VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

FACULTY OF INFORMATION TECHNOLOGY

TFTP Server a Klient

TFTP Server and Client

PROJEKT DO PŘEDMĚTU ISA

PROJECT FOR COURSE ISA

AUTOR PRÁCE JMÉNO PŘÍJMENÍ

AUTHOR **Tomáš Vlach (xvlach24)**

BRNO 2023

Obsah

1. Úvod

1. Cíle projektu

Cílem projektu je vytvořit dva spustitelné soubory *“tftp-server”* a *“tftp-client”* které budou dokázat komunikovat pomocí protokolu TFTP popsaném níže a umožňovat tak uživately volně stahovat a nahrávat soubory na vzdálený TFTP server.

Projekt také musí implementovat možnost rozšíření a to Options extension, Blocksize option a Timeout Interval and Transfer Size Options uvedené níže.

2. Jak funguje TFTP Protocol

Protocol TFTP (Trivial File Transfer Protocol) je jednoduchou verzí normálně používané protokolu SFTP, který používá UDP komunikaci pro přenos kontrolních hlaviček i dat. Je popsán v RFC 1350 [1]. TFTP také může využívat rozšíření základního popisu jako například options extension v RFC 2347 [2] a nebo blocksize option [3].

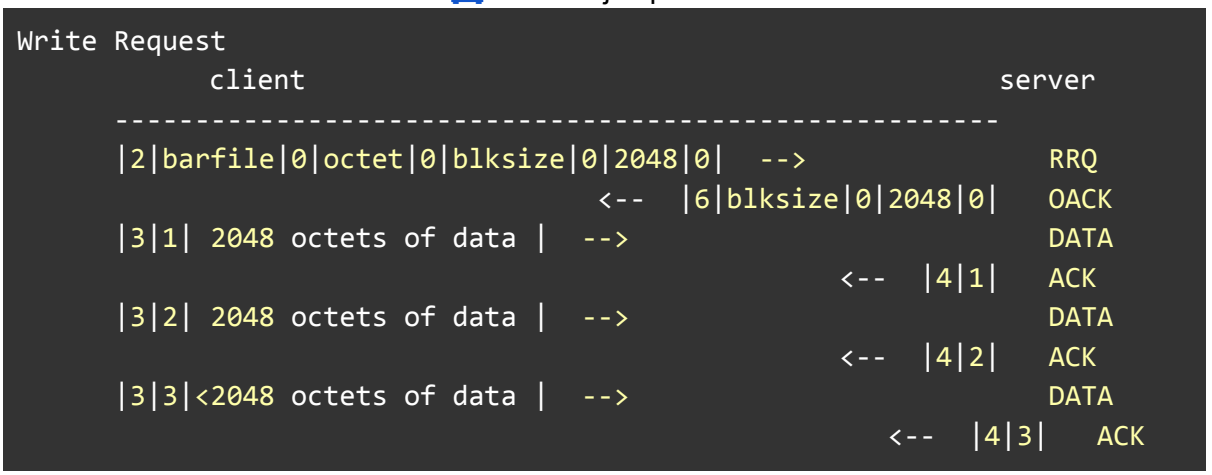
Každý přenos dat mezi serverem a klientem ať už v jakémkoli směru musí být iniciován klientem, který zasílá požadavek na server ve formátu popsaném v protokolu. Tento požadavek může být ať už stáhnutí nějakého souboru a nebo nahrání souboru na vzdálený server. Server při přijetí požadavku se zkontroluje zda požadavek může vyřídit a pokud ano tak zahájí přenos dat ať už zaslání potvrzení jakýchkoliv options z rozšíření v OACK paketu a nebo pokud žádné volby neobsahuje tak pošle DATA packet s prvními daty a nebo ACK packet aby naznačil klientovi že očekává aby mu poslal data.

Data jsou odesílána zamykacím způsobem, což znamená že pouze jeden paket dat se posílá naráz a pak se následně čeká až přijde potvrzení že druhá strana data obdržela tím že pošle ACK paket. Toto umožňuje jistotu že druhá strana obdržela daná data a to že při přenosu je po rukou pouze jeden paket dat a nemohou se tedy promíchat ani zabírat příliš mnoho místa na paměti.

