majd pedig magát a fájlt is. A tabok segítségével lehetőség van az implementációk közül választani, melyikkel akarjuk az átvitelt megvalósítani. Duplán kattintva a fájlra vagy a Copy gombbal a kiválasztott fájl egyszer kerül továbbításra a megadott beállításokkal, ebben az esetben a csomagméretet és a sebességkorlátot van lehetőség kiválasztani. A Sweep copy gomb felel a tesztelésért, ebben az esetben a kiválasztott fájl többször kerül továbbításra. A megadott minimális és maximális csomagméret között a megadott lépésközzel minden mérettel továbbítja egyszer a fájlt, így lehetőség van összehasonlítani a különböző csomagméretekkel való küldések közti sebességkülönbséget. Továbbá az átviteli rátának is megadhatunk egy minimumot, egy maximumot és egy lépésközt. Ez esetben pedig vizsgálhatjuk például, hogy mi az a maximális átviteli ráta, amivel stabil marad az átviteli sebesség. Ezen felül állíthatjuk, hogy a szerver oldalán felülíródjon-e a fájl amikor az már létezik, hogy akarunk-e CRC vizsgálatot, titkosítást,



2.1. ábra. FMFT kliens teszt felülete

illetve, hogy részletes vagy rövid naplózást hajtsunk-e végre az átviteleknél.

A felületet átalakítottam a továbbfejlesztési lehetőségek tekintetbe vételével, ez látható a 2.2-es ábrán. Így sokkal inkább hétköznapi használatra van kialakítva, mint teszteléshez. A bal oldalon a kliens, a jobb oldalon pedig a szerver fájlrendszere böngészhető. A kliens oldalon kiválasztott fájl a szerver oldalon kiválasztott mappába lesz lementve az átvitel során. A küldést a jobbra mutató nyíl megnyomásával indíthatjuk, az adott fájl bekerül az átviteli listába. A listából mindig az elsőnek bekerült elem lesz elsőként továbbítva, majd amikor sikeres az átvitel, bekerül a jobb oldalon látható befejezett átviteli listába. Csak akkor kerül többször továbbításra ugyanaz a fájl többszöri gombnyomás esetén, ha a cél hely vagy a forrás hely különböző. Bár az új kliens még fejlesztés alatt áll, alapbeállítások mellett képes már az állományok átvitelére. A továbbiakban tervezem a fájlátviteli módszerek beállításainak a lehetőségét hozzáadni a menühöz, illetve a szerver felől a kliens felé történő vissz irányú átvitelt is megvalósítani. Továbbá a régi klienssel szemben lehetőségünk van akár teljes mappákat is kijelölni átvitelre, ebben az esetben a mappákon belül található összes fájl, beleértve az almappákban lévőket is, felvételre kerül a küldési listába. Több állomány átvitele során a kapcsolatfelvételeket és az adatcsomagok küldését át tudjuk lapolni, így sokkal hatékonyabbá válik a kombinált átvitel. Emellett



2.2. ábra. FMFT kliens új felülete

nem feltétlenül kell megvárni, míg egy fájl összes csomagja átmegy több menetben mielőtt új állomány küldésébe kezdenénk, hanem az egyes átviteli munkamenetekben az összes állományt egymás után kezelhetjük. Így az egymás utáni csomagküldési sorozatok között több idő van a visszajelzéseket megvárni, ami kevesebb újraküldött csomaghoz és ezáltal jobb átviteli teljesítményhez vezet.

A fájl elküldéséért a `SendFile` függvény felel, melynek forráskódja szerepel a 2.1 kódban. Konzolos kliens esetén ez egyszer hívódik meg, míg a grafikus felület esetén pedig minden sorban következő átvitelhez meghívódik. Paraméterként megkapja a küldendő fájl nevét az elérési úttal, a szerver IP címét, a szerver portját és az átviteli rátát. A kapott argumentumok alapján itt történik meg a kapcsolatfelvétel a szerverrel, továbbá itt állítom be az átvitel adatait valamint azokat az elemeket, melyek a fejlécbe kerülnek. Olvashatóként nyitom meg az átvinni kívánt fájlt, mivel nem lesz szükség a módosítására. Elindítom az időmérést, ez alapján lesz kiszámítva az átvitel átlagsebessége. Végül, miután minden elő lett készítve az átvitelhez, az `m_Scheduler` időzítő lejártára állított signal segítségével a `scheduleSingleControl` slot meghívódik, melynek során a küldés megvalósul. Itt történik meg a fájl feldarabolása és az átviteli sebesség eléréséhez szükséges csomagok száma alapján a küldendő csomagok kiválasztása és elküldése is.

```

1 bool CClient::SendFile(QString fileName, QString serverName, quint16 serverPort,
2                       quint16 transferRate)
3 {
4     if (m_bIsTransferring)
5         return false;
6
7     m_dTransferRate      = transferRate;
8     m_ui32SentTotal      = 0;
9     m_ui32ReceivedTotal  = 0;
10    m_ui32CRCErrorNum     = 0;
11    m_ui32NextSeek       = 0;
12    m_ui32SliceIndex      = 0;
13
14    m_ui32FileTransferID++;
15
16    m_ServerHostAddress = QHostAddress(serverName);
17    m_ui16ServerHostPort = serverPort;
18    m_ui16ServerDataPort = serverPort + 1;
19
20    m_ControlSocket.connectToHost(serverName, serverPort + 0);
21    m_DataSocket.connectToHost(serverName, serverPort + 1);
22    m_ControlSocket.waitForConnected(1000);
23    m_DataSocket.waitForConnected(1000);
24
25    m_sDataFileNameToSend = fileName;
26    m_DataFile.setFileName(m_sDataFileNameToSend);
27
28    bool fileOpened = m_DataFile.open(QIODevice::ReadOnly);
29
30    m_ui32ChunkNumber = m_DataFile.size() / FMFT_UDP_SIZE +
31                        ((m_DataFile.size() % FMFT_UDP_SIZE) > 0 ? 1 : 0);
32    m_ui32SlicesToSend = m_ui32ChunkNumber;
33
34    m_StartDateTime = QDateTime::currentDateTime();
35    m_Timer.start();
36
37    m_bSlices.clear();
38    m_bSlices.resize(m_ui32ChunkNumber, false);
39
40    *m_pSBFileTransferID = qToBigEndian(m_ui32FileTransferID);
41    *m_pSBPacketID = FMFT_CONTROL_ID;
42    *m_pSBChunkNum = qToBigEndian(m_ui32ChunkNumber);
43    m_baFileName = fileName.toUtf8();
44    int size = m_baFileName.size();
45
46    memcpy(m_pSBFileName, m_baFileName.data(), size);
47
48    m_ui64HeaderSize = FMFT_CONTROL_HEADER_SENT_SIZE + size;
49    *m_pSBCRC = qToBigEndian(qChecksum(m_pSBCRCLoad, m_ui64HeaderSize - FMFT_CRC_SIZE));
50
51    connect(&m_Scheduler, &QTimer::timeout, this, &CClient::scheduleSingleControl);
52
53    return true;
54 }

