Vysoké učení technické v Brně
Fakulta informačních technologií

Počítačové komunikace a sítě – 2. projekt
varianta EPSILON

Simple File Transfer Protocol

manuál a dokumentace

25. dubna 2021 Ondřej Ondryáš

Obsah

Úvod **2**

1 Překlad a spuštění **2**
1.1 Server 2
1.2 Klient 3

2 Práce se serverovou aplikací **3**

3 Práce s klientskou aplikací **3**

4 Implementace protokolu SFTP **3**
4.1 Připojení a přihlášení 4
4.2 Režimy přenosu 4
4.3 Výpis obsahu adresáře 5
4.4 Přejmenování souboru 5
4.5 Přenos souboru ze serveru na klienta 6
4.6 Přenos souboru z klienta na server 6

5 Implementační detaily **7**
5.1 Struktura kódu 7
5.2 Zjištění adres podle rozhraní 8
5.3 Asynchronní rozhraní socketů 8
5.4 Přenos po blocích 9

6 Testování **9**

Reference **10**

Úvod

Tato dokumentace popisuje použití a implementaci klientské a serverové aplikace pro přenos souborů podle protokolu *Simple FTP*, který je definován v RFC 913[1]. Serverová část podporuje obsluhu více uživatelů najednou.

Obě aplikace byly naprogramovány v jazyce C# nad platformou .NET Core ve verzi 3.1 a byly otestovány na operačních systémech Windows 10 a Ubuntu 20.04.

1 Překlad a spuštění

Na 64bitovém operačním systému Linux s nainstalovaným GNU Make je obě aplikace po rozbalení archivu možné přeložit příkazem `make`. Přiložený `Makefile` obsahuje cíle `client`, `server` a výchozí cíl `both`. Tyto cíle spouštějí *dotnet CLI* v režimu, který publikuje aplikace ve formě jednoho souboru, ale závislé na nainstalovaném prostředí .NET Core 3.1.

S použitím *dotnet CLI* je možné aplikace přeložit i v jiných režimech. Použití *dotnet CLI* je nutné také pro přeložení a spuštění projektů na operačním systému Windows. Aplikace lze spustit například příkazem `dotnet run` spuštěným v adresáři `IpkEpsilon.Client` nebo `IpkEpsilon.Server`. Více informací je možné nalézt v dokumentaci *dotnet CLI*[2].

Následující text uvažuje spustitelné soubory `ipk-simpleftp-server` a `ipk-simpleftp-client` na platformě Linux, tak jako je vygeneruje spuštění `make`.

1.1 Server

Před prvotním spuštěním severu je nutné vytvořit databázi uživatelů. Databáze uživatelů má podobu textového souboru v kódování ASCII[3]. Každý řádek obsahuje právě jeden znak `:`, který odděluje neprázdné uživatelské jméno a neprázdné heslo. Prázdné řádky jsou ignorovány, bílé znaky na začátcích a koncích řádků jsou odříznuty. Pokud soubor obsahuje neplatné záznamy, aplikace se ukončí s návratovým kódem 2.

Server je možné spustit následujícím příkazem:

```
./ipk-simpleftp-server -u cesta_k_databázi_uživatelů [-i rozhraní] [-p port]
```

Parametr `-u` je povinný a obsahuje cestu k souboru s databází uživatelů.
Parametr `-i` je nepovinný a nastavuje název rozhraní, na jehož adrese má server naslouchat (viz 5.2). Tento parametr je možné **použít opakovaně** a specifikovat tak více rozhraní. Pokud není kterékoliv rozhraní nalezeno nebo mezi nimi žádné nemá přiřazenou alespoň jednu IPv4 nebo IPv6 adresu, aplikace se ukončí s návratovým kódem 1. Pokud není tento parametr použit, server naslouchá na všech rozhraních (viz 5.2).
Parametr `-p` je nepovinný a nastavuje port, na kterém má server naslouchat. Pokud není předaná hodnota platné celé číslo v rozsahu 1–65535, aplikace se ukončí s návratovým kódem 1. Pokud není tento parametr použit, server naslouchá na portu 115[4].