Úryvek z RFC 2347 [2] vizualizující přenos RRQ s volbou blocksize:

```
Read Request
            client                                              server
      ------------------------------------------------------
      |1|foofile|0|octet|0|blksize|0|1432|0|  -->                 RRQ
                                    <--  |6|blksize|0|1432|0|   OACK
      |4|0|  -->                                                 ACK
                              <--  |3|1| 1432 octets of data |   DATA
      |4|1|  -->                                                 ACK
                              <--  |3|2| 1432 octets of data |   DATA
      |4|2|  -->                                                 ACK
                              <--  |3|3|<1432 octets of data |   DATA
      |4|3|  -->                                                 ACK
```

Úrivek z RFC 2347 [2] vizualizující přenosWRQ s volbou blocksize:



2. Návrh řešení

1. Zvolený jazyk

Pro řešení tohoto projektu jsem zvolil programovací jazyk C z důvodů mých zkušeností s ním a minimálním zkušenostem s C++.

2. Zpracování vztupů

Projekt nevyžaduje komplikované vstupní argumenty s několika přepínači a maximálně jeden stdin vstup, pro server složku kterou bude považovat jako výchozí a pro klienta soubor který bude chtít nahrávat.

Zbylé argumenty jak bylo již zmíněno jsou přepínače. Pro klienta je to -p port, -h ip/hostovací jméno serveru na který se bude chtít připojovat, -t cílovou cestu souboru pro uložení, což platí jak pro stahování tak posílání a -f který určuje jaký soubor by se měl na vzdáleném serveru stáhnout. Pro server existuje pouze jeden přepínač a to -p port na které se má server pokusit otevřít socket.

Pro zpracování tohoto množství vstupů bude stačit nějaká základní funkce, kterou buď bude implementována a nebo bude použita již nějaká existující.

3. Design logiky

Oba programy budou začínat parsováním a kontrolou vstupů, následovat bude příprava informací pro síťové spojení a připojení. Od připojení se budou programy velmi lišit kde klient vytvoří požadavek a pošle ho na server a server bude v nekonečné smyčce čekat na příchod požadavku.

Když server obdrží požadavek vytvoří separátní process pro zpracování požadavku a bude znovu čekat na nový požadavek.

Separátní process přečte požadavek, zkontrolujte zda s ním není žádný problém a pokud vše projde zašle zpět OACK paket pro zahájení přenosu. Algoritmus přenos se bude lišit mezi zasíláním dat ze serveru a zasíláním dat z klienta pouze na jeho počátku a proto se bude zásadní udělat algoritmus zaměnitelný aby se mohl použít pro obě strany.

4. Generace paketů

Pro generaci paketů budou implementovány separátní funkce a to pro každý typ. Mezi tyto funkce budou patřit funkce pro vytvoření paketu daného typu, přečtení paketu daného typu, vypsání paketu daného typu na výstup a pro některé pakety i funkce pro zaslání daného paketu.

Pro jednodušší práci na úrovni UDP paketu se budou konvertovat při použití funkce pro tvorbu paketu proměnné na char array který se dá rovnou poslat skrze UDP socket.

3. Implementace řešení

1. GitHub repozitář projektu

Odkaz na Github implementace je zde.

2. Zvolený jazyk

Při práci na projektu jsem narazil na několik zásadních nevýhod mnou zvoleného jazyka, kde několikrát se stalo že kvůli jeho limitacím a nebo neshodě očekávaných výsledků s realitou se musela logika a struktura projektu několikrát změnit a to způsobilo dosti závažné zpoždění a nevzhlednou strukturu programu.

3. Zpracování vstupů

Pro zpracování vstupních argumentů byla použita funkce ‘getopt’ z důvodů jednoduchého zpracování a velké možnosti úprav práce se vstupy. Kontrola správných kombinací vstupních argumentů se provádí přes sérii kontrol pomocí ‘if’.

4. Logika programu

Implementaci základní logiky se podařilo až na dvě výjimky implementovat podle návrhu. Kontrola argumentů i vytvoření údajů pro síťovou komunikaci je řešeno jednoduchými operacemi, ale připojení a zasílání dat se setkali s problém při implementaci.

Pro správný chod TFTP protokolu server musí změnit port přes který při řešení požadavku bude používat aby neblokoval výchozí port, což se setkalo s problémem obrácení komunikace mezi serverem a klientem. Klient potřeboval obdržet nové pakety od nového socketu serveru, ale by stále připojený k výchozímu. Řešením se nakonec stalo invertování připojení, kdy před odesláním odpovědi na požadavek server vytvoří nový socket a uživatel zrušil svůj starý a vytvoří nový a začne na něm poslouchat a čekat na zprávu od nového socketu serveru. Prakticky se z klienta stane server který čeká na to až se k němu někdo připojí, což provede server s novým socket neblokující půdovní.