```

2.1. kód. Kliens oldali átviteli függvény

2.3. FMFT szerver

A szerver program alapvetően eseménykezelésen alapul. Várakozik mindaddig, míg nem kap kapcsolatfeltevő csomagot egy kientstől, vagy folyamatban lévő átvitel esetén nem kap egy adatcsomagot. Az első ami megtörténik a futtatáskor, hogy elindul a kontroll csatornán a kapcsolatfeltevő üzenetekre való várakozás. Amennyiben ez megtörténik, meghívódik a `readyReadSingleControlChannel`, ez látható a 2.2 kódrészletben. A slot felolvassa a kapott csomagot. Ha új kapcsolat felvétele történik, akkor beállítja az átvitelről kapott információkat és előkészíti az átvitelt az alapbeállításokkal.

```

1 void CServer::readyReadSingleControlChannel()
2 {
3     while (m_ControlSocket.hasPendingDatagrams())
4     {
5         m_i64DataRead = m_ControlSocket.readDatagram(m_ReceiveBuffer, FMFT_MAX_UDP_SIZE,
6             &m_SenderHostAddress, &m_ui16SenderHostPort);
7
8         if (m_bIsTransferring == true)
9         { // connection established
10
11             *m_pSBFileTransferID = *m_pRBFileTransferID;
12             *m_pSBPacketID = FMFT_CONTROL_ID;
13             *m_pSBCRC = qToBigEndian(qChecksum(m_pSBCRCLoad, FMFT_ID_SIZE));
14             m_ControlSocket.writeDatagram(m_SendBuffer, FMFT_DATA_HEADER_SIZE,
15                 m_SenderHostAddress, m_ui16SenderHostPort);
16         }
17         else
18         { // new connection
19             m_Timer.start();
20
21             *m_pSBFileTransferID = *m_pRBFileTransferID;
22             *m_pSBPacketID = FMFT_CONTROL_ID;
23             *m_pSBCRC = qToBigEndian(qChecksum(m_pSBCRCLoad, FMFT_ID_SIZE));
24
25             m_ControlSocket.writeDatagram(m_SendBuffer, FMFT_DATA_HEADER_SIZE,
26                 m_SenderHostAddress, m_ui16SenderHostPort);
27             m_ui64TimeStamp = m_Timer.nsecsElapsed() / 1000;
28
29             m_DataFileNameToReceive = QString::fromUtf8(m_pRBFileName,
30                 m_i64DataRead -
31                 FMFT_CONTROL_HEADER_SENT_SIZE);
32             m_ui32ChunkNum = qFromBigEndian(*m_pRBChunkNum);
33             m_bIsTransferring = true;
34             m_ui32StoredFileTransferID = m_ui32IncomingFileTransferID;
35             m_ui32CRCErrorNum = 0;
36             m_ui32ReceivedPackets = 0;
37             m_ui32NextSeek = 0;
38             m_ClientHostAddress = m_SenderHostAddress;
39             m_ui16ClientHostPort = m_ui16SenderHostPort;
40             m_ui16ClientHostDataPort = m_ui16ClientHostPort + 1;
41             m_ui32CountDown = m_ui32ChunkNum;
42             m_ui32ClientIPv4 = m_ClientHostAddress.toIPv4Address();
43
44             m_DataSocket.connectToHost(m_ClientHostAddress, m_ui16ClientHostDataPort);
45
46             m_bSlicesArrived.clear();
47             m_bSlicesArrived.resize(m_ui32ChunkNum, false);

```

```

48     QStringList      list = m_DataFileNameToReceive.split("/");
49     QString          fileNameNoPath = list.last();
50
51
52     m_DataFile.setFileName(fileNameNoPath);
53
54     if (m_DataFile.exists())
55         m_DataFile.remove();
56
57     bool fileOpened = m_DataFile.open(QIODevice::WriteOnly);
58
59     m_StartDateTime = QDateTime::currentDateTime();
60 }
61 }
62
63 }

```

2.2. kód. Szerver oldali kontroll csatornán érkező csomagok kezelése

A másik nagy feladata a szervernek az átvitelkor érkező csomagok kezelése az adatcsatornán. Ezért felel a 2.3-as kódrészletben látható `readyReadSingleDataChannel`. Egy while ciklusban történik a folyamat, ami addig fut míg van olvasatlan csomag. Vizsgálom, hogy az érkezett csomag attól a küldőtől származik-e aki elindította az átvitelt, illetve hogy az átviteli azonosító ehhez az átvitelhez tartozik-e. Ezek után ellenőrzöm, hogy az adott csomag érkezése rögzítve lett-e már. Ha megfelel a csomag az elvártaknak, akkor rögzítem a megérkezését és erről választ küldök a kliensnek. Végül az utolsó csomag megérkezése után bezárom a fájlt.

```

1 void CServer::readyReadSingleDataChannel()
2 {
3     while (m_DataSocket.hasPendingDatagrams())
4     {
5         m_i64DataRead = m_DataSocket.readDatagram(m_ReceiveBuffer, FMFT_MAX_UDP_SIZE,
6             &m_SenderHostAddress, &m_ui16SenderHostPort);
7
8         m_ui32PacketID = qFromBigEndian(*m_pRBPacketID);
9         *m_pSBFileTransferID = *m_pRBFileTransferID;
10        *m_pSBPacketID = *m_pRBPacketID;
11        *m_pSBCRC = qToBigEndian(qChecksum(m_pSBCRCLoad, FMFT_ID_SIZE));
12
13        // ... here comes the packet checking ...
14
15        if (m_bSlicesArrived[m_ui32PacketID] == false)
16        { // eliminates multiple arrive registration
17            m_i64DataRead -= FMFT_DATA_HEADER_SIZE;
18
19            m_DataSocket.writeDatagram(m_SendBuffer, FMFT_DATA_HEADER_SIZE,
20                m_ClientHostAddress, m_ui16ClientHostDataPort);
21
22            m_ui32ReceivedPackets++;
23
24            if (m_ui32NextSeek != m_ui32PacketID)
25            {
26                m_DataFile.seek((qint64)FMFT_UDP_SIZE * (qint64)m_ui32PacketID);
27                m_ui32NextSeek = m_ui32PacketID;
28            }
29
30            m_DataFile.write(m_pRBLoad, m_i64DataRead);

```

```

31         m_ui32NextSeek++;
32         m_bSlicesArrived[m_ui32PacketID] = true;
33         m_ui32CountDown--;
34     }
35     else
36     {
37         m_DataSocket.writeDatagram(m_SendBuffer, FMFT_DATA_HEADER_SIZE,
38                                     m_ClientHostAddress, m_ui16ClientHostDataPort);
39         continue;
40     }
41 }
42
43 if ((m_ui32CountDown == 0) && (m_bIsTransferring == true))
44 { // file shall be closed after last package
45     m_ui64TimeStamp = m_Timer.nsecsElapsed() / 1000000;
46     m_i64FileSize = m_DataFile.size();
47     m_DataFile.close();
48     m_bIsTransferring = false;
49 }
50 }
51 }

```

2.3. kód. Szerver oldali adatcsatornán érkező csomagok kezelése

2.4. Párhuzamosítás

A párhuzamosítás szükségessége elsőként a felhasználói felületen mutatkozott meg, ugyanis amíg a háttérben nagy mennyiségű munkafolyamat fut, addig a felhasználói felület elveszíti a responzivitását. Ha leválasztjuk a munkaszálról a grafikus felületet, a felület elemei – mint például a folyamatjelző sáv illetve a különböző nyomógombok – megfelelő visszajelzést tudnak adni. Ehhez nagy segítséget nyújt a Qt signal – slot rendszere. A következő lényeges változás a kapcsolat felállításának és az adatátvitelnek az elkülönítése volt szerver és kliens oldalon egyaránt. A szerver alkalmazásban ez lehetőséget biztosít arra, hogy egy időben több klienst is kiszolgáljak az egyes processzor magokon, illetve a maximálisan létrehozható szálak számának meghatározásával korlátozhatom a csatlakozó kliensek számát. Megfelelően erős CPU mellett egy mag több klienst is kiszolgálhat egy szálon. A kapcsolatok kezelését tovább optimalizáltam: a kapcsolat létrejövetele és az adatátvitel két külön porton zajlik, illetve minden porthoz külön szálát indítok el. Adott esetben az ugyanazon kienstől származó egymás utáni adatátvitel zavarhatják egymást amennyiben az első átviteltől kezdve a csomagok annyit késnek, hogy azok a második adatátvitel során érkeznek meg. A csomagkeveredés minimalizálása érdekében az adatátvitel 256 elemből álló adatport tartományt kapott. Ezek az elemek körkörösén kerülnek felhasználásra, így elegendő idő biztosított arra, hogy minden kóbor csomag

rendben megérkezessen. A 256 egy praktikus érték, ugyanis egy előjel nélküli bájtal lehet indexelni modulo értékek számolása nélkül. Ennek köszönhetően akár egy klientsől is indulhat több átvitel párhuzamosan ugyanazon szerver irányába.

2.5. Ütemezés

A munka során nagy hangsúlyt kapott a megfelelő csomag küldés ütemezés kiválasztása. Abban az esetben, ha folyamatosan küldjük a csomagokat a klientsől a szervernek, nagy mennyiségű lesz az elveszett csomag. Emellett a processzor használat is nagymértékben emelkedik valamint a hálózatot is túlságosan leterheljük [17].