Parametry mohou být předány v libovolném pořadí. Pokud je použit neočekávaný parametr, aplikace se ukončí s návratovým kódem 1.

[1] `https://tools.ietf.org/html/rfc913`

[2] `https://docs.microsoft.com/cs-cz/dotnet/core/tools/dotnet`

[3] Konec řádku může být označen sekvencí CRLF nebo LF.

[4] Na linuxových systémech musí být typicky aplikace naslouchající na portu nižším než 1024 spuštěna s právy superuživatele, případně mít oprávnění `CAP_NET_BIND_SERVICE` [1].

1.2 Klient

Klienta je možné spustit následujícím příkazem:

```
./ipk-simpleftp-client -h IP_nebo_jméno_serveru [-p port]
```

Parametr `-h` je povinný a obsahuje IP adresu serveru, ke kterému se klient připojuje. Pokud není hodnota parametru platnou IPv4 nebo IPv6 adresou, parametr se považuje za *hostname* a je proveden pokus o přeložení na adresu pomocí DNS. Pokud je neúspěšný, aplikace se ukončí s návratovým kódem 1.
Parametr `-p` je nepovinný a nastavuje port serveru, ke kterému se klient připojuje; platí stejná omezení jako u serveru.

Parametry mohou být předány v libovolném pořadí. Pokud je použit neočekávaný parametr, aplikace se ukončí s návratovým kódem 1.

2 Práce se serverovou aplikací

Na standardní výstup (`stdout`) jsou vypisovány informace o stavu serveru a o připojování a odpojování jednotlivých klientů. Každému klientovi je vygenerován unikátní identifikátor, díky kterému je možné rozlišit, ke kterému klientovi se vážou vypsané informace. Záznamy o varováních a chybových stavech jsou vypisovány na standardní chybový výstup (`stderr`).

Běžící server nepodporuje žádný přímý vstup od uživatele. Server je možné korektně ukončit zasláním signálu *SIGINT* (typicky stisknutím klávesové zkratky Ctrl+C v okně terminálu).

3 Práce s klientskou aplikací

Klientská aplikace se ihned po spuštění pokusí připojit k serveru (podle zadaných parametrů). Pokud se připojení zdaří, klient čeká na vstup od uživatele (signalizováno symbolem `>` na začátku řádku). Po stisknutí klávesy Enter je příkaz typicky v nezměněné podobě odeslán na server k vyhodnocení. Některé příkazy (`STOR`, `RETR`) však vyžadují další vstup od uživatele (zadání cesty k souboru), tyto mají tedy speciální chování popsané v dalších kapitolách. Zprávy přijaté od serveru v reakci na zaslaný příkaz jsou vypisovány na standardní výstup (typicky začínají znakem `+`, `-` nebo `!`). Kromě toho jsou na standardní výstup a standardní chybový výstup vypisována také různá stavová a chybová hlášení klientské aplikace, ta je možné od zpráv serveru rozpoznat tak, že začínají sekvencí `[INFO]`, `[WARN]` nebo `[ERR ]`.

Klienta je možné kdykoliv korektně ukončit zasláním signálu *SIGINT* (typicky stisknutím klávesové zkratky Ctrl+C v okně terminálu) nebo použitím příkazu `DONE`.

4 Implementace protokolu SFTP

Následující kapitola souhrnně popisuje proces použití této implementace protokolu SFTP. Protokol je navržen tak, že většinu řízení obstarává server. Klienta bychom mohli označit za „tenkého“, protože sám nevykonává komplexnější logiku a především zprostředkovává rozhraní pro zasílání příkazů na server a zobrazování odpovědí. Proto je dále popsáno zejména chování serveru v jednotlivých fázích komunikace, některé kapitoly však zároveň popisují speciální chování klientské aplikace ve stavech, kdy je nutné.

Protokol podle specifikace nerozlišuje velikosti písmen příkazů. V této implementaci není velikost písmen rozlišována ani u prvního parametru příkazů `TYPE`, `LIST` a `STOR`. Pokud ale server běží na operačním

systému se souborovým systémem, který rozlišuje velikosti písmen, je mezi nimi rozlišováno i v rámci protokolu SFTP. V následujícím textu budou příkazy protokolu uváděny velkými písmeny.

