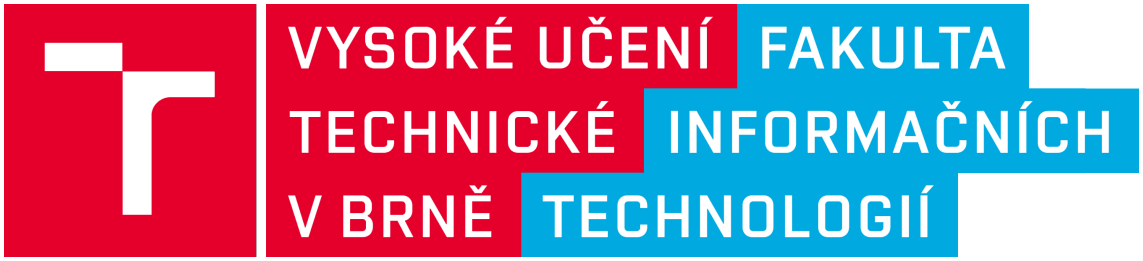
Síťové aplikace a správa sítí

Dokumentace k projektu

Varianta TFTP klient

Jakub Šuráň

xsuran07



Obsah

Kapitola 1

Úvod

Náplní této dokumentace je blíže představit realizaci mého projektu do předmětu Síťové aplikace a správa sítí. Hlavní náplní tohoto projektu byla implmentace TFTP klienta. Tedy aplikace, která je schopná přenášet soubory z a na server pomocí velmi jednoduchého protokolu TFTP.

Prvním krokem řešení projektu bylo podrobně se obeznámit s TFTP protokol. Tato část zahrnovala studium relevantních (především RFC) dokumetů. Podrobně se shrnutí nastudovaných poznatků o TFTP protokolu věnuje kapitola 2.

Následovala samotná implmentace. Jako jazyk implementace byl zvolen C++. Řešení je založe na použití BSD soketů. Rozšíření multicast nebylo implementováno v žádné míře. Podrobně se implementačním detailům věnuje kapitola 3.

Poslední fází bylo náležité testování správnosti aplikace. O tomto blíže pojednává kapitola 4.

Kapitola 2

Teorie

V rámci této kapitoly je blíže představen protokol TFTP. Důraz je kladen zejména na popis jeho stavové logiky, používaných typů paketů a příkladů jeho použití. Postupně jsou také probrány formální dokumenty, kde byly specifikovány důležité části tohoto protokolu.

2.1 Představení protokolu

TFTP (Trivial File Transfer Protocol) představuje jednoduchý protokol pro přenos souborů. V porovnání s jinými protokoly stejného zaměření (např. FTP) je jeho implementace mnohem jednodušší a výsledný program má relativně malé paměťové nároky. Umožňuje provádět zapis na, případně čtení souborů ze serveru. Naopak pomocí něho není možné získat obsah adresářů, není podporováno šifrování, apod. Z této charakteristiky se odvíjí jeho hlavní případy použití. Především se jedná o situace, kdy není vyžadována pokročilá funkcionalita a jsou k dispozici velmi limitované zdroje (především paměť). Jedním z nějčastějších případů užití TFTP proto je bootování bezdiskových počítačů ze sítě (protokoly BOOTP, PXE, atd.). Mezi další klasické použití patří přenos konfiguračních souborů, případně firmwaru v podobě binárních souborů pro síťová zařízení (např. router) [9].

2.2 TFTP verze 1

TFTP protokol byl poprvé standardiován roku 1980 v dokumentu IEN 133, kde byla podrobně popsána stavová logika i struktura jednotlivých paketů. Celá tato podkapitola proto vychází z tohoto dokumentu - [7].

2.2.1 Stavová logika TFTP

TFTP protokol je postaven nad protokolem UDP. Jelikož je UDP bezstavový, je nutné veškerou stavovou logiku řešit přímo na aplikační úrovni. Toho je docíleno tím, že soubor je přenášen pomocí datových bloků konstantní délky 512 bajtů. Po přijetí každého datového bloku musí druhá strana poslat paket s potrvzením, čímž signalizuje, že může být vyslána další datový blok. Bez obdrženého potrvzení nemůže být poslán další datový blok. Pokud blok obsahuje méně než 512 bytů, znamená to, že se jedná o poslední blok přenášeného souboru (přijetí tohoto paketu musí být také

následně potvrzeno). Díky tomuto mechanismu je zaručeno, že při přenášení určitého datového bloku je jisté, že všechny předchozí bloky souboru byly úspěšně doručeny a potrvzeny. U strany přijímající soubor proto není nutné řešit pořadí doručených bloků. Zároveň je díky tomu snadné ošetření případné ztráty jakéhokoli paketu (s datovým blokem nebo potrvzením) v síti. U očekávaného příjemce ztraceného paketu po určité době vyprší časovač a tato strana znovu pošle svůj poslední paket. Původní odesílatel ztraceného paketu následně zareaguje znovuposláním ztraceného paketu.

Pokud během komunikace dojde u jakékoli ze stran k chybě (např. bylo vyžádáno čtení neexistujícího souboru, obdržení nesprávně formátovaného paketu, atd), tato strana pošle paket ozamující chybu. Tento paket zpravidla vede k ukončení celé komunikace a není nijak potrvzován. Po jeho odeslání se příslušná strana může okamžitě ukončit a je tedy možné, že se tento paket ztratí v síti a druhá strana jej nikdy nedostane. Existuje pouze jedna situace, kdy poslání paketu oznamujícího chybu nevede k ukončení komunikace. Pokud je u obdrženého paketu detekován nesprávný zdrojový port, dojde k poslání paketu oznamujícího chybu původci problémového paketu. Původní komunikace, narušná nesprávným paketem, však ukončena není.

2.2.2 Popis komunikace

TFTP definuje pět druhů paketů - požadavek na čtení (RRQ), požadavek na zápis (WRQ), paket s potvrzením (ACK), pakety s datovými bloky (DATA) a pakety oznamující chyby (ERROR). V dalším textu budou pro identifikaci paketů jednotlivých typů využívány zkratky uvedené v závorkách v předchozí větě. Typ paketu je vždy uložen v hlavičce TFTP paketu (první dva bajty TFTP paketu). Další obsah pakutů již závisí na jeho konkrétním typu (pro všechny bude rozebráno dále). Komunikaci vždy zahajuje klient. Nejprve serveru posílá RRQ nebo WRQ paket. Pokud se jedná o validní požadavek, server zahájí komunikaci. V případě RRQ, server odpoví DATA paketem s prvním blokem dat vyžádaného souboru. V případě WRQ odpoví server ACK paketem (s blokem číslo 0). Následně se obě strany střídají ve výměně DATA a ACK paketů dokud nedojde k ukonční komunikace. Oba druhy těchno paketů (tj. DATA a ACK) obsahují číslo bloku, který obsahují, resp. potrvzují. Tato čísla bloků představují nepřerušenou rostoucí posloupnost, která začíná na hodnotě 1 (výjimkou je ACK paket odeslaý jako odpověď na WRQ - obsahuje číslo bloku 0).