Következő kísérletként egy maximális átviteli rátát határoztam meg, ehhez egy másodpercenként sorozatokat továbbít a kliens. Ez azt jelenti, hogy az előre meghatározott sebességnek megfelelő mennyiségű csomagot küld ki a csomagok méretét figyelembe véve. Majd ezután vár, amíg el nem indul a következő egy másodperces időszakasz. Így a hálózati pufferek túlságosan nagy terhelést kaptak szabályos időközönként.

Ezután az időintervallumot milliszekundumra módosítottam, ennek a hátulütője az volt, hogy az előre kiszámolt kiküldendő csomagok száma általában túl kicsinek minősült nagy csomagméretek esetén. Ennek következtében a kívánt küldési sebességet nem közelítette eléggé az elért sebesség. Ezzel szemben a processzor és a hálózat terhelése kiegyenlítettebbé vált.

Végso megoldásként a nanoszekundumos időzítés minősült kielégítőnek. A kliens megvizsgálja az előző egy másodpercben küldött csomagokat. Ha az elért sebesség alatta vagy fölötte van a kívánt sebességnek, akkor ezt korrigálva számítja a következő időintervallumban küldendő csomagok számát. Átlagosan ez egy extra csomag küldésének szükségességét jelenti. Ezzel a stratégiával jelentősen jobban lehet közelíteni az előre meghatározott átviteli sebességet. Így ugyan minimálisan nő a processzor terhelése, viszont minimalizálható a hálózat terhelése, mivel az elveszett csomagok száma is jelentősen csökkent. A `CLastSecond` osztály felel a csomagok ütemezéséért, ez szerepel a 2.4-es kódrészletben. Ennek segítségével követhető nyomon, hogy az elmúlt egy másodpercben mennyi brutto (fejléc + hasznos teher) adatot küldtem már át. Ez induláskor akár az egy másodpercnél kisebb időtartamokra is helyesen működik az időarányosítás segítségével.

```

1 class CLastSecond
2 {
3 public:
4     inline void append(ListElement element)
5     {
6         m_List.append(element);
7         m_ui64SizeSum += element.size;
8     }
9
10    inline void removeFirst(qint64 time)
11    {
12        while ((m_List.size() > 0) && (time - m_List.first().time > (qint64)1000000000))
13        {
14            m_ui64SizeSum -= m_List.first().size;
15            m_List.removeFirst();
16        }
17
18        if (m_List.size() > 0)
19        {
20            m_i64TimeFrame = time - m_List.first().time;
21            m_dMbps = (double)(m_ui64SizeSum * 8000) / (double)m_i64TimeFrame;
22            // (8000) = (8 bits * 1000 for nanosecs)
23        }
24        else
25        {
26            m_i64TimeFrame = 0;
27            m_dMbps = 0.0;
28        }
29    }
30
31    QLinkedList<ListElement>    m_List;
32    qint64                      m_i64TimeFrame;
33    quint64                     m_ui64SizeSum = 0;
34    double                      m_dMbps;
35 };

```

2.4. kód. Ütemezésért felelős osztály

2.6. Titkosítás

Az átvitel biztonságosságának a legfontosabb eleme a titkosítás, ugyanis nem szeretnénk, hogy az adott esetben érzékeny adataink nyilvánosságra kerülhessenek. Első lehetőségként a Datagram Transport Layer Security (DTLS) került a látóterembe. A DTLS egy olyan protokoll, amely az átvitel titkosítása mellett az egész kapcsolatot is titkosítja, így véd a lehallgatás, az üzenetek módosítása valamint a hamis üzenetek ellen. Mivel ez elég bonyolult és erőforrásigényes, így elsőként egy egyszerűbb megoldást kerestem. A Qt minitaprogramjai közt találtam is egyet, ami tartalmaz egy nagyon egyszerű implementációt stringek és byte arrayek titkosítására. Ezt a SimpleCrypt osztály valósítja meg, amely szimmetrikus titkosítást tesz lehetővé egy megadott kulccsal [18]. Ezt az osztályt felhasználva valósítottam meg a saját programomban a fájlok védelmét. További nagy előnyként

lehetőség van a SimpleCrypt osztály segítségével tömöríteni egy hosszú stringet, ami nem csak a méretét csökkenti, de a titkosítás bemeneteként sokkal inkább tűnik véletlenszerű szövegnek, így nehezebbé téve a feltörését. Ezt nem érdemes alkalmazni kis méretű stringeknél ugyanis a tömörítési eljárás után az eredmény még hosszabb is lehet mint az eredeti szöveg volt. Bináris tartalomra ez még fokozottabban igaz lesz.

Ahogy a 2.5-ös kódrészletben is látható, a titkosításhoz példányosítom a SimpleCrypt osztályt, ez lesz az m_SimpleCrypt, melynek kliens oldalon az encryptToByteArray függvényével titkosítok. Ehhez az m_pSendDataPointerbe már felolvasott adatokat használok fel. A titkosított változatot a pointer által mutatott memóriaterületre másolom vissza a memcpy segítségével. Ez a titkosított adat kerül továbbításra a következő csomagban a szerver felé. Szerver oldalon a decryptToByteArray metódussal tudom visszafejteni a csomagban kapott adatokat, melyeket az m_pReceiveDataPointerrel jelölt helyen tárolok el, a visszafejtés után ez kerül kiírásra a fájlba.

```
1 //Szerver
2 memcpy(m_pReceiveDataPointer, m_SimpleCrypt.decryptToByteArray(QByteArray(
3     m_pReceiveDataPointer, dataRead)).constData(), dataRead);
4
5 //Kliens
6 memcpy(m_pSendDataPointer, m_SimpleCrypt.encryptToByteArray(QByteArray(
7     m_pSendDataPointer, dataRead)).constData(), dataRead);
```

2.5. kód. Titkosítás szerver és kliens oldalon

2.7. CRC vizsgálat

A Cyclic Redundancy Check egy hiba kereső kód [19]. Leggyakrabban digitális hálózatokon, adathordozókon és adattároló eszközökön alkalmazzák a nyers adatokon történő véletlenszerű változások kiszűrésére. A Qt 5.9 verziójától kezdve a QByteArray osztályban található egy qChecksum nevű függvény, mely a CRC-16-CCITT algoritmust felhasználva készít egy ellenőrző összeget a megadott adathoz [20]. Ezt használtam fel a programomban. A küldést megelőzve elkészítem minden csomaghoz a CRC értéket, majd küldés után ismét kiszámolom azt. Amennyiben ez a két érték nem egyezik akkor a szerver nem küld visszajelzést a kliensnek a megérkezett csomagról. Ugyanis ez azt jelenti, hogy hiba került az adott csomagba így a szerver ezt eldobja, a kliens pedig újra fogja küldeni később a következő körben. Ezáltal kiszűrhetővé váltak a hibás csomagok, mellyel megbízhatóbbá vált az átvitel a küldött fájl sértetlenségét tekintve.