Příkaz, na který server odpoví kladně (odpověď začíná znakem `+` nebo `!`), je dále označován jako *úspěšná operace* nebo *úspěšně provedený příkaz*. Odpovědi začínající znakem `-` jsou označovány jako *neúspěšně provedené příkazy*.

Server pro každého připojeného klienta udržuje informaci o aktuálním adresáři. Po přihlášení je jako aktuální adresář nastaven pracovní adresář spuštěného serveru. Všechny příkazy, které jako argument požadují cestu k souboru nebo adresáři, podporují jak cesty relativní (vůči aktuálnímu pracovnímu adresáři klienta), tak absolutní. **Server tedy zpřístupňuje celý souborový systém stroje**, na kterém je spuštěn, a to v rozsahu oprávnění uživatele, pod kterým běží. Formát cesty závisí na operačním systému, na kterém server běží.

4.1 Připojení a přihlášení

Ihned po navázání spojení server zašle klientovi uvítací zprávu `+Hello, your client ID is X`, kde za `X` je dosazen náhodně vygenerovaný unikátní 26znakový identifikátor klienta[5]. Pokud klientovi nepřijde uvítací zpráva s `+` na začátku, klient je ukončen.

Server nyní očekává příkaz `USER` s uživatelským jménem. Pokud je zasláno platné uživatelské jméno (které existuje v databázi uživatelů), server očekává příkaz `PASS` s heslem. Po úspěšně provedeném příkazu `USER` je přijat také příkaz `ACCT`, ten je ale ignorován a jeho zaslání je prázdnou, avšak úspěšnou operací[6]. Pokud je zasláno správné heslo, server odpoví kladně; v opačném případě musí klient znovu zaslat příkaz `PASS` s heslem odpovídajícím předtím zvolenému uživatelskému jménu. Změna uživatelského jména není po provedení příkazu `USER` možná.

Po úspěšném přihlášení server přijímá příkazy `TYPE`, `LIST`, `CDIR`, `KILL`, `NAME`, `DONE`, `RETR` a `STOR`.

4.2 Režimy přenosu

Specifikace protokolu SFTP určuje tři režimy přenosu souboru, mezi kterými je možné přepínat příkazem `TYPE`. Výchozím režimem je režim `BINARY`, který přenáší data po osmibitových bajtech. Soubory jsou v tomto režimu přenášeny mezi serverem a klientem v nezměněné podobě. Tento režim je implementován velmi efektivně a jeho použití je doporučováno.

Alternativou je režim `ASCII`, který modifikuje odesílané soubory tak, aby odpovídaly přibližně formátu *NETASCII* [3]. To znamená, že jsou v odesílaném souboru nahrazena všechna zakončení řádku `LF` zakončením řádku `CRLF`. Pokud se v odesílaném souboru nachází bajt `CR`, za kterou nenásleduje bajt `LF`, je přenesen jako sekvence `CR` a nulového bajtu. Implementace tohoto režimu není příliš efektivní (viz 5.4) a jeho použití není doporučeno. Protože klient i server pracují výhradně s osmibitovými bajty, změní se při nastavení tohoto režimu pouze logika odesílání, a to přímo na klientovi i při obsluze tohoto klienta na serveru; logika příjmu souboru je stejná jako u binárního režimu. Použití příkazu `TYPE` tedy modifikuje stav klienta a jeho tvar je proto kontrolován ještě před odesláním na server.

[5] Jako identifikátor je vygenerován GUID, což je 128bitová hodnota s extrémně nízkou šancí vzniku duplicit [2]. Ten je zakódován do čitelné podoby pomocí Base32.

[6] Funkcionalitu *účtů* a příkazu `ACCT` autor považuje v kontextu dnešní doby za nadbytečnou a protože nebyla vyžadována zadáním (v porovnání s funkcionalitou uživatelských jmen a hesel, která byla zadáním jasně popsána), nebyla zařazena ani do implementace. Příkaz je přijímán, aby server odpovídal RFC 913 a aby byl kompatibilní s klienty, kteří tuto funkcionalitu podporují.