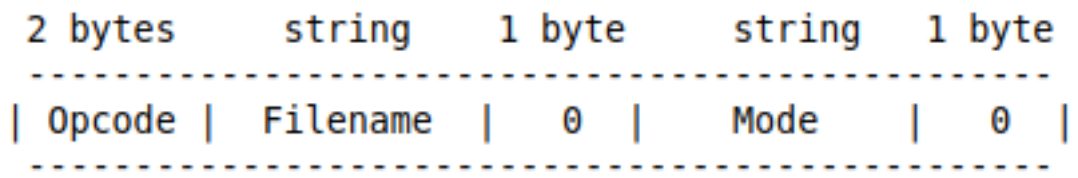
TFTP používá k identifikaci stran účatnících se přenosu dvě speciální čísla, tzv. TID (transfer identifiers). Tato čísla se používají jako porty u UDP paketu. Hodnoty TID by se měly volit náhodně, aby byla malá pravděpodobnost volby stejného čísla u více různých TFTP spojení. Navázání komunikace pak probíhá následovně. Klient si nejprve zvolí svůj TID, který se použije jako zdrojový port UDP datagramu paketu s požadavkem na čtení/zápis. Jako cílový port se použije port, na kterém daný server naslouchá (zpravidla 69 - tato hodnota je vyhrazena pro TFTP protokol). Server si následně zvolí svůj vlastní TID pro toto spojení a použije tuto hodnotu jako zdrojový port své odpovědi (jako cílový port použije TID klienta). Pro další komunikaci se již dále používají hodnoty obou zvolených TID.

2.2.3 TFTP pakety

Jak již bylo řečeno, TFTP protokol používá pět druhů paketů. Každý z nich obsahuje TFTP hlavičku, kde je uveden typ daného paketu. Dále jsou v daných paketech data

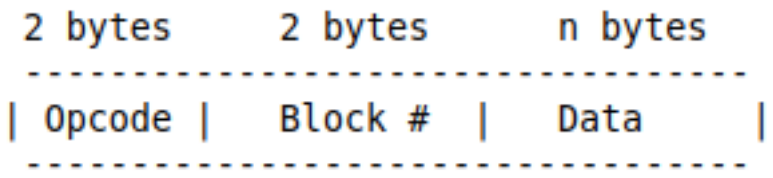
specifická pro daný typ.

RRQ a WRQ pakety mají stejný formát (podrobně je možné vidět na obrázku 1). První dva bajty udávají typ paketu - pro RRQ paket je zde hodnota 1, pro WRQ paket je zde hodnota 2. Za typem následuje jméno souboru, ktrý se bude přenášet. Jmnéno souboru je zadáno jako posloupnost ASCII znaků ukončená nulovým bajtem. Dále následuje určení módu přenosu, opět ve formě posloupnosti ASCII znaků ukončené nulovým bajtem. Jako platné hodnoty pro mód přenosu je možné použít tyto řetězce (bez ohledu na velikost písmen) - netascii, octet a mail. Netascii mód znamená, že soubor bude přenášen ve formě NETASCII znaků a očekává se, že přijímající strana si soubor převede do svého vlastního formátování. NETASCII představuje 8-bitové rozšíření původního 7-bitového ASCII a bylo definováno v RFC-764. Skládá se z tisknutelných znaků, mezery a osmi speciálních znaků. Tento mód vyžaduje, aby v přenášeném souboru byly konce řádku složeny z dvojice znaků CR+LF. Dále je nutné, aby každý CR znak byl následován LF znakem nebo nulovým bajtem[6]. Octet mód znamená, že soubor bude přenášen ve formě surových 8 bitových bajtech. Je tedy zaručeno, že jak příjemce, tak i odesílatel budou mít u sebe po provedení transferu shodné verze souborů (tj. nebude rozdíl v kódování atd.). Mail mód je stejný jako netascii mód, pouze jméno souboru je zde interpretováno jako emailová adresa příjemce a je nutné jej pouze s WRQ.



Obrázek 1: Formát RRQ a WRQ paketů

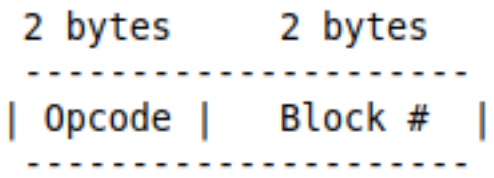
Bloky dat jsou přenášené v DATA paketech (formát podrobně na obrázku 2). Pro DATA paket je v hlavičce (tedy v políčku opcode) hodnota 4. Za hlavičkou následuje číslo přenášeného datového bloku. Jak již bylo řečeno, toto číslo začíná na hodnotě 1 a postupně se inkrementuje pro každý další datový blok. Díky tomuto políčku je tedy velmi snadné rozlišit případné duplicitné DATA pakety. Nakonec DATA paket obsahuje samotná data. Těch muže být od 0 do 512 bajtů. Pokud je dat právě 512, nejedná se o poslední datový blok. V opačném případě se jedná o signál, že daný paket obsahuje poslední datový blok přenášeného souboru.



Obrázek 2: Formát DATA paketu

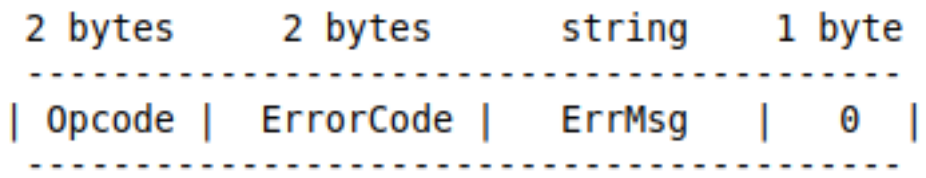
Dalším typem paketů jsou ACK pakety (formát na obrázku 3). Hlavička pro tento typ paketů obsahuje hodnotu 3. Za ní se nachází číslo datového bloku, který daný ACK paket potrvzuje. Posláním ACK paketu s číslem bloku x strana přijímající soubor signalizuje, že úspěšně obdržela DATA paket s blokem číslo x a žádá o zaslání DATA paketu s blokem x+1. Z pohedu strany přijímající soubor je každé obdržení

DATA paketu s blokem číslo y chápano tak, že předchozí ACK paket s blokem číslo y-1 byl úspěšně doručen a nyní je nutné odeslat nový ACK paket s blokem číslo y.