A kliensben a vizsgálat során először felolvasom a fájlból a megadott csomagmérettel a következő csomagban küldendő adatokat, ezeknek a `qChecksum` függvénnyel számolom ki a CRC értékét, melyet a csomaggal együtt elküldök a csomag fejlécében. A szerver oldalon kiolvasom a fejlécből ezt a CRC értéket, majd vizsgálom, hogy ez a kiolvasott érték megegyezik-e az érkezett adaton kiszámított CRC értékkel. Amennyiben ez a két érték nem egyezik meg, eldobom az érkezett csomagot. Továbbá növelem az `m_ui32CRCError` értékét, melyben a felfedezett hibákat tárolom. Ezt a folyamatot tartalmazza a 2.6-os kódrészlet.

```
1 //Szerver
2 m_ui16CRCValue = qFromBigEndian(*(quint16 *)m_pReceiveDataPointer);
3
4 if (m_ui16CRCValue != qChecksum(m_pReceiveDataPointerWithCRC, dataRead))
5 {
6     m_ui32CRCError++;
7 }
8
9
10 //Kliens
11 dataRead = m_file.read(m_pSendDataPointerWithCRC, m_ui16ChunkSize);
12
13 *(quint16 *)m_pSendDataPointer = qToBigEndian(qChecksum(m_pSendDataPointerWithCRC,
14                                                         dataRead));
```

2.6. kód. CRC vizsgálat szerver és kliens oldalon

2.8. Logolás

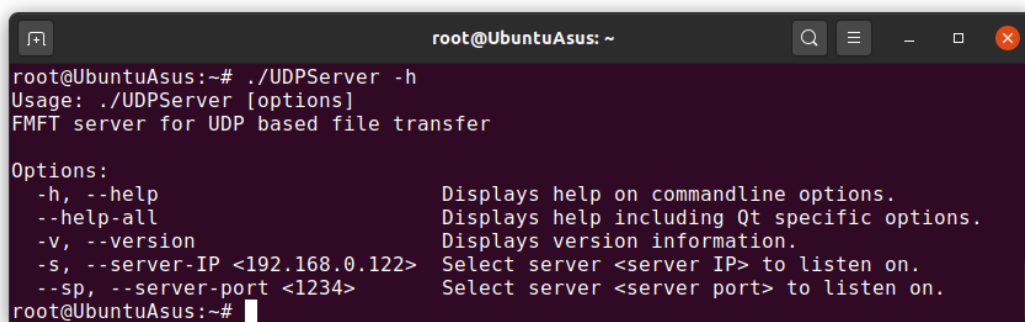
Egy jó szoftvernek elengedhetetlen követelménye a megfelelően részletes naplózás, melyre nagy szükség van a valós életbeli tesztek kiértékeléséhez, adott esetben a hibák vizsgálatahoz, valamint a továbbfejlesztési lehetőségek kereséséhez. A teszteléskor a több egymás utáni küldés során készült naplófájl jobb megkülönböztethetősége miatt a fájl nevében tárolom a küldés pontos időpontját, az átvitt adatblokk méretét bájtokban, az átvitel időtartamát, a maximális sebességhatárt, a küldött csomagok méretét valamint a csomagok újraküldési arányát százalékban mérve. Ám a küldés folyamata közben a napló fájl még csak egy ideiglenes nevet kap, mely tartalmazza a küldéshez használt adatcsatorna portját, ezt automatikusan átnevezem a küldés lezárulásával, ekkor kerülnek bele végül a mért adatok. Így a küldésről a legfontosabb információkat lehet látni a fájl megnyitása nélkül is.

A programban lehetőséget biztosítottam egy hosszabb részletes logoláshoz — bár ez

kis fájlok esetén akár nagyobb is lehet mint maga a küldött fájl —, illetve ezen felül lehetőség van a rövid naplózásra melyben csak a legszükségesebb információkat jegyzem le. Rövid naplózás esetén a fájlba belekerül a kliens IP címe, a kommunikációhoz használt port illetve a küldéshez használt adatcsatorna port száma, ezen felül a szerver IP címe valamint a kommunikációs portja. Továbbá tárolásra kerül a küldéshez használt csomagméret, az elküldött csomagok száma, a csomagok újraküldési aránya százalékban mérve, a kapcsolatfelvételhez szükséges időtartam valamint a küldés során elért átlagsebesség.

2.9. Paraméterezés

Ahogy már a programpár működését ismertettem, a szerver alkalmazás grafikus felület nélkül konzolról is használható. A 2.3-as ábrán látható, `-h` opcióval felhossa a lehetséges



```
root@UbuntuAsus: ~  
root@UbuntuAsus:~# ./UDPServer -h  
Usage: ./UDPServer [options]  
FMFT server for UDP based file transfer  
  
Options:  
-h, --help                Displays help on commandline options.  
--help-all              Displays help including Qt specific options.  
-v, --version            Displays version information.  
-s, --server-IP <192.168.0.122> Select server <server IP> to listen on.  
--sp, --server-port <1234> Select server <server port> to listen on.  
root@UbuntuAsus:~#
```

2.3. ábra. Szerver program parancssori argumentumai

argumentumokat. Mivel a küldéshez kapcsolódó opciókat a csatlakozó kientől fogja megkapni, a szerver indításakor mindössze az IP címet és portot van lehetőségünk megadni. Ezzel kiválaszthatjuk mely hálózati lábon várjuk a kapcsolatfelvételt illetve beállíthatjuk, hogy melyik port legyen az, amin kommunikálni fogunk. Az alapértelmezett lábat az ethernet és wifi típusú interfészek közül választom ki, ezáltal kiszűröm a virtuális lábakat, így biztosan valós láb lesz kiválasztva.

A szerver futtatásakor megadható argumentumokat a 2.7-es kódrészletben hozom létre. Itt létrehozok két `QCommandLineOption`t. Az első a `serverNameOption`, ez fog felelni a szerver IP címének magadásáért, a `serverPortOption` pedig a szerver portjáért. Miután ezeket létrehoztam, beállítom a program leírását és a help oldal kiírásának engedélyezését, a programverzió lekérésének lehetőségét, majd hozzáadom a két létrehozott opciót.

Végül pedig a `process` függvény segítségével feldolgozom a kapott argumentumokat a beállítások szerint. A kliens oldalon ugyanilyen módon teszem lehetővé az opciók megadását parancssoros futtatás esetén.

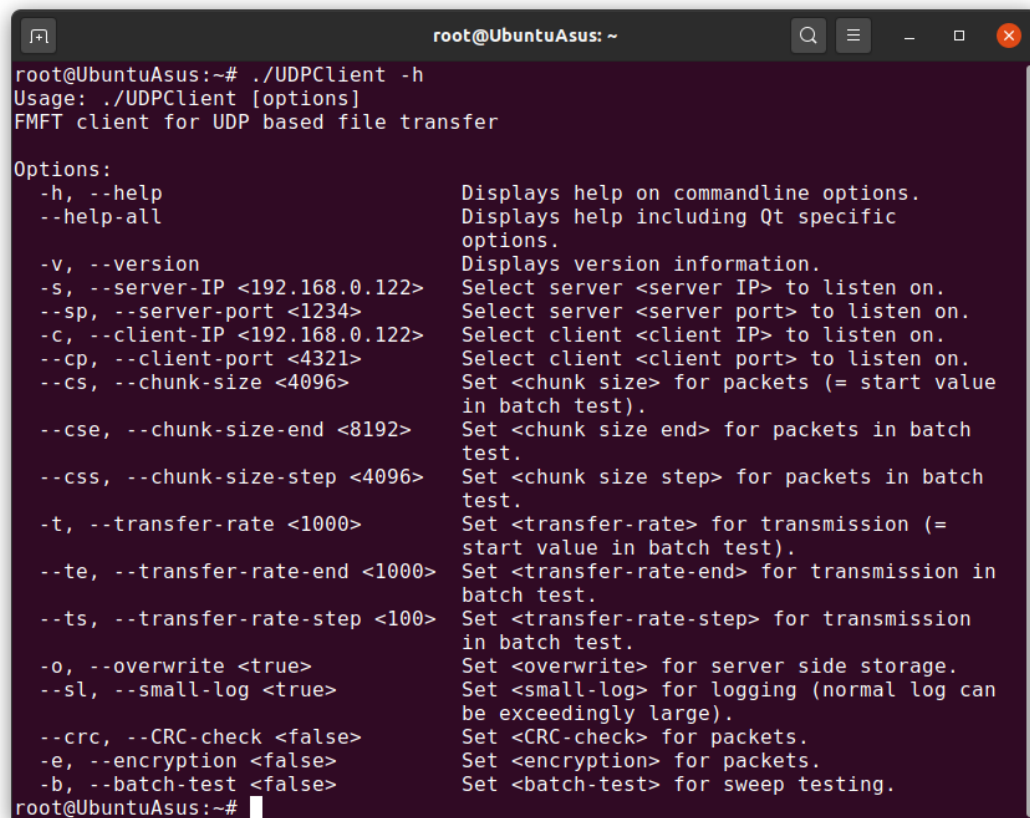
```

1 QCommandLineOption serverNameOption(QStringList() << "s" << "server-IP",
   QCoreApplication::translate("main", "Select server <server IP> to listen on."),
   QCoreApplication::translate("main", serverDefaultAddress.toString().toLatin1()),
   serverDefaultAddress.toString().toLatin1());
2
3 QCommandLineOption serverPortOption(QStringList() << "sp" << "server-port",
   QCoreApplication::translate("main", "Select server <server port> to listen on."),
   QCoreApplication::translate("main", "1234"), "1234");
4
5 parser.setApplicationDescription("FMFT server for UDP based file transfer");
6
7 parser.addHelpOption();
8 parser.addVersionOption();
9 parser.addOption(serverNameOption);
10 parser.addOption(serverPortOption);
11
12 parser.process(a.arguments());