Režim `CONTINUOUS` je podle specifikace na cílových platformách této implementace ekvivalentní s režimem `BINARY`.

4.3 Výpis obsahu adresáře

Pro výpis informací o obsahu adresáře uživatel použije příkaz `LIST V` nebo `LIST F`. Příkaz může obsahovat také absolutní nebo relativní (vůči aktuálnímu adresáři) cestu k jinému adresáři, jehož obsah má být vypsán[7].

Varianta s argumentem `F` vypíše pouze seznam *souborů* v adresáři. První řádek odpovědi na úspěšně provedený příkaz obsahuje celou cestu k adresáři, jehož obsah je vypisován, a na každém dalším řádku je jméno jednoho souboru z adresáře.

Varianta s argumentem `V` vypíše celou cestu k adresáři a seznam souborů a podadresářů tohoto adresáře v následujícím formátu (jednotlivé položky jsou odděleny znakem tabulátoru).

```
Typ_položky Atributy Velikost Datum_vzniku Datum_posledního_zápisu Název
```

Například:

```
+/home/student/IpkEpsilon
D 00000010 0 2021-04-21T23:25:40 2021-04-21T23:25:40 IpkEpsilon.Server
D 00000010 0 2021-04-23T00:36:28 2021-04-23T00:36:28 IpkEpsilon.Common
D 00000010 0 2021-04-23T00:37:42 2021-04-23T00:37:42 IpkEpsilon.Client
F 00000080 544 2021-04-24T19:47:24 2021-04-24T19:47:24 Makefile
```

Typ položky je buď `D` (adresář), nebo `F` (soubor).
Atributy jsou hexadecimální hodnota označující bitový součet hodnot příznaků `FileAttributes`[8] daného souboru.
Velikost je celková velikost souboru v bajtech. U adresářů je tato položka vždy `0`.
Datum vzniku a posledního zápisu je ve formátu *úplného zobrazení pro kalendářní datum a čas dne* podle (ČSN) ISO 8601 [4, str. 27]: `yyyy-MM-dd'T'HH:mm:ss`.

Ačkoliv specifikace protokolu i pro rozšířený výpis (`LIST V`) ukládá vypisovat *soubory*, nespecifikuje formát tohoto výpisu; protože je tak tedy možné jasně rozlišit mezi souborem a adresářem, tato implementace vypisuje také adresáře, a to zejména z důvodu rozšíření uživatelského komfortu. Ve zkráceném výpisu (`LIST F`) jsou vypisovány pouze soubory, aby se implementace držela specifikace.

4.4 Přejmenování souboru

Soubor je možné přejmenovat příkazem `NAME název_souboru`. Tento příkaz není klientem nijak speciálně ošetřován a jeho použití odpovídá specifikaci protokolu: pokud je příkaz `NAME` úspěšný, server očekává zaslání nového názvu souboru příkazem `TOBE nový_název_souboru`. Pokud je zaslán jiný příkaz než `TOBE`, operace je přerušena a server dále **ne**očekává nový název souboru. Příkazem `NAME` je možné soubor také přesunout do jiného adresáře, a to použitím nové relativní nebo absolutní cesty v příkazu `TOBE`.

Adresáře není možné přejmenovat.

[7] Příkaz `LIST V` bude typicky ekvivalentní příkazu `LIST V .` (znak tečky typicky označuje aktuální adresář).
[8] `https://docs.microsoft.com/en-us/dotnet/api/system.io.fileattributes?view=netcore-3.1`

4.5 Přenos souboru ze serveru na klienta

Klient žádá o vzdálený soubor příkazem `RETR název_souboru`. Tento příkaz je klientem zpracován speciálním způsobem. Pokud vzdálený soubor existuje, server zašle odpověď s délkou souboru v bajtech[9]. Klient se následně uživatele zeptá na lokální cestu k uložení souboru.

```
> RETR large_file
419430400
Where should be the file stored? Press Enter to save to current directory.
>
```

Stisknutím klávesy Enter bez zadání názvu se soubor uloží do aktuálního adresáře, a to pod stejným jménem, jako má vzdálený soubor. Pokud soubor na zadané cestě už existuje, klientská aplikace si vyžádá potvrzení, jestli jej má přepsat.

```
Target file exists. Should I overwrite it? Continue? [Y/n]
```

Uživatel musí stisknout klávesu `Y` nebo `Enter` pro potvrzení přepsání, nebo `N` pro zrušení operace. (V případě stisku jiné klávesy je otázka zopakována.) Klient následně *automaticky* zašle serveru zprávu `SEND` nebo `STOP` a přenos je uskutečněn. Během přenosu je vypisován počet zatím přenesených bajtů.