Obrázek 3: Formát ACK paketu

Posledním typem je ERROR paket (formát na obrázku 4). Tomuto typu paketu v hlavičce přísluší hodnota 5. Za ní následuje číselný kód chyby, která nastala (např. soubor nenalezen, porušní přístupových práv, atd.). Za ním následuje popis chyby ve formě pochopitelné pro člověka - sekvence ASCII znaků ukončená nulovým bajtem. ERROR paket se chápe jako potrvzení ke všem ostatním typům paketů (platí tedy i pro RRQ a WRQ pakety).



Obrázek 4: Formát ERROR paketu

2.3 TFTP verze 2

TFTP protokol byl dále rozšířen v RFC 1350. Zde bylo opraveno několi neduhů původního návrhu. Jedna relativně velká chyba v návhu je známa jako tzv. Syndrom čarodějova učně (Sorcerer's Apprentice Syndrome). Dále bude tento syndrom označován jako SAS. Přestože SAS nezpůsobuje rozbití přenosu samotného (tj. pokud se transfer celý dokončí, přenesený soubor bude v pořádku), jedná se o velmi závažný problém. Může totiž způsobit zahlcení sítě.

Podstatou SAS je, že původní návrh TFTP protokolu vyžadoval, aby příjemce po přijetí každého paketu poslal odpovídající odpověď. Tedy přijetí každého ACK paketu vede k poslání odpovídajícího DATA paketu a naopak. To samo o sobě problém není. Problémem však je, že TFTP je postaveno nad bezstavovým UDP, tudíž hojně využívá časovače - po odeslání paketu se určitý čas čeká na odpověď a pokud nedorazí před vypršením časovače, dojde k znovuodeslání posledního paketu. Kombinací těchto dvou faktorů mohla např. nastat tato situace. Klient K zahájil čtení souboru ze serveru S. Ze začátku komunikace probíhá bez problémů - server úspěčně odeslal i-1 datových bloků a klient stejný počet ACK paketů. Avšak po odeslání i-tého bloku se příslušný DATA paket v síti zpozdí (nikoli ztratí, pouze zpozdí). Na straně serveru dojde k vypršení časovače a znovuposlání i-tého DATA paketu. Klient posléze obdrží oba tyto DATA pakety a dle specifikace TFTP je nucen oba potvrdit ACK pakety, neboť jsou oba validní. Server následně obdrží dva ACK pakety a na oba odpoví atd. Dojde tedy ke zduplikování provozu, což vede k většímu

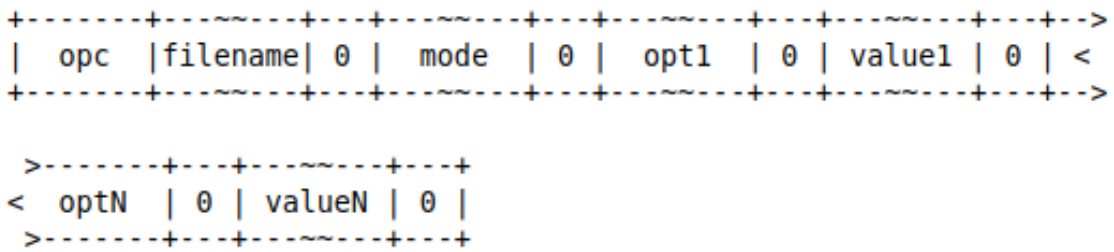
zatížení sítě. Z toho důvodu je velmi pravděpodobné, že se budou opožďovat další pakety - tj. další duplikace provozu a situace se bude ještě zhoršovat.[1]

Naštěstí, řešení SAS není příliš složité a bylo zakomponováno do 2. revize TFTP. Jádrem řešení je snaha o rozbití cyklu popsaného v předchozím odstavci. Specifikace TFTP protokolu proto byla upravena tak, aby pouze první obdržený ACK paket s číslem bloku i způsobil odeslání DATA paketu s blokem číslo i+1. Ostatní případné kopie stejného ACK paketu jsou ignorovány. DATA paket tedy může být znovuposlán pouze na základě vypršní časovače a nikoli na základě duplicitních ACK paketů.[8]

2.4 Mechanismus rozšiřujících možností

Dalšího důležitého rozšíření se TFPT protokol dočkal v RFC 2347. Tento dokument představil mechanismus, pomocí něhož se server a klient můžou před zahájením přenosu domluvit na dodatečných parametrech. Jedná se o rozšíření, které je zpětně kompatibilní - není nutné rozšíření používat. Navíc servery, které neimplementují tato rozšíření, jsou zpravidla bez problémů schopny komunikovat s klienty implementující tato rozšíření (servery je jednoduše ignoruje)[3].

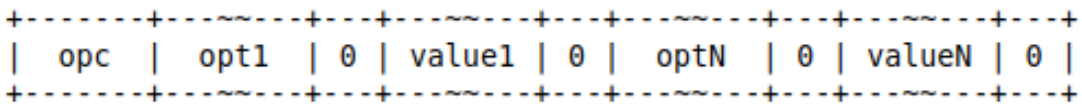
Zbytek podkapitoly je založen na [4]. Rozšiřující možnosti fungují následujícím způsobem. Pokud si klient přeje navrhnout serveru použití nějaké rozšiřující možnosti, upraví svůj RRQ nebo WRQ dle formátu uvedeného na obrázku 5. Do RRQ nebo WRQ paketu umístí klasická políčka (tj. hlavičku, jméno souboru a mód) a za nimi může uvést libovolný počet párů "název rozšiřující možnosti"a "hodnota". Jak "název rozšiřující možnosti", tak i "hodnota"jsou specifikovány pomocí sekvence ASCII znaků (bez ohledu na velikost znaků) ukončené nulovým bajtem. Rozšiřujících možností může být v paketu libovolné množství, ale maximální velikost paketu je zastropována na 512 bajtů. Na serveru následně je, jak se rozhodne naložit s tímto návrhem.