```

2.7. kód. Parancssori argumentumok létrehozása

A kliens programnak ugyan van grafikus felülete, ám a könnyebb tesztelhetőség érdekében lehetőséget biztosítottam parancssoros futtatásra is, ez látható a 2.4-es ábrán. Ezt úgy oldottam meg, hogy ha futtatáskor kap valamilyen paramétert, abban az esetben felület nélkül indul el. Ahogyan már ismertettem korábban, a szerver a kienstől fogja kapni a küldés paramétereit, így ezeket innen van lehetőségünk beállítani. Az első paraméter az elküldendő fájl neve lesz teljes elérési úttal minden esetben. Megadhatjuk a szerver IP címét. Abban az esetben, ha ezt nem adjuk meg, az adott kliens gép saját IP címét kapja majd meg. Hasonlóan ahogy a szervernél is történik, a kliens IP címét megadva beállíthatjuk melyik hálózati lábon küldje ki az alkalmazás a csomagokat. A szerver portot megadva mondhatjuk meg, melyik porton érhetjük el a szerveret. A kliens portot beállítva pedig azt adjuk meg, melyik porton várjuk vissza a választ. Állíthatjuk a csomagméretet és az átviteli sebességet is. Ha a `batch test` be van kapcsolva, lehetőség van megadni kezdő csomagméretet, lépés méretet és a végső csomagméretet. Ez esetben a küldés végre lesz hajtva a kezdő mérettel és a megadott lépésenként minden mérettel a végső méretig. Ugyanez a lépésközönkénti futtatás megtehető az átviteli sebességhatárokkal is. Továbbá még állítható, hogy ha már létezik szerver oldalon az adott nevű fájl, akkor felülírja-e. Vezérelhető továbbá, hogy rövid vagy részletes naplózás legyen, végezzünk-e CRC tesztet illetve titkosítva legyenek-e a csomagok.



```
root@UbuntuAsus: ~  
root@UbuntuAsus:~# ./UDPClient -h  
Usage: ./UDPClient [options]  
FMFT client for UDP based file transfer  
  
Options:  
-h, --help                Displays help on commandline options.  
--help-all              Displays help including Qt specific  
                        options.  
-v, --version            Displays version information.  
-s, --server-IP <192.168.0.122> Select server <server IP> to listen on.  
--sp, --server-port <1234> Select server <server port> to listen on.  
-c, --client-IP <192.168.0.122> Select client <client IP> to listen on.  
--cp, --client-port <4321> Select client <client port> to listen on.  
--cs, --chunk-size <4096> Set <chunk size> for packets (= start value  
                        in batch test).  
--cse, --chunk-size-end <8192> Set <chunk size end> for packets in batch  
                        test.  
--css, --chunk-size-step <4096> Set <chunk size step> for packets in batch  
                        test.  
-t, --transfer-rate <1000> Set <transfer-rate> for transmission (= start value in batch test).  
--te, --transfer-rate-end <1000> Set <transfer-rate-end> for transmission in  
                        batch test.  
--ts, --transfer-rate-step <100> Set <transfer-rate-step> for transmission  
                        in batch test.  
-o, --overwrite <true> Set <overwrite> for server side storage.  
--sl, --small-log <true> Set <small-log> for logging (normal log can  
                        be exceedingly large).  
--crc, --CRC-check <false> Set <CRC-check> for packets.  
-e, --encryption <false> Set <encryption> for packets.  
-b, --batch-test <false> Set <batch-test> for sweep testing.  
root@UbuntuAsus:~#
```

2.4. ábra. Kliens program parancsori argumentumai

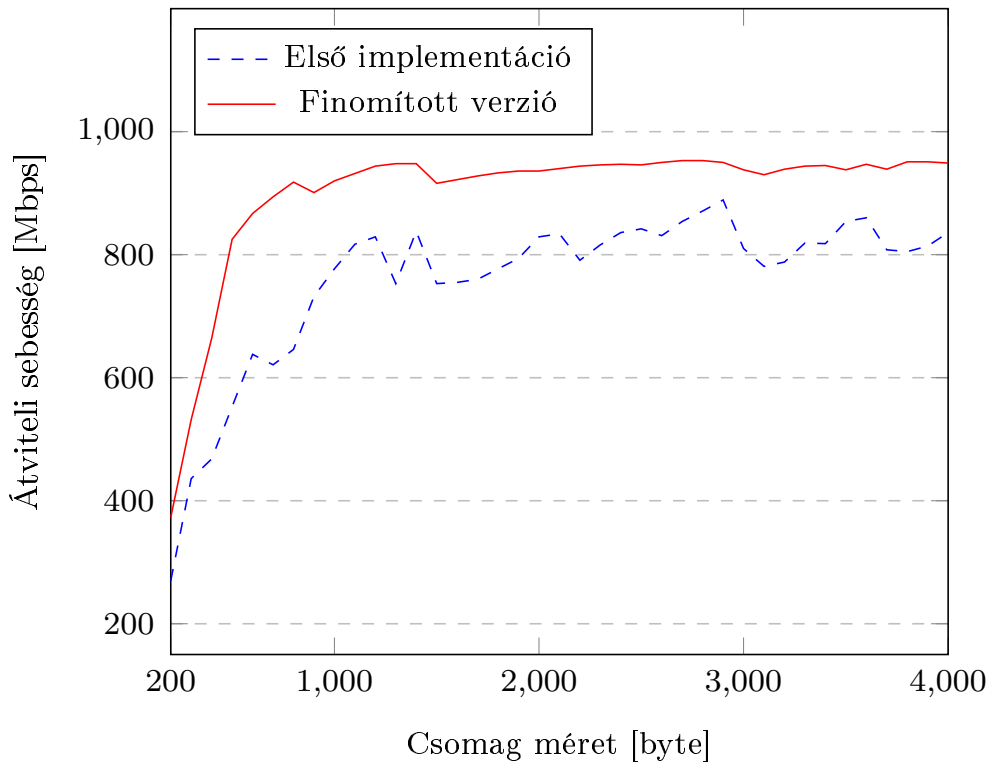
3. fejezet

Átviteli tesztek

Ebben a fejezetben az általam végrehajtott tesztekéről és azok eredményeiről fogok beszámolni. A teszteket az én két saját számítógépem között hajtottam végre, miközben csak a gerinchálózat felé volt aktív hálózati kapcsolat a routeren. Így minimális az egyéb hálózati forgalom által okozott sebességcsökkenés. A teszteket ugyanazzal a 100 MB-os tesztfájllal hajtottam végre, az átviteleket különböző csomagméretekkel valósítottam meg 200 byte-tól egészen 4000 byte-ig, valamint a maximális átviteli sebességként 1000 Mbps volt rögzítve minden esetben a könnyebb összehasonlíthatóság érdekében.

3.1. Csomagméret tesztek

Ahogy az látható a 3.1-es diagramon, kék szaggatott vonallal jelöltem az elsőként elkészült átvitelre képes programpár által végrehajtott teszt sorozatot. Látható, hogy itt az elért sebesség elég ingadozó, legelőször az 1100 byte-os csomagméreteknél éri el a 800 Mbps körüli sebességet. A legjobb értéket 2900 byte-nál mértem, ami 889 Mbps volt. A piros folytonos vonallal jelöltem az utolsó szoftververzióval végrehajtott tesztsorozat eredményeit. A tesztet, hasonlóan az előzőhöz, 200 byte és 4000 byte közötti csomagméretekkel hajtottam végre ugyanazon a 100 MB-os teszt fájlon. Ebben az esetben látható, hogy az átvitel már kisebb csomag méretnél is eléri a 800 Mbps-ot, ebben a tesztben egész pontosan 500 byte-os méretnél lépi át először, majd 800 byte-nál a 900 Mbps-os sebességet is átlépi. Ezen felül stabilizálódik a rendszer, nincsenek nagy sebességbeli ingadozások, így megbízhatóan stabil sebességgel történtek az átvitelek.