Protokol neumožňuje ukončení přenosu v jeho průběhu. Jedinou možností, jak přenos ukončit, je ukončení celé klientské aplikace, což je možné provést i v průběhu přenosu zasláním signálu *SIGINT* (typicky stisknutím klávesové zkratky Ctrl+C v okně terminálu).

Server zamkne soubor pro čtení ihned po přijetí příkazu `RETR`. Díky tomu nemůže být soubor změněn během čekání na potvrzení nebo zrušení operace klientem. Pokud klient kdykoliv v průběhu operace vynuceně ukončí spojení, server přenos ukončí a uvolní prostředky (včetně zamknutého souboru)[10].

4.6 Přenos souboru z klienta na server

Klient spouští přenos souboru příkazem `STOR mód_přenosu název_vzdáleného_souboru`. Mód přenosu je buď `OLD`, který na serveru vytvoří nový soubor, nebo přepíše existující soubor daného jména; nebo `APP`, který na serveru vytvoří nový soubor, nebo připojí přijatý obsah na konec už existujícího souboru daného jména.

Protokol specifikuje také mód přenosu `NEW`, který je relevantní na platformách, které podporují ukládání několika různých verzí (*generací*) souboru téhož jména. To na aplikační úrovni cílové platformy typicky neumožňují, tato implementace serveru tedy příkaz v tomto módu podporuje, ale pokud soubor s požadovaným jménem už existuje, vždy bude zamítnut.

Tento příkaz je klientem ošetřován speciálním způsobem. Klient kontroluje správnost žádaného módu přenosu. Pokud je korektní, klient se zeptá na cestu zdrojového lokálního souboru:

```
> STOR OLD new_file_name
What file should be sent?
>
```

[9] Klient nekontroluje, jestli je na disku dostatečné množství místa pro uložení souboru.

[10] To představuje možnost provést útok typu DoS – klient může provést pro libovolný soubor požadavek na odeslání a nikdy jej nepotvrdit, soubor tak zůstane otevřený až do ukončení serveru nebo klienta a jiným klientům je znemožněno do souboru zapisovat. V budoucí verzi serveru by mohl být implementován časový limit pro potvrzení přenosu.

Příkaz `STOR` je na server odeslán *až poté*, co klient zkontroluje, že zadaný soubor existuje a je čitelný. Pokud server povolí uložení souboru pod daným jménem, klient *automaticky* zašle příkaz `SIZE`, který serveru sdělí počet přijímaných bajtů.

Server kontroluje možnost zápisu do daného souboru až po přijetí příkazu `SIZE`, a to proto, aby zbytečně nedržel otevřený soubor, který klient potenciálně nemusí začít přenášet. V tomto případě totiž protokol umožňuje serveru ukončit operaci až v reakci na příkaz `SIZE`. Server nekontroluje, jestli je na disku dostatečné množství místa pro uložení souboru.

5 Implementační detaily

5.1 Struktura kódu

V následujících podkapitolách je popsáno, jakým způsobem je členěn kód obou aplikací. Obě aplikace využívají společnou knihovnu `IpkEpsilon.Common`, která obsahuje pomocné metody, například pro logování nebo pro konverzi souborů při použití přenosového módu ASCII.

Server

Kód serveru je rozdělen na dva logické celky: síťový (ve jmenném prostoru `IpkEpsilon.Server.Network`) a protokolový (ve jmenném prostoru `IpkEpsilon.Server.Sftp`).