Obrázek 5: Formát RRQ a WRQ paketů s rozšiřujícími možnostmi

Aby server mohl rozumně podporovat rozšiřující možnosti, byl definován nový druh paketu - paket s potrvrzením rozšiřujících možností (OACK). Formát OACK paketu je na obrázku 6. Hlavičce tohoto typu paketu přísluší hodnota 6. Za ní následuje libovolný počet párů "název rozšiřující hodnoty"a "hodnota". Tyto páry představují rozšíření, která se server rozhodl akceptovat. Nemusí zde tedy být všechna rozšíření, která navrhoval klient (rozšíření, která nepodporuje server vynechal).

Díky tomuto doplnění může server na žádost klienta s rozšiřujícími možnostmi odpovědět třemi způsoby. Může poslat OACK paket s rozšířeními, která se rozhodl akceptovat. Pokud se klientovi nelíbí, jaká rozšíření server akceptoval, může spojení



Obrázek 6: Formát OACK paketu

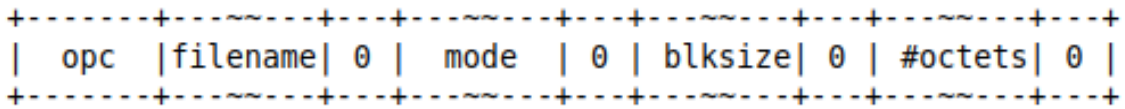
následně ukončit pomocí ERROR paketu se speciálním novým typem chyby (číslo 8). V opačném případě komunikace probíhá klasickým způsobem - při žádosti o čtení odpoví ACK paketem s číslem bloku 0, při žádost o zápis prvním DATA paketem. Druhou možností je, že server rozšiřující možnosti nepodporuje a proto je ignoruje. Odpoví proto tak, jakoby rozšiřující možnosti v žádosti od klienta vůbec nebyly - na RRQ odpoví prvním DATA paketem, na WRQ ACK paketem. Třetí možností je, že server nedokázal zpracovat žádost od klienta a odpoví ERROR paketem. Tím je komunikace ukončena

2.5 Rozšíření blksize

Jedna z nejdůležitějších rozšiřujících možností byla definována v rámci RFC 2348. Jedná se o rozšíření blksize. Smyslem tohoto rozšíření je vyřešit omezení původního návrhu protokolu. V původním návrhu byla velikost datového bloku pevně zastropována na 512 bajtů. Vzhledem k tomu, že se k číslování bloků je využíváno 16-bitové číslo, maximální velikost přenášeného souboru je 512*65535 = 32MB. A právě tento problém se snaží vyřešit rozšíření blksize. Umožňuje serveru a klientovi domluvit se na jiné velikost bloku než je výchozích 512 bajtů.

Zbytek podkapitoly je založen na [2]. Pokud chce klient navrhnout serveru vlastní velikost bloku, upraví úvodní RRQ nebo WRQ paket tak, že do sekce vyhrazené rozšiřujícím možnostem dvojici řetězců "blksize"a navrhovanou velikost bloku zapsanou v ASCII (viz obrázek 7). Jako validní velikosti bloku je možné použít jakoukoli hodnotu z rozmezí 8 až 65464.

Pokud server podporuje rozšíření blksize a rozhodne se akceptovat hodnotu navrhovanou klientem, pošle klientovi OACK paket. Tento OACK paket obsahuje i potrvzení pro rozšíření blksize, kde jako hodnota tohoto rozšíření je uvedena hodnota rovná nebo menší než hodnota původně navrhnutá klientem. Klient se následně rozhodne, zda se mu hodnota blksize potvrzená serverem líbí. Pokud ne, pošle zpět serveru ERROR paket s číslem chyby 8. Jinak pokračuje klasicky v komunikaci, avšak používá novou velikost bloku.

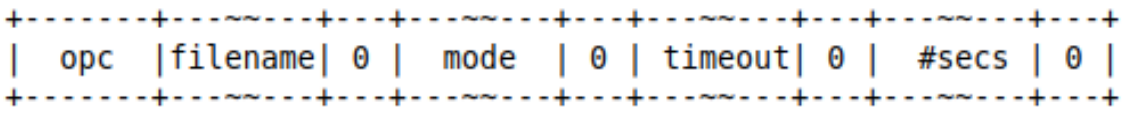


Obrázek 7: Formát paketu s rozšířením blksize

2.6 Rozšíření timeout a tsize

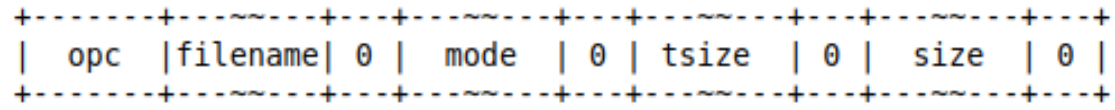
Další rozšiřující možnosti TFTP protokolu byly definovány v RFC 2349. Jedná se o rozšíření timeout a tsize. Rozšíření timeout umožňuje, aby se klient a server domluvli na intervalu časovače. Rozšíření tsize vedle toho umožňuje straně přijímající přenášený soubor zjistit velikost souboru ještě před samotným zahájením vlastního přenosu. Zbytek podkapitoly vychází z [5].

Pokud chce klinet se serverem vyjednávat o velikost intervalu časovače, přidá do sekce pro rozšíření úvodního RRQ nebo WRQ paketu dvojici řetězců "timeout"a navrhovanou velikost intervalu v ASCII (viz obrázek 8). Navrhovaná velikost musí být v intervalu 1 až 255, aby byla považována za validní. Pokud se server podporuje toto rozšíření a rozhonde se navrhovanou velikost intervalu akceptovat, přidá do OACK paketu potvrzení mj. pro možnost "timeout". Hodnota v OACK paketu musí být stejná jako navrhoval klient.



Obrázek 8: Formát paketu s rozšířením timeout

Pro rozšíření tsize je situace velmi podobná, avšak o trochu komplikovanější. Mírně se totiž liší sémantika tohoto rozšíření pro RRQ a WRQ pakety. Pokud si klient přeje vyjednávat o tomto rozšíření, v obou případech do svého paketu s požadavkem přidá do sekce pro rozšíření pár "tsize"a velikost souboru v ACII (viz 9). A právě vyplnění položky velikost souboru se liší pro RRQ a WRQ pakety.