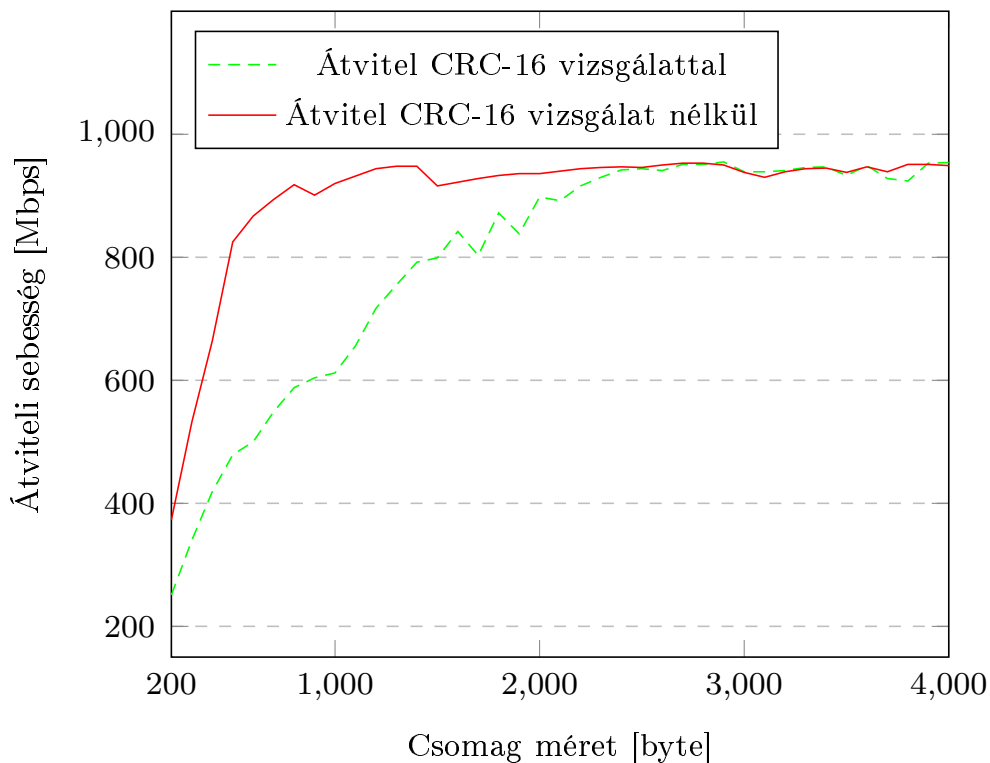


3.1. ábra. Az első és az utolsó szoftver verzió közti stabilitási különbség

3.2. Csomagméret tesztek CRC vizsgálattal

A 3.2-es ábrán az utolsó szoftver verzióval készített tesztet használtam fel összehasonlítási alapnak, amit már az előző ábrán is lehetett látni piros folytonos vonallal. A másik teszt sorozatot CRC-16 vizsgálatot végrehajtva készítettem el a program párral, ezt zöld szaggatott vonallal jelöltem. Ellentétben a CRC vizsgálat nélküli tesztel, itt csak nagyobb, 1600 byte-os csomag méretnél lépjük át a 800 Mbps-os határt, pontosan 842 Mbps sebességgel, valamint 2200 byte fölött elérjük a 900 Mbps határt. A két görbét tekintve a CRC-16 vizsgálat esetén a sebesség szinte végig a nyers csomagok átvitelének sebessége alatt van 2500 byte fölött viszont tartja a sebességet, hasonlóan a vizsgálat nélküli átvitelhez. A lassú emelkedés magyarázható a CRC vizsgálat műveletigényével, ami kis csomagméreteknél a küldési sebességhez képest sok időt vesz igénybe. A felső sebességkorlát miatt, amikor már az adott időintervallumban kevesebb csomag lesz elküldve, a vizsgálat elvégzéséhez több idő áll rendelkezésre, így magyarázható a stabilizálódás. Más tesztfájl méreteknél is hasonló viselkedést mutat a program párosom. A különbség

3% körüli 2400 byte feletti csomag méreteknél. Ezzel szemben kis méreteknél akár 40%-os különbséget is mutathat az eltérés, az ábrán a legnagyobb különbség 500 byte-os csomagméretnél fordul elő, itt 40%-os a differencia.

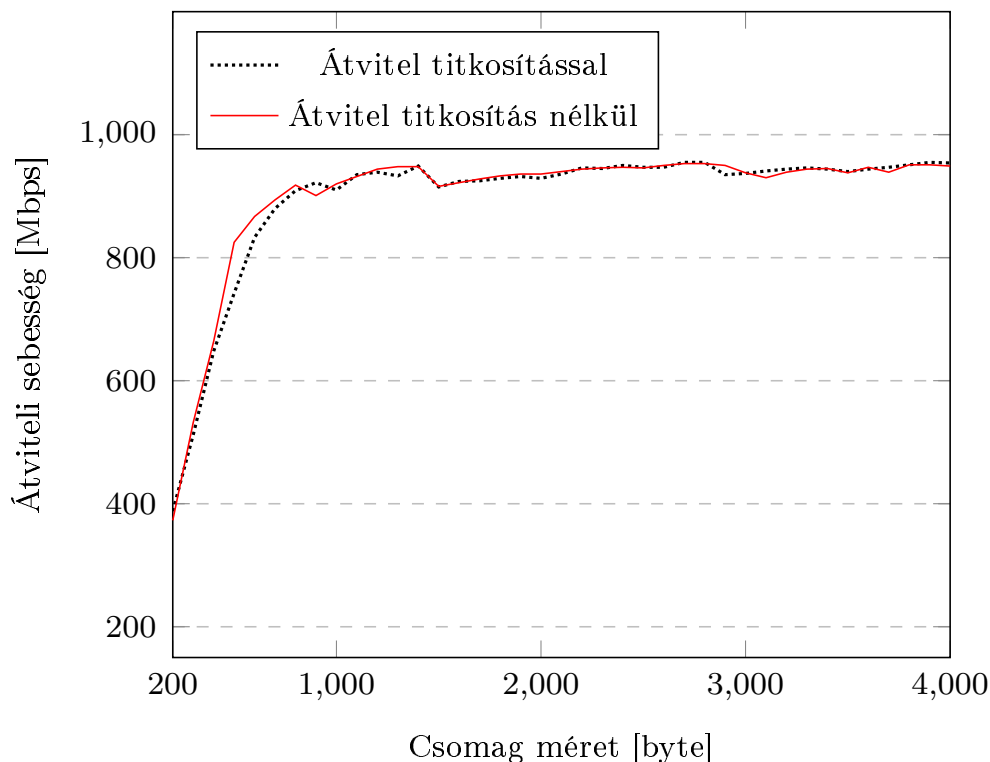


3.2. ábra. Sebességbeli különbség CRC vizsgálattal és nélküle

3.3. Csomagméret tesztek titkosítással

A fájl titkosításával végzett teszthez is az előzőekhez hasonlóan a piros folytonos vonallal jelölt utolsó szoftver verzióval készített teszt sorozatot használtam fel. A 3.3-as ábrán fekete pontozott vonallal ábrázoltam a titkosított átvitelt. Ahogy 2.6-os pontban írtam, egy egyszerű és kis számítási igényű titkosítást választottam, hogy a lehető legkevésbé rontsa az átviteli sebességet. Ahogy az ábrán is látható minimális a különbség a két átvitel között, szorosan követik egymást. Ebben az esetben a legnagyobb eltérés 500 byte-os csomagméretnél lett, itt majdnem 10%-os a sebességbeli különbség, a tesztek nagy részében 3% alatt volt. Néhány helyen látható, hogy át is lépi a titkosítással végzett teszt sebessége a titkosítás nélkülit, ez persze nem azt jelenti, hogy titkosítva a fájlokat, még

gyorsabb átvitelt is kaphatunk eredményül, csupán a teszt körüli körülmények változása okozta ezt.