Druhý zásadní problém problém při implementaci nastal při tvorbě funkcí *“send_file”* a *“recieve_file”* u kterých při testování bylo zjištěno že dokázají odesílat pakety v pořádku, ale pokud se pokusí pomocí funkce *“recvfrom”* získat paket jim zaslaným tak funkce *“recvfrom”* uzavře pro to použitý socket bez varování ani pořádného vysvětlení. Toto chování se opakovaně stávalo pouze pokud byla funkce *“recvfrom”* použita zanořená ve funkci a proto místo dvou funkcí pro jedna pro zasílání dat a druhá pro přijímání dat bylo zvoleno implementovat tyto funkce přímo do hlavní funkce *“main”*, kde se *“recvfrom”* choval podle očekávání. Bohužel to vede k nepřehlednosti kód a komplikace úprav algoritmů přenosu.

Funkce pro přenos zůstali v kódu jako komentáře pro případnou úpravu pokud se nalezne původ chyby vyvolané na zachytávání příchozích zpráv.

```
int recieve_file(int _sock, struct sockaddr_in* _servaddr, struct
sockaddr_in* _cliaddr, int _len, char* _filePath, char* _mode, int
_blockSize)

int send_file(int _listenfd, struct sockaddr_in* _servaddr, struct
sockaddr_in* _cliaddr, int _len, char* _filePath, char* _mode,int
_blockSize)
```

5. Generace paketů

Pro každý typ packetu TFTP byl vytvořen vlastní soubor *“TYP_pack.c”* v adresáři *“include/packets”*, kde TYP je typ daného packetu, například *“ack_pack.c”*. Každý tento soubor obsahuje několik funkcí, které všechny pracují nad danou paketou.

Pro tvoření pakety z proměných existují *“TYP_packet_create”* funkce vytvářející pole charakterů ve tvaru požadovaného paketu.

```
char* OACK_packet_create(int* _returnSize, int _blockSize, int _timeout, int
_tsize)
```

Pro překlad těchto polí při přijetí paketu každý soubor paketu také obsahuje *“TYP_packet_read”* funkce, které dokázají pole rozložit na potřebné proměnné.

```
int OACK_packet_read(char* _packet, int _packetLenght, int* _blockSize, int*
_timeout, int* _tsize)
```

Každý paket má také svoji funkci pro vypsání na stderr (výstup pro tento projekt) ve funkci *“TYP_packet_write”*.

```
void OACK_packet_write(char* _ip, int _sourcePort, int _blockSize, int _timeout,
int _tsize)
```

Některé pakety také mají funkce pro jejich automatické zaslání a to ve tvaru *“TYP_packet_send”*, které dokázali vytvořit a zaslat daný paket na určenou destinaci.

```
void OACK_packet_send(int _listenfd, struct sockaddr_in* _servaddr, struct
sockaddr_in* _cliaddr, int _cliaddrSize, int _blockSize, int _timeout, int
_tsize)
```

4. Testování

1. Testovací prostředí

Jako hlavní testovací prostředí bylo zvoleno WSL (Windows Subsystem for Linux) z důvodů nejlepšího přístupu z dostupných linuxových prostředí.

Další testy byli také prováděny na linuxovém serveru merlin.fiv.vutbr.cz pro kontrolu kompatibility mezi výcero systémy.

Pro kontrolu správných přenosů bylo také využito nástroje Wireshark a přenášená data byli hlavně texty.

2. Testy

Test byli řazeny podle typu přenosu a velikosti přenášeného souboru. Jako první byl malý soubor o 400 bytech (<512 bytes) přes oba přenosy.

Příkaz:

```
./bin/tftp-client -p 69 -h 127.0.0.1  -t ./bin/client/smalltest -f smalltest
```

Výstup na straně klienta:

```
OACK 127.0.0.1:1025: blksize=8192 timeout=1 tsize=400
DATA 127.0.0.1:1025:2048 1
```

Výstup na straně serveru:

```
RRQ 127.0.0.1:2048 "./server/smalltest" netascii {$OPTS}
ACK 127.0.0.1:2048 0
ACK 127.0.0.1:2048 1
```

Výstup ve Wireshark:

```
1    0.000000000    127.0.0.1    127.0.0.1    UDP    95    2048 → 69 Len=53
2    0.001943229    127.0.0.1    127.0.0.1    UDP    76    1025 → 2048 Len=34
3    0.002263792    127.0.0.1    127.0.0.1    UDP    47    2048 → 1025 Len=5
4    0.003760556    127.0.0.1    127.0.0.1    UDP    446    1025 → 2048 Len=404
5    0.004347595    127.0.0.1    127.0.0.1    UDP    47    2048 → 1025 Len=5
```