Obrázek 9: Formát paketu s rozšířením tsize

V případě RRQ zde klient uvede "0". V případě, že server podporuje rozšířeni tsize, tak v rámci OACK paketu uvede jako hodnotu rozšíření tsize velikost přenášeného souboru v bajtech. Klient si následně může zkontrovat, zda má na svém disku potřebné množství volné paměti. Pokud ne, odpoví ERROR paketem s číslem chyby 3. Jinak se normálně pokračuje v komunikaci.

Naopak v případě WRQ klient vyplní jako hodnotu rozšíření tsize velikost soubrou v bajtech. Následně je opět na serveru, aby si zkontroloval, že má dostatek místa. Pokud ano a podporuje rozšíření tsize, zopakuje v OACK paketu velikost souboru uvedenou klientem.

Kapitola 3

Implementace

V této kapitole bude blíže představena implementační část projektu. Jak již bylo zmíněno, jako jazyk implementace bylo zvoleno C++. Postupně zde budou v krátkosti diskutovány důležité třídy vytvořené v rámci řešení a další implementační detaily.

3.1 Třída terminal

Jádrem celého řešení je třída *terminal*. Jak již název napovídá, úkolem této třídy je vytvořit a spravovat interaktivní terminál celé aplikace. Zadávací promt tohoto terminálu je blokující - je nutné počkat na dokončení poslední operace, než je uživateli nabídnuto zadání dalšího příkazu.

Samotné zpracovávání příkazů probíhá následovně. Nejprve je ze standardního vstupu od uživatele načnen příkaz. Jako příkaz je chápana sekvence znaků ukončená znakem konce řádku. Následně je tento příkaz rozdělen do jednotlivých podčástí (jako separátory jsou uvažovány bílé znaky). Tyto podčásti jsou dále poskytnuty instanci třídy *parser* (viz dále), která se pokusí příkaz rozpoznat a zkontrolovat jeho syntakticku správnost. Pokud je příkaz vyhodnocen jako validní, třída *terminal* jej následně provede a vyzve k uživatele k zadání dalšího příkazu.

Podporovány jsou tři druhy příkazů - "help", "quit"a TFTP požadavek. Příkaz "help"vypíše jednoduchou nápovědu k používání terminálu. Příkaz "quit"způsobí ukončení terminálu a tedy i celé aplikace (stejného chování je možné docílit i zadáním EOF - na linuxu např. pomocí CRTRL+D). Poslední příkaz je nejzajímavější - jedná se o samozný TFTP požadavek, tedy čtení nebo zápis souboru. K jeho realizaci je použita instance třídy *tftp_client* (viz dále). Pokud TFTP požadavek selže, nedojde k ukončení terminálu - uživatel má možnost zadat další příkaz.

3.2 Třída parser

Třída *parser* primárně slouží ke zpracování a zkontrolování příkazů "help"a "quit". U těchto příkazů záleží na velikosti znaků - vyžaduje se, aby byly všechny malé. Dále se kontroluje, že na načteném řádku se kromě bílých znaků nenachází žádné další - tj. oba tyto příkazy je možné používat pouze samostatně, nelze je s ničím kombinovat.

Pokud třída *parser* neznámý příkaz (tj. nedokáže jej zpracovat), předpokládá se, že se jedná o TFTP požadavek. Příkaz je proto přeposlán instanci třídy *tftp_parameters*.

3.3 Třída tftp_parameters

Třída *tftp_parameters* slouží ke zpracování parametrů TFTP požadavku. Akceptované parametry budou nyní představeny.

Mezi základní parametry patří specifikace, zda se má jednat o čtení (přepínač -R) nebo zápis (přepínač -W). Aby byl TFTP požadavek jako celek validní, musí obsahovat právě jeden z těchto přepínačů.

Dalším důležitým parametrem je specifikace přenášeného souboru. Toho lze docílit použitím přepínače -d absolutni_cesta/nazev_souboru. Uvedení tohoto přepínače je také vyžadováno. Poskytnutý argument tohoto přepínače je interpretován takto. První část, absolutni_cesta, specifikuje absolutní cestu na serveru - tj. kde se má hledat soubor při požadavku na čtení, případně kam má být sobour uložen uložen při požadavku na zápis. Druhá část, nazev_souboru, specifikuje název přenášeného souboru. Na straně klienta platí následující. Při požadavku na čtení bude přenášený soubor uložen do aktuálního lokálního adresáře. Při požadavku na zápis se přenášený soubor také hledá v aktuálním lokálním adresáři. Platí tedy, že argument přepínače -d (absolutni_cesta/nazev_souboru) se vždy stane součástí úvodního paketu TFTP komunikace a slouží ke specifikaci uložení přenášeného souboru na straně serveru. Na straně klienta se pracuje vždy pouze se souborem s názvem nazev_souboru v aktuálním lokálním adresáři.

Ke specifikaci módu přenosu se používá přepínač -c mode. Jako hodnoty argumentu jsou akceptovány pouze hodnoty "ascii" (případně "netascii") a "binary" (octet) uvedené malými písmeny.

Dále je možné použít parametry k nastavení rozšiřujících TFTP možností. Rozšíření timeout odpovídá přepínač -t timeout. Hodnota *timeout* může být pouze z rozsahu 1 až 255 (dle RFC 2349). Pro vyžádání rozšíření blksize lze použít přepínač -s size. Jako hodnota *size* jsou akceptovány pouze hodnoty z intervalu 8 až 65464 (dle RFC 2348). Poku není přepínač -s použit, uvažuje se použití bloků o velikost 512 bajtů.

Dále je možné využít přepíač -a adresa, port. Tento přepínač slouží ke specifikaci serveru, kterému bude požadavek odeslán. Podporovány jsou jak ipv4, tak i ipv6 adresy. Pokud není tento přepínač použit, implicitně se uvažuje jako adresa ipv4 localhost (127.0.0.1) a číslo port 69 (vyhrazená hodnota pro TFTP).

Posledním možným přepínačem je -m. Tento parameter slouží k vyžádání si přenosu skrze multicast (RFC 2090). Tento přepínač je možné použít, ale je bez efektu - podpora pro rozšíření multicast není implementována.

3.4 Třída tftp_client

Třída *tftp_client* tvoří nejdůležitější část celé implementace. Má na starosti zajištění a řízení TFTP komunikace se specifikovaným serverem. Kromě toho uživatel průběžně informuje o průběhu (časové razítka odeslaných a přijatých paketů + krátký rozobor jejich obsahu) a výsledku komunikace. Základní stavební kámen této třídy tvoří BSD sokety. Nyní budou v krátkosti představeny důležité detaily implementace.