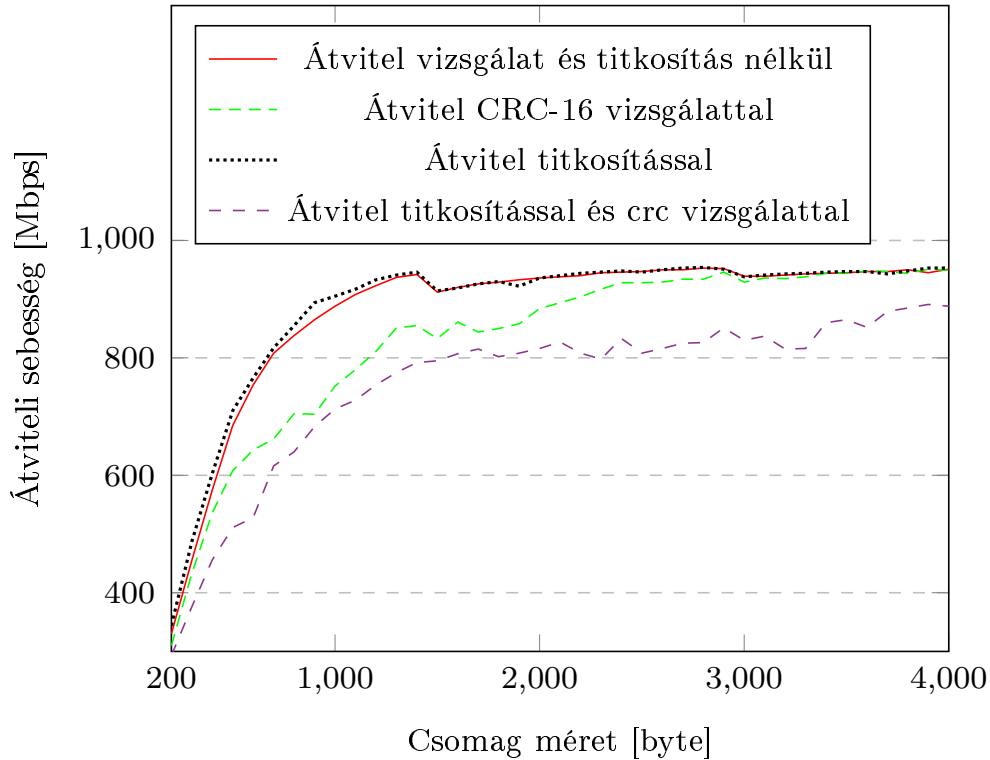


3.3. ábra. Sebességbeli különbség titkosítással és anélkül

3.4. Átlagsebesség a különböző technikák esetén

Az utolsó ábrám elkészítéséhez több egymás utáni tízelemes átvitelsorozatot valósítottam meg ugyanazokkal a paraméterekkel, hogy egy átlagos értéket számolhassak, ami jobban tükrözi a különböző átviteli opciókkal elérhető sebességet. Ennek eredményét a 3.4-es ábrán szemléltetem. Elsőként a piros folytonos vonallal jelölt tesztet hajtottam végre, melyben se CRC vizsgálatot, se titkosítást nem végeztem az átvitt fájlra. Ahogy a korábbi ábrák is mutatták, itt is meggyőződhetünk róla, hogy a kisebb csomagméretek esetén kisebb átviteli sebesség érhető el. Először 1100 byte-nál érem el a 900 Mbps sebességet, így megállapítható, hogy valós életbeli átvitel esetén érdemes ennél nagyobb csomagméretet választani a legjobb sebesség elérése érdekében. Egy kisebb sebességbeli esés tapasztalható 1400 byte és 1500 byte-os méretek között, illetve 2900 byte és 3000 byte között is, valószínűleg ezek a belső folyamatok felé kevésbé optimális méretek lesznek. Feltehetőleg

nagyobb csomagméretek esetén is hasonló közökkel további esések tapasztalhatók majd, bár a méretek növekedésével egyre kisebbek lesznek az esések. Azonban látható, hogy tartjuk végig a 900 Mbps feletti sebességet a csökkenések ellenére is.



3.4. ábra. Átlagsebességek a különböző technikáknál adott csomagméretek esetén

A CRC vizsgálattal végrehajtott átvitelt jelöltem zöld szaggatott vonallal. Itt továbbra is tapasztalható az ingadozó átviteli sebesség, de 2000 byte-os méret felett már stabilizálódik, így ajánlott ettől magasabb csomagmérettel dolgozni. A 900 Mbps feletti sebességet 2200 byte-os csomagméretnél érem el. Az előző ábrához hasonlóan a – fekete pontozott vonallal jelölt – titkosítással végzett átvitelsorozat szorosan követi az első piros vonalat, kisebb méreteknél még fölé is emelkedik minimálisan. Ez azt jelzi, hogy a SimpleCrypt a nevéhez híven nagyon egyszerűen számolható, nem lesz tőle CPU limitált az átviteli eljárásom. A tesztek során tapasztaltam, hogy kis csomagméreteknél az elért sebesség nagyon változó lehet, például 500 byte-os csomagméretnél mértem 825 Mbps és 475 Mbps átviteli sebességeket is. Így határozottan kimondhatom, hogy nem érdemes elhagyni a titkosítást az átvitelkor, mert nem befolyásolja olyan mértékben a teljesítményt, hogy megérje a biztonságot csökkenteni. Végül ezúttal végeztem olyan tesztet is melyben

a CRC vizsgálatot és a titkosítást is végrehajtom az átvitt fájlra. Ezt lila szaggatott vonallal jelöltem. Itt sajnos a teljesítmény jelentősen csökken, a sok plusz számítás miatt nem éri el a többi opció által stabilan tartott 900 Mbps fölötti sebességet. Azonban 1600 byte-os méretnél átlépi a 800 Mbps-os sebességet, és az alá nem esik vissza, így megállapítható, hogy ha a biztonságot előre tartjuk a sebességnél, akkor is egy stabil és gyors átvitelt kaphatunk. Külön ezt a kombinált beállítást vizsgálva csak a 4000 byte feletti csomagméreteknél értem el a stabil 900 Mbps fölötti átvitelt.

Eredmények

Kutatómunkám eredményeképp rendelkezek egy hatékony és könnyen kezelhető fájlátviteli rendszerrel, ami elérhető több operációs rendszerre is. A kliens program képes alkönyvtárakat kezelni és egy vagy több kiválasztott fájlt átküldeni a kijelölt szervernek. Készítettem hozzá grafikus felhasználói felületet, mellyel könnyedén kiválaszthatjuk a továbbítandó állományokat, de használható konzolról is, mely segíti az egyszerű tesztelhetőséget. A szerver kizárólag konzolos felületen futtatható, képes fájlokat fogadni a hozzá csatlakozó kliensektől, akár párhuzamosan több kienstől is. Az átvitelt a szoftverben több opcióval is finomhangolhatjuk. Választható az átvitt fájl titkosítása amennyiben biztosak akarunk lenni benne, hogy nem fér hozzá illetéktelen az adatainkhoz. Garantálható az adatok sértetlensége a CRC vizsgálat használatával. Illetve elhagyhatjuk ezeket a beállításokat, ha a lehető leggyorsabb és leghatékonyabb átvitel a cél. Továbbá sikerült elérnem a célként kitűzött stabil 800 Mbps feletti sebességet.