Vstupní bod programu, metoda `Main`, provede parsování parametrů příkazové řádky[11] a následně pro každou zjištěnou IP adresu (viz 5.2) vytvoří instanci `SocketServer`. Tyto objekty spravují sockety, které jsou použity pro příjem klientů (viz také 5.3).

Jakmile je klient přijat, vznikne v jiném vlákně pro komunikaci s ním další socket, který je obalen vytvořenou instancí třídy `ClientHandler`. V té je nastartována smyčka, ve které server čeká na zprávu od klienta, zpracuje ji a poté odešle odpověď. Právě problém zpracování odpovědi tvoří předěl mezi síťovou a protokolovou částí.

Před začátkem zmíněné smyčky je vytvořen objekt implementující rozhraní `ISftpProvider`. Po přijetí celé zprávy je zavolána metoda `HandleCommand` tohoto objektu, která příkaz zpracuje a vrátí objekt typu `CommandExecutionResult`, který popisuje, jakou akci má `ClientHandler` vykonat (odeslat textovou zprávu nebo odeslat soubor – proud dat) a v jakém stavu má být po vykonání této akce (příjem dalšího příkazu, příjem souboru nebo ukončení spojení). Řízení protokolu je tak úplně odstíněno od problému síťové komunikace, což zvyšuje čitelnost kódu a hypoteticky umožňuje použít stejný kód pro řízení protokolu s jinou implementací komunikace s klientem.

K dosažení rozumného oddělení těchto dvou celků napomáhá také využití abstrakcí[12]. `ClientHandler` nevytváří objekty poskytující práci s protokolem sám, ale za pomoci továrního třídy implementující rozhraní `ISftpProviderFactory`; ta navíc vytváří objekty implementující rozhraní `ISftpProvider` – síťová část je tak zcela odstíněna od konkrétního typu poskytovatele protokolu a bylo by tak naopak možné ji použít v kombinaci s jiným protokolem podobného návrhu.

V rámci samotné protokolové části je tento návrh použit pro odstínění `SftpProvider` objektu od poskytovatele databáze uživatelských účtů. Implementace `FileAccountProvider` rozhraní `IAccountProvider`

[11] To je provedeno „ručně“, ale pokud by se počet či variabilita parametrů zvyšovaly, bylo by vhodné použít knihovnu.

[12] Zde je patrná inspirace návrhovými vzory, které využívá například ASP.NET – *Inversion of Control* a *Dependency Injection* [5]. Použití skutečného kontejneru závislostí by pro projekt tohoto rozsahu bylo zbytečné, ale všechny konkrétní typy, jejichž instance vytváří síťová část, jsou vkládány v metodě `Main`.

čte uživatele ze souboru, podle zadání. Bylo by ale možné vytvořit další implementaci, která by například četla uživatele z databáze.

Klient

Implementace klientské aplikace je podstatně jednodušší než serverové části, neboť klienta tvoří v podstatě jen smyčka odesílání příkazů a příjem a zpracovávání odpovědí. Ačkoliv by bylo možné zde využít synchronní rozhraní socketů, zejména kvůli možnosti ošetřeného přerušení činnosti v jakékoliv části programu je i zde využit asynchronní přístup. I zde je mezi dvě třídy rozdělena logika síťové komunikace a logika protokolu, avšak na rozdíl od serveru jsou tyto třídy svou implementací silně provázané.

Vstupní bod programu, metoda `Main`, provede parsování parametrů příkazové řádky a vytvoří instance tříd `SocketClient` a `SftpClient`. Poté je spuštěna hlavní smyčka, která je implementována v metodě `SftpClient.Run`. Zde je provedeno připojení i obsluha chyb. Řízení komunikace má tedy na starost třída řídící protokol `SftpClient`, metody `SocketClient` slouží spíše pro usnadnění často prováděných operací, jako je odeslání a příjem zprávy.