Příkaz:

```
./bin/tftp-client -p 69 -h 127.0.0.1  -t smalltest2 ./bin/client/smalltest
```

Výstup na straně klienta:

```
OACK 127.0.0.1:1025: blksize=512 timeout=1 tsize=404
ACK 127.0.0.1:1025 1
```

Výstup na straně serveru:

```
WRQ 127.0.0.1:2048 "./server/smalltest2" netascii blksize=512 timeout=-4
tsize=404
DATA 127.0.0.1:2048:1025 1
```

Výstup ve Wireshark:

```
1    0.000000000    127.0.0.1    127.0.0.1    UDP    96    2048 → 69 Len=54
2    0.000556811    127.0.0.1    127.0.0.1    UDP    76    1025 → 2048 Len=34
3    0.002207079    127.0.0.1    127.0.0.1    UDP    450    2048 → 1025 Len=408
4    0.002428403    127.0.0.1    127.0.0.1    UDP    47    1025 → 2048 Len=5
```

Test na větším souboru (2000 bytů) nad základní velikost 512 bytů ale menší než maximum jednoho TFTP paketu s option blocksize.

Příkaz:

```
./bin/tftp-client -p 69 -h 127.0.0.1  -t ./bin/client/mediumtest -f mediumtest
```

Výstup na straně klienta:

```
OACK 127.0.0.1:1025: blksize=40960 timeout=1 tsize=2000
DATA 127.0.0.1:1025:2048 1
```

Výstup na straně serveru:

```
RRQ 127.0.0.1:2048 "./server/mediumtest" netascii blksize=0 timeout=1 tsize=0
ACK 127.0.0.1:2048 0
ACK 127.0.0.1:2048 1
```

Výstup ve Wireshark:

```
1    0.000000000    127.0.0.1    127.0.0.1    UDP    96    2048 → 69 Len=54
2    0.001774515    127.0.0.1    127.0.0.1    UDP    76    1025 → 2048 Len=34
3    0.002004425    127.0.0.1    127.0.0.1    UDP    47    2048 → 1025 Len=5
4    0.003434041    127.0.0.1    127.0.0.1    UDP    2046    1025 → 2048
Len=2004
5    0.003694539    127.0.0.1    127.0.0.1    UDP    47    2048 → 1025 Len=5
```

Příkaz:

```
./bin/tftp-client -p 69 -h 127.0.0.1  -t mediumtest2 ./bin/client/mediumtest
```

Výstup na straně klienta:

```
OACK 127.0.0.1:1025: blksize=41984 timeout=1 tsize=2002
ACK 127.0.0.1:1025 1
```

Výstup na straně serveru:

```
WRQ 127.0.0.1:2048 "./server/mediumtest2" netascii blksize=41984 timeout=1
tsize=2002
DATA 127.0.0.1:2048:1025 1
```

Výstup ve Wireshark:

```
1    0.000000000    127.0.0.1    127.0.0.1    UDP    97    2048 → 69 Len=55
2    0.000675362    127.0.0.1    127.0.0.1    UDP    76    1025 → 2048 Len=34
3    0.002617228    127.0.0.1    127.0.0.1    UDP    2050    2048 → 1025
Len=2008
4    0.002829214    127.0.0.1    127.0.0.1    UDP    47    1025 → 2048 Len=5
```

Test na souboru vetším než maximum pro UDP a TFTP (150 000 bytů).

Příkaz:

```
./bin/tftp-client -p 69 -h 127.0.0.1  -t ./bin/client/bigtest -f bigtest
```

Výstup na straně klienta:

```
OACK 127.0.0.1:1025: blksize=65500 timeout=1 tsize=18928
DATA 127.0.0.1:1025:2048 1
DATA 127.0.0.1:1025:2048 2
DATA 127.0.0.1:1025:2048 3
```

Výstup na straně serveru:

```
RRQ 127.0.0.1:2048 "./server/bigtest" netascii blksize=0 timeout=1 tsize=0
ACK 127.0.0.1:2048 0
ACK 127.0.0.1:2048 1
ACK 127.0.0.1:2048 2
ACK 127.0.0.1:2048 3
```