Továbbfejlesztési lehetőségek

Az FMFT szoftverpárban sok távlati lehetőség rejlik. A jövőben bővíteni fogom további titkosítási lehetőségekkel is az alkalmazásaimat, például a DTLS-el, melynek segítségével az egész kommunikációs csatornát titkosíthatjuk a biztonság növelése érdekében. A megbízhatóságot is tovább javítom majd, a jelenlegi csomagonkénti CRC-16 vizsgálat mellé az egész fájlra is végezni fogok egy CRC-32 vagy CRC-64 vizsgálatot a fájl méretétől függően. Az átvitelt kétirányúvá teszem, hogy a kliens ne csak küldeni tudjon a szervernek, hanem a szervertől kérni, vagyis fogadni is tudjon fájlokat. Ehhez igazodva készítettem el már az új grafikus felületet is a kliens számára, melyet még bővíteni lehet további használatot egyszerűsítő funkciókkal, valamint a fájlátviteli beállítások megadásának lehetőségével. Ezen felül szüneteltethetővé teszem az átvitelt, melyet a programból való kilépés esetén is bármikor újraindíthatok majd tervezetten, és onnan folytatódik tovább az átvitel, ahol az előzőleg abbamaradt. Ennek megoldásával a megszakadt küldéseket folytathatóvá tudom tenni. Tervezem a további párhuzamosítást, hogy még gyorsabbá válhasson a küldési és fogadási folyamat, valamint vannak még nem implementált optimalizálási lehetőségek, melyek javítani tudják a sebességet. A csomagfejlécek tartalmának újragondolásával a küldött byteok számát is csökkenteni lehet, úgy hogy a származtatott adatok elhagyásával az ebből fakadó redundáns adatforgalom tovább minimalizálható.

Irodalomjegyzék

- [1] UDP-based Data Transfer Protocol (UDT) (September 9, 2020), URL <https://udt.sourceforge.io/index.html>.
- [2] ANGLANO, C., CANONICO, M., A Comparative Evaluation of High-Performance File Transfer Systems for Data-intensive Grid Applications, *13th IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises* (June 2004):pp. 283–288.
- [3] ANGLANO, C., CANONICO, M., Performance analysis of high-performance file transfer systems for Grid applications, *Special Issue: First International Workshop on Emerging Technologies for Next-generation GRID (ETNGRID 2004)*, vol. Vol18(Issue8) (July 2006):pp. 807–816, doi:10.1002/cpe.976, URL <https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.976>.
- [4] MADHURI, D., REDDY, P. C., Performance comparison of TCP, UDP and SCTP in a wired network, in *2016 International Conference on Communication and Electronics Systems (ICCES)*, Coimbatore, India (October 2016), ISBN 978-1-5090-1066-0, pp. 1–6, doi:10.1109/CESYS.2016.7889934.
- [5] SAWASHIMA, H., HORI, Y., SUNAHARA, H., Characteristics of UDP Packet Loss: Effect of TCP Traffic, in *Proceeding of the 7th Annual Conference of the Internet Society*, Kuala Lumpur, Malaysia (June 1997), URL https://web.archive.org/web/20160103125117/https://www.isoc.org/inet97/proceedings/F3/F3_1.HTM.
- [6] HORI, Y., SAWASHIMA, H., SUNAHARA, H., OIE, Y., Performance Evaluation of UDP Traffic Affected by TCP Flows, in *IEICE TRANSACTIONS on Communications*, vol. E81-B (August 1998), ISSN 0916-8516, pp. 1616–1623.

- [7] HE, E., LEIGH, J., YU, O., DEFANTI, T. A., Reliable Blast UDP : Predictable High Performance Bulk Data Transfer, in *Proceedings of IEEE Cluster Computing, Chicago, Illinois* (September 2002), URL <https://www.evl.uic.edu/cavern/papers/cluster2002.pdf>.
- [8] LAZAR, G., PENE, R., *Mastering Qt 5: Create stunning cross-platform applications*, Packt Publishing Ltd., Birmingham, England (December 15, 2016).
- [9] ROURKE, M., *Learn WebAssembly: Build web applications with native performance using Wasm and C/C++*, Packt Publishing Ltd., Birmingham, England, 1st edn. (September 24, 2018).
- [10] Qt for WebAssembly, URL https://wiki.qt.io/Qt_for_WebAssembly, last edited on 4 December 2020.
- [11] ENG, L. Z., *Qt5 C++ GUI Programming Cookbook: Practical recipes for building cross-platform GUI applications, widgets, and animations with Qt 5*, Packt Publishing Ltd., Birmingham, England, 2nd edn. (March 27, 2019).
- [12] Qt QML (April 7, 2021), URL <https://doc.qt.io/qt-6/qtqml-index.html>.
- [13] Boost C++ Libraries (April 7, 2021), URL <https://www.boost.org/doc/libs/>.
- [14] PIETERSE, V., KOURIE, D. G., CLEOPHAS, L. G., WATSON, B. W., Performance of C++ bit-vector implementations, in *Proceedings of the 2010 Annual Conference of the South African Institute of Computer Scientists and Information Technologists, SAICSIT 2010, Bela Bela, South Africa, October 11-13, 2010*, ACM International Conference Proceeding Series, ACM (2010), pp. 242–250, doi: 10.1145/1899503.1899530, URL <https://doi.org/10.1145/1899503.1899530>.
- [15] GETTMAN, K., STEWART, R., XIE, Q., *Stream Control Transmission Protocol (SCTP): A Reference Guide*, Addison-Wesley Professional, 1st edition edn. (October 2001), ISBN 978-0201721867.
- [16] LEUNG, V. C., RIBEIRO, E. P., WAGNER, A., IYENGAR, J., *Multihomed Communication with SCTP (Stream Control Transmission Protocol)*, CRC Press Taylor & Francis Group, 1st edn. (2012).

- [17] MITTAL, R., AGARWAL, R., RATNASAMY, S., SHENKER, S., Universal Packet Scheduling, in *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation*, NSDI'16, USENIX Association, USA (2016), ISBN 9781931971294, p. 501–521.
- [18] Simple encryption with SimpleCrypt, URL https://wiki.qt.io/Simple_encryption_with_SimpleCrypt, last edited on 16 June 2015.
- [19] ChecksumIso3309 (September 9, 2020), URL <https://doc.qt.io/qt-5/qt.html#ChecksumType-enum>.
- [20] qChecksum (September 9, 2020), URL <https://doc.qt.io/qt-5/qbytearray.html#qChecksum>.

Kódjegyzék

2.1. Kliens oldali átviteli függvény	11
2.2. Szerver oldali kontroll csatornán érkező csomagok kezelése	12
2.3. Szerver oldali adatcsatornán érkező csomagok kezelése	13
2.4. Ütemezésért felelős osztály	15
2.5. Titkosítás szerver és kliens oldalon	17
2.6. CRC vizsgálat szerver és kliens oldalon	18
2.7. Parancssori argumentumok létrehozása	20

Ábrák jegyzéke

2.1. FMFT kliens teszt felülete	9
2.2. FMFT kliens új felülete	10
2.3. Szerver program parancssori argumentumai	19
2.4. Kliens program parancssori argumentumai	21
3.1. Az első és az utolsó szoftver verzió közti stabilitási különbség	24
3.2. Sebességbeli különbség CRC vizsgálattal és nélküle	25
3.3. Sebességbeli különbség titkosítással és anélkül	26
3.4. Átlagsebességek a különböző technikáknál adott csomagméretek esetén . .	27

Saját munka leírása

Az implementáció elkészítése előtt alaposan felmértem a rendelkezésre álló technikákat és lehetőségeket. Teszteltem több már létező fájlátviteli rendszert és összegyűjtöttem ezekből az ígéreteseket. Megvalósítottam a fájlátvitelt az SCTP protokollt használó minta projektben, majd vizsgáltam a vele elérhető teljesítményt. A programpár elkészítése előtt aktívan részt vettem a tervezési munkálatokban. A fejlesztés során az implementáció elkészítése közben minden lépésben közreműködtem, egyes modulokat a programterv alapján teljesen önállóan implementáltam. Az első szoftver verzió elkészülte után minden egyes verzióval részletes teszteket végeztem a továbbfejlesztési lehetőségeket és hibákat keresve.

Köszönetnyilvánítás

A kutatást az „Integrált kutatói utánpótlás-képzési program az informatika és számítástudomány diszciplináris területein” (EFOP-3.6.3-VEKOP-16-2017-00002) című projekt támogatta. A projekt az Európai Unió támogatásával, az Európai Szociális Alap társfinanszírozásával valósult meg. A dolgot továbbá a Debreceni Egyetem FIKP-20428-3/2018/FEKUTSTRAT projektje és a QoS-HPC-IoT Laboratóriuma is támogatta.