Svému okolí jsou instance této třídy dostupné přes metodu *communicate* - slouží k vyžádání si TFTP komunikace s konkrétními parametry. Na zažátku se vždy provede nezbytná inicializace. Dojde k vytvoření soketu (UDP datagram s nastavenou adresou serveru) a otevření přenášených souborů (jak již bylo řečeno, na straně klienta se přenášené soubory při čtení vždy ukládají do aktuálního lokálního ad-

resáře a při zápisu se také hledají v lokálním adresáři). Kromě toho také dojde ke zpracování rozšiřujících možností úvodního TFTP požadavku (tsize, timeout a blksize). U rozšíření timeout a blksize platí, že jsou pouzžita pouze pokud si je uživatel vyžádal pomocí patřičných přepínačů. Rozšíření tsize se použije vždy, kdy se jako mód přenosu použije "binary"(tj. při vyžádání si "netascii"se vynechá). Poslední částí přípravné fáze je kontrola, že vyžádaná velikost datového bloku může být uspokojena i rozhraním s nejmenším MTU. K tomu je využita kombinace funkcí *getifaddrs*[1] (získání rozhraní) a *ioctl*[2] (získání velikosti MTU daného rozhraní). Pokud je velikost bloku vyžádaná uživatelem příliš velká, použije se maximální zjištěná velikost (tj. velikost nejmenšího MTU z dostupných rozhraní mínus velikosti IP, UDP a TFPT hlaviček) a uživatel je varován hláškou o této změnně.

Poté již následuje vytvoření paketu s úvodním požadavkem včetně vyžádaných rozšíření a jeho odesláním na server. Při obderžení paketu se neprve zkontroluje, že byl poslán očekávaným serverem a má správnou hodnotu TID (při první odpovědi od serveru se toto nekontroluje a místo toho se hodnota TID uloží). Pokud hodnota TID nesedí, klient pošle odesílateli tohoto nečekaného paketu příslušnou ERROR odpověď a pokračuje v čekání na legitimní odpověď. Stavové řízení TFTP komunikace je zajištěno takto. Při odesílání paketu se do interního atributu uloží očekávaný typ paketu odpovědi. Po přijetí paketu se nejprve zkontroluje, zda typ odpovědi odpovídá očekávanému typu. Pokud ne, dojde k odeslání ERROR paketu a ukončení komunikace. Při kontrole typu odpovědi se počítá i s tím, že někdy může mít odpověď jiný než očekávaný typ a přesto se nejedná o chybu. ERROR paket může být jako odpověď serverem kdykoli validně poslán - v takovém případě dojde ke zpracování paketu a ukončení komunikace (k ukonccení nemusí dojít, pokud se jedná o chybu vzniklou v důsledu vyjednávání o rozšířeních, viz dále). Dále např. jako odpověď na úvodní požadavek s rozšířeními nemusí server poslat OACK paket. Pokud rozšíření nepodporuje, může je ignorovat a pokračovat v komunikaci v klasické komunikace. I s touto možností implementovaný klient počítá. Pokud paket projde těmito kontrolami, zpracuje se a pokud má být odeslána nějaká odpověď (např. na duplicitní ACK paketu se neposílá odpověď), odešle se a celý proces se opakuje.

Jelikož je TFTP postaveno nad UDP protokolem, náležitá pozornost je věnována i problematice timeoutů. Ty jsou v implementaci využívány na několika úrovních. Jeden z timoutů je nastaven ihned po vytvoření používaného soketu pomocí funkce *setsockopt* [3] a nastavení možnosti SO_RCVTIMEO. Díky tomu pak blokující funkce *recvfrom*[4] čeká maximálně specifikovaný časový interval (použito 5 s) než vrátí chybovou hodnotu. Takto je tedy zajištěno, že klient nebude čekat donekončna na odpověď od neexitujícího serveru, případně po předčasném ukončení legitimního serveru. Pokud během čekání dojde k vypršení totoho intervalu, klient znovu odešle svůj poslední paket. K zajištění toho, aby toto znovuodesílání nepokračovalo do nekonečna slouží další timeout. Ten je implementován pomocí časových razítek. Při prvním odeslání paketu se do interního atributu uloží aktuální čas T. Při čekání na odpověď se před vždy před zahájením čekání pomocí *recvfrom* zkontroluje, zda od času T již neuplynula maximální doba (dovoleno 4x znovuodeslat paket) od

[1] https://man7.org/linux/man-pages/man3/getifaddrs.3.html
[2] https://man7.org/linux/man-pages/man2/ioctl.2.html
[3] https://linux.die.net/man/3/setsockopt
[4] https://linux.die.net/man/2/recvfrom

odeslání původního paketu. Pokud již byla přesažena maximální doba čekání, dojde k ukončení komunikace. Při znovuodesílání pakutů je počítáno i s případem, kdy klient sice obdrží paket, ale nejedná se o legitimné paket (paket se špatným TID, duplicitní ACK paket, atd.). Proto se ještě před zahájením čekání na odpověď zkontroluje, zda nemá být znovuodeslán poslední paket.

Poslední důležitou částí je přístup k vyjednávání o rozšiřujících možnostech. Jak již bylo naznačeno, klient počítá i s případy, kdy server rozšiřující možnosti v úvodním požadavku ignoruje a odpoví rovnou prvním DATA paketem při požadavku na čtení nebo ACK paketem při zápisu. V takovém případě se normálně pokračuje v komunikaci s výchozími hodnotami. Pokud server odoví OACK paketem, zkontroluje se, že akceptované rozšíření jsou validně potrvzena (tj. hodnota timetou musí být shodná jako navrhovaná a hodnota blksize musí být rovná nebo menší než navrhovaná). Pokud úspešně projde tato kontrola, klient si uloží obdržené hodnoty a používá je v následné komunikaci. Pro klienta má největší dopad rozšíření blksize. Platí, že implicitně se počítá s výchozí hodnotou (tj. 512 bajtů) a teprve pokud server v rámci OACK paketu potvrdí novou hodnotu, začne i klient počítat s jinou velikostí datového bloku. Po potvrzení nové velikokosti davého bloku nejprve dojde ke kontrolu, zda interní buffery jsou dostatečně velké ke zvládnutí nové velikosti datového bloku. Pokud ne, dojde k jejich zvětšení. Platí, že pokud server na úvodní požadavek s rozšířeními odpoví ERROR paketem s číslem chyby 8 (tj. problém s jedním z rozšíření), nedojde k ukončení komunikace. Místo toho klient znovupošle úvodní požadavek - tentokrát však zahodí jedno z rozšíření. Strategie postupného zahazování rozšíření je implementována takto. Nejprve klient zkontroluje, jesli problematický požadavek obsahoval rozšíření tsize. Pokud ano, vyřadí ho a znovupošle úvodní požadavek bez tohoto rozšíření. Pokud ne, pokusí se stejným způsobem odstranit rozšíření timeout a poslat úvodní požadavek. Pokud problematický požadavek neobsahoval ani rozšíření timeout, pokusí se klient odstranit rozšíření blksize a poslat tedy úvodní požadavek bez rozšiřujících možností (v případě, že problematický požadvek neobsahoval ani rozšíření blksize jedná se zjevně o chybu ze strany sereru a komunikace je ukončena s chybou).