Výstup ve Wireshark:

```
5    5.310018406    127.0.0.1    127.0.0.1    UDP    93    2048 → 69 Len=51
6    5.312519823    127.0.0.1    127.0.0.1    UDP    76    1025 → 2048 Len=34
7    5.312803526    127.0.0.1    127.0.0.1    UDP    47    2048 → 1025 Len=5
8    5.332441074    127.0.0.1    127.0.0.1    UDP    65546    1025 → 2048
Len=65504
9    5.335155839    127.0.0.1    127.0.0.1    UDP    47    2048 → 1025 Len=5
10    5.356319669    127.0.0.1    127.0.0.1    UDP    65546    1025 → 2048
Len=65504
11    5.358269615    127.0.0.1    127.0.0.1    UDP    47    2048 → 1025 Len=5
12    5.360422209    127.0.0.1    127.0.0.1    UDP    19046    1025 → 2048
Len=19004
13    5.361322167    127.0.0.1    127.0.0.1    UDP    47    2048 → 1025 Len=5
```

Příkaz:

```
./bin/tftp-client -p 69 -h 127.0.0.1  -t bigtest2 ./bin/client/bigtest
```

Výstup na straně klienta:

```
OACK 127.0.0.1:1025: blksize=65500 timeout=34 tsize=18761
ACK 127.0.0.1:1025 1
ACK 127.0.0.1:1025 2
ACK 127.0.0.1:1025 3
```

Výstup na straně serveru:

```
WRQ 127.0.0.1:2048 "./server/bigtest2" netascii blksize=65500 timeout=34
tsize=18761
DATA 127.0.0.1:2048:1025 1
DATA 127.0.0.1:2048:1025 2
DATA 127.0.0.1:2048:1025 3
```

Výstup ve Wireshark:

```
1    0.000000000    127.0.0.1    127.0.0.1    UDP    94    2048 → 69 Len=52
2    0.000574199    127.0.0.1    127.0.0.1    UDP    76    1025 → 2048 Len=34
3    0.020899467    127.0.0.1    127.0.0.1    UDP    65546    2048 → 1025
Len=65504
4    0.033840571    127.0.0.1    127.0.0.1    UDP    47    1025 → 2048 Len=5
5    0.051295857    127.0.0.1    127.0.0.1    UDP    65546    2048 → 1025
Len=65504
6    0.053038437    127.0.0.1    127.0.0.1    UDP    47    1025 → 2048 Len=5
7    0.055116368    127.0.0.1    127.0.0.1    UDP    19058    2048 → 1025
Len=19016
8    0.055794020    127.0.0.1    127.0.0.1    UDP    47    1025 → 2048 Len=5
```

Testování několika chyných hlášek:

Příkaz:

```
./bin/tftp-client -p 69 -h 127.0.0.1  -t ./bin/client/bigtest -f bigtest
```

Výstup na straně klienta:

```
ERROR: Selected file already exists, can't rewrite: ./bin/client/bigtest
```

Příkaz:

```
./bin/tftp-client -p 69 -h 127.0.0.1  -t ./bin/client/bigtest2 -f bigtest12
```

Výstup na straně klienta:

```
ERROR 127.0.0.1:1025:2048 1 "File not found"
```

Příkaz:

```
./bin/tftp-client -p 69 -h 127.0.0.1  -t bigtest ./bin/client/bigtest
```

Výstup na straně klienta:

```
ERROR 127.0.0.1:1025:2048 6 "File already exists"
```

3. Výsledek testování

Z testů vychází že hlavní funkcionalita jak serveru tak klienta funguje a odbornější testování během vývoje pomohlo vychytat spoustu menší problémů s programy.

5. Závěr

Pro příští implementaci podobného projektu určitě budu volit jiný programovací jazyk, preferovaně objektový, i když bych se ho musel naučit.

Znalosti získané z tohoto projektu budou v budoucnu užitečné jako například jak linux systémy řeší socket nebo strukturování paketů.

Projekt až na několik vyjímek hodnotím kladně a jsem spokojen s tím co jsem byl schopným pod danými podmínkami zvládnout ale rád bych měl ještě více času na to ho dokončit.

6. Zdroje

[1] Sollins, K., "The TFTP Protocol (Revision 2)", STD 33, RFC 1350 (Online at https://datatracker.ietf.org/doc/html/rfc1350), MIT, July 1992.
[2] Malkin, G., and A. Harkin, "TFTP Option Extension", RFC 2347 (Online at https://datatracker.ietf.org/doc/html/rfc2347), May 1998.
[3] Malkin, G., and A. Harkin, "TFTP Blocksize Option", RFC 2348 (Online at https://datatracker.ietf.org/doc/html/rfc2348), May 1998.
[4] Malkin, G., and A. Harkin, "TFTP Timeout Interval and Transfer Size Options", RFC 2349 (Online at https://datatracker.ietf.org/doc/html/rfc2349), May 1998.