Kapitola 4

Testování

Nepostradatelnou součástí projektu bylo i náležité otestování výsledné aplikace. Jako testovací server byl použit tftpd server[1]. Tento server totiž podporuje připojení pomocí IPV4 i IPV6. Kromě toho podporuje i rozšiřující možnosti tsize, blksize a timeout. Poskytuje tedy vše potřebné ke komplexnímu otestování celé aplikace.

4.1 Testovací prostředí

Bylo použito následující nastavení testovacího prostředí. Klient byl spouštěn skrze program Virtualbox v poskytnutém virtuálním stroji (Ubuntu 20.04). Testovací server běžel na mém lokálním počítači (shodou okolností také Ubuntu 20.04). Pomocí odpovídajícího konfiguračního souboru bylo v případě potřeby upraváno jeho nastavení. Spravný průběh TFTP komunikace byl zachycován a kontrolován pomocí programu Wireshark. V neposlední řadě, pro otestování správnosti přenesených souborů byla používána unixová utilitka *diff*.

Ve výchozím nastavení byl testovací server nakonfigurován s možností –create (umožní vytvářet nové soubory) a bez možnosti –secure (tj. je vyžadováno uvádění absolutné cesty místa uložení na serveru).

Samotné testování probíhalo ve dvou fázích. Nejprve bylo otestováno fungování nad protokolem ipv4. Následně byl testovací server překonfigurován tak, aby byl přístupný skrze ipv6 rozhraní. Tím byla otestována podpora protokolu ipv6.

Níže představené testy tedy byly provedeny jak pro ipv4, tak i ipv6.

4.2 Čtení souborů různých typů a velikostí

V první fázi testování bylo vyzkoušeno, že klient je schopen správně přijímat soubory ze serveru. Toto bylo testováno jak pro mód přenosu "binary"i "ascii". Pro náležité otestování byly vyzkoušny soubory různých typů i velikostí.

Nejprve byl testován přenost obyčejných textových souborů. Kvůli pokrytí různých možností byl přenesen malý soubor (tj. vleze se do jednoho bloku), střední soubor (řádově desítky bloků) i velký soubor (řádově stovky bloků). Otestovány byly i mezní případy - prázdný soubor a soubor o velikost přesně jednoho bloku (tj. očekává se, že se poté pošle jeden prázdný DATA paket).

[1] https://linux.die.net/man/8/tftpd

Následně byly vyzkoušeny i jiné typy souborů. Konkrétně se jednalo o obrázky - různé formáty (png, jpeg a gif) a velikosti. Dále i spefické typy souborů - docx, pdf a zip.

Aby bylo otestováno "reálné"použití klienta (TFTP se primárně používá na přenos binárních a bootovacích souborů), byl otestován i přenos binárních souborů.

Pro každý přenesený soubor bylo pomocí utilitky *diff* ověřeno, že výsledný soubor se neliší od původního - tj. přenos soubor nijak nezměnil. Ve všech případech bylo dosaženo kladného výsledku. To platí pro přenosy v obou přenosových módech ("ascii"i "binary"). Tj. při módu "ascii"je přenášený soubor při transferu zakódován do netascii, ale při ukládání se překóduje do klasické unixové podoby. Toto chování je komplatibilní jak s chováním tfpd serveru, tak i referenčním klientem tftp[2].

4.3 Zápis souborů různých typů a velikostí

Následovalo otestování, že klient se zvládne vypořádat i se zápisem souboru na server. Kvůli pokrytí co nejvíce možností byly opět zvoleny soubory různých typů a velikostí (podobně jako v předešlé části). Analogicky jako v případě čtení, také zde byl otestován přenos v obou podporovaných módech.

V této části byl nejvíce kladen důraz na vyzkoušení, že klient zvládá překódování do netascii (nahrazení konců řádků dvojicí CR+LF, atd.). Důraz byl kladen i na mezní situace - tj. konec řádku (dvojici CR + LF) je nutné rozdělit do dvou bloků, více CR znaků za sebou a správné zakódování skutečné dvojice CR + LF v souboru.

4.4 Základní test rozšiřujících možností

Poté bylo přistoupeno k otestování podpory rozšiřujících možností. Pro tuto část byl server nakonfigurován, aby podporoval všechna tři implementovaná rozšíření - tj. blksize, tsize i timeout.

U rozšíření tsize bylo testování velmi přímočaré. Klient byl implementován tak, aby rozšíření tsize automaticky přidával do všech požadavků, kde je použit mód přenosu "binary"(tj. pro mód "ascii"se nepoužije). Bylo otestováno, že při požadavku na čtení klient přidá do úvodního paketu rozšíření tsize s hodnotou 0 - tj. poptává velikost souboru od serveru. Poté co mu ji v rámci OACK paketu server pošle (u tohoto testování zaručeno), uloží si ji do interního atributu a ve výpisech je dostupná i uživateli. Při požadavku na zápis bylo zkoumáno, zda klient správně zjistí velikost přenášeného souboru a pošle jej serveru. V obou případech testy dopadly úspěšne.

U rozšíření timeout nebylo nutné příliš důkladné testování, neboť toto rozšíření má význam spíše pro server. Proto bylo pouze otestováno, že pokud si uživatel toto rozšíření vyžádá přepínačem, klient jej s příslušnou hodnotou přidá do úvodního paketu. To, že rozšíření je přidáno validně bylo potvrzeno tím, že server jej potrvdí v rámci OACK paketu (v tomto případě zaručeno).

Testování rozšíření blksize bylo nejzajímavější. Nejprve bylo otestováno, že pokud si toto rozšíření vyžádá přepínačem, je skutečně použito. Toto probíhalo podobně jako v případě rozšíření timeout. Násladně bylo vyzkoušno, že pokud server akceptuje novou velikost bloku, klent si po obdržení potvrzení (v rámci OACK paketu) upraví svůj interná atribut uchovávající velikost bloku a v případě potřeby

[2] https://linux.die.net/man/1/tftp

provede i zvětšní interních bufferů. Nakonec se zkoumalo, že klient v rámci komunikace opravu pracuje s novou velikosí bufferu a je schopen v pořádku přijímat soubory. Otestována byla i situace, kdy uživatel zadá klientovi velikost bloku přesahující velilkosti dostupných MTU. V takovém případě klient vypíše varování a použije maximální dostupnou velikost bloku.

4.5 Test odmítnutí rozšiřujících možností

Při této části testování bylo využito toho, že používaný server nabízí možnost skrze konfigurační soubor vypout podporu pro vybraná rozšíření (děje se tak pomocí přepínače –refuce název rozšíření).

V první fázi byla vypnuta podpora pro všechna rozšíření. Následně bylo testováno, že klient se zvládne vypořádat se situací, kdy server všechna navrhovaná rozšíření ignoruje a pokračuje v kumunikaci jakoby je požadavek neobsahoval. Díky tomu bylo potrzeno, že přestože klient navrhuje v úvodním požadavku některá rozšíření (a očekává tedy odpověď pomocí OACK), je schopen akceptovat i "klasické"odpovědi - tj. DATA při čtení nebo ACK. při zápisu.

V další fázi byla v jednu chvíli vypnuta podpora pouze pro jedno rozšíření. Tím bylo testováno, že klient s hodnotami v rozšířeních reálně pracuje pouze a jedině v případě, že mu jej server validně potvrdí. Za všechny možnosti bude uveden příklad s rozšířením blksize. V konfiguraci serveru byla vypnuta podpora pro toto rozšíření. Dásledně byl vygenerován pomocí klient dotaz na čtení souboru obsahující rozšíření blksize a tsize. Server v rámci OACK paketu potvdil pouze tsize rozšíření. Klient si správně poznačil velikost přenášeného souboru a dále s ní pracoval. Naopak, vzhledem k tomu, že blksize nebylo potvzeno, klient v rámci komunikace pracoval správně s výchozí velikostí datového bloku.

Poslední fází bylo otestování toho, že klient je schopen se vypořádat i s tím, že mu server při vyjednávání o rozšířeních odpoví chybou. Z důvodu zjednodušení práci byl pro tento případ dočasně mírně upraven kód klient. Úprava spočívala v tom, že se dočasně odstanily kontroly, že uživatel zadává hodnoty přepínačů TFTP požadavku -t a -s ve správném intervalu. Díky tomu bylo možné vygenerovat požadavek, který obsahoval nevalidní hodnoty u rozšíření timeout a blksize. Server na takový požadavak odpověděl ERROR paketem s číslem chyby 8, nicméně klient správně okamžitě neukončil komunikaci. Místo toho postupně z požadavku odstraňoval rozšíření a odesílal jej znovu. Postpně tak došlo k odstranění všech rozšíření a tedy vytvoření požadavku, který již byl serverem akceptován. Komunikace tak proběhla úspěšně.

4.6 Test chybových stavů

Poslední částí bylo test, že se klient zvládne úspěšně vypořádat s různými chybovými stavy. Do této části spadaly testy snažicí se zadávat neplatné příkazy a špatné kombinace parametrů u TFTP požadavku (případně jejich hodnoty). Dále sem patří i ověření, že klient zvládne i zadání adresy serveru, která sice je validní, ale neodpovídá žádnému skutečnému serveru. Zde ke slovu přišly timeouty - klient čekal na odpověď na svůj požadavek, postupně jej znovu posílal a posléze komunikaci ukončil

jako neúspěšnou. Podobně byla testována i situace, kde se server během komunikace neočekávaně ukončí. I zde zafungovali timeouty klienta.

Kapitola 5

Závěr

V rámci tohoto projektu se úspěšně podařilo naimplementovat a otestovat jednoduchého TFTP klienta. Samotnému procesu impomentace předcházela fáze studia relevantních dokumentů popisující protokol TFTP. Jednalo se zejména o dříve zmíněne RFC. Poté již bylo možné přejít k samotné implementační fázi. Výsledný klient po spuštění nabídne uživateli interaktivní příkazovou řádku, kam je možné zadávat jednoduché příkazy - mj. parametry TFTP požadavku. Jednotlivé TFTP požadavku jsou blokující.

Klient používá TFTPv2, tedy upravenou verzi protokolu, která mj. řeší tzv. Problém čarodějova učně (SAS). Podporována jsou i některá rozšíření tohoto protoklu. Konkrétně se jedná o blksize, tsize a timeout. Podpora pro rozšíření multicast nebyla implementována.

Bibliografie

[1] R. Braden. *Requirements for Internet Hosts – Application and Support*. RFC 1123. RFC Editor, říj. 1989. URL: https://datatracker.ietf.org/doc/html/rfc1123.

[2] G. Malkin a A. Harkin. *TFTP Blocksize Option*. RFC 2348. RFC Editor, květ. 1998. URL: https://datatracker.ietf.org/doc/html/rfc2348.

[3] G. Malkin a A. Harkin. *TFTP Option Extension*. RFC 1785. RFC Editor, břez. 1995. URL: https://datatracker.ietf.org/doc/html/rfc1785.

[4] G. Malkin a A. Harkin. *TFTP Option Extension*. RFC 2347. RFC Editor, květ. 1998. URL: https://datatracker.ietf.org/doc/html/rfc2347.

[5] G. Malkin a A. Harkin. *TFTP Timeout Interval and Transfer Size Options*. RFC 2349. RFC Editor, květ. 1998. URL: https://datatracker.ietf.org/doc/html/rfc2349.

[6] J. Postel. *TELNET PROTOCOL SPECIFICATION*. RFC 764. RFC Editor, čvn. 1980. URL: https://www.rfc-editor.org/rfc/rfc764.

[7] K. Sollins. *The TFTP Protocol*. IEN 133. Led. 1980. URL: https://www.rfc-editor.org/ien/ien133.txt.

[8] K. Sollins. *THE TFTP PROTOCOL (REVISION 2)*. RFC 1350. RFC Editor, čvc. 1992. URL: https://datatracker.ietf.org/doc/html/rfc1350.

[9] Wikipedia contributors. *Trivial File Transfer Protocol*. [Online; navštíveno 27-říjen-2021]. 2021. URL: https://en.wikipedia.org/wiki/Trivial_File_Transfer_Protocol.