5.2 Zjištění adres podle rozhraní

Zadání vyžaduje při spouštění serveru možnost volby rozhraní, na kterém má naslouchat. Oproti zadání poskytuje tato implementace možnost specifikovat těchto rozhraní více. Server využívá pro síťovou komunikaci rozhraní BSD socketů. Socketům, které mají přijímat spojení, ale musí být přiřazena IP adresa, nikoliv rozhraní [6, 7]. Proto je nutné podle zadaných názvů rozhraní vyhledat jejich IP adresy a pro každou nalezenou adresu poté vytvořit samostatný socket přijímající klienty (v našem případě instanci třídy `SocketClient`). Musíme také uvažovat, že rozhraní může mít více unicastových IP adres – v takovém případě je vhodné přijímat spojení na všech z nich (alternativou by bylo poskytnout uživateli možnost zvolit pomocí parametrů přímo IP adresy, na kterých se má poslouchat, ne rozhraní).

Tento proces je implementován zejména v metodě `Program.GetInterfaceAddresses`. Ta první najde zadané rozhraní – předaný identifikátor musí být roven hodnotě vlastnosti `Id` nebo `Name`[13,14] jednoho ze síťových rozhraní v systému (porovnává se bez ohledu na velikost písmen). Pokud toto rozhraní není nalezeno, aplikace je ihned ukončena. Následně jsou vybrány všechny unicastové adresy typu IPv4 a IPv6, které jsou navráceny. Výsledky volání této metody pro všechna zadaná rozhraní jsou spojeny a pro každou z adres je vytvořen objekt, který vytváří socket a spravuje přijímání klientů. Pokud jsou zadaná nějaká rozhraní, ale na žádném z nich není nalezena alespoň jedna vhodná IP adresa, aplikace je ukončena.

5.3 Asynchronní rozhraní socketů

Na první pohled zvláštním se může jevit kombinování staršího rozhraní pro provádění asynchronních operací (dvojice `BeginAccept` a `EndAccept`) a novějšího async/await modelu ve třídě `SocketServer`. Ve vyžadované verzi prostředí .NET (Core 3.1) bohužel ještě neexistovala async/await metoda pro přijímání spojení `BeginAsync`, které by bylo možné zaslat přerušení pomocí `CancellationToken`[15]. Tento model je ale využíván pro řízené ukončení serveru při obsluze signálu *SIGINT*. Z tohoto důvodu je použit starší asynchronní model, který tato přerušení podporuje, a po přijetí klienta, které proběhne v jiném

[13] `https://docs.microsoft.com/en-us/dotnet/api/system.net.networkinformation.networkinterface`

[14] Na Linuxu mívají obvykle stejnou hodnotu, na Windows je `Id` systémovým identifikátorem, zatímco `Name` obsahuje čitelné jméno rozhraní.

[15] Implementace async/await rozhraní pro sockety v .NET je vcelku rozsáhlým projektem a je možné její průběh sledovat na Githubu: `https://github.com/dotnet/runtime/issues/33418`.

vlákně, je teprve spuštěna asynchronní operace využívající async/await model (ostatní použité metody pro komunikaci přes sockety totiž vyžadované přerušování podporují).

5.4 Přenos po blocích

Při přenosu souboru v binárním režimu je vždy využíváno blokového režimu: Ze souboru je načten blok o velikosti 1 400 B[16] a tento blok je předán socketu k přenesení (nebo naopak – ze socketu je přečteno až 1 400 B a přečtené bajty jsou zapsány do proudu představujícího otevřený soubor). Aplikace tak v žádném kroku nemá alokováno příliš mnoho paměti.

ASCII režim bohužel takto jednoduše a efektivně data mezi socketem a proudem přenášet neumožňuje, protože je nutné vstupní soubor analyzovat a nahrazovat v něm některé sekvence bajtů jinými. Zde byla zvolena z důvodu jednoduchosti ne příliš efektivní implementace, která vede na načítání celého vstupního souboru do paměti. Soubor je totiž čten bajt po bajtu a za pomoci konečně stavového řízení jsou případně pozměněné sekvence bajtů ukládány do paměti. (Toto je implementováno v pomocné metodě `IpkEpsilon.Common.AsciiStreamUtils.LoadAsciiFile`.) Ta je potom síťové vrstvě předána jako proud, odesílání tedy poté probíhá stejným způsobem jako při přímém čtení ze souboru. Bylo by možné také použít blokový přístup, vyžadovalo by to ale hlubší zásahy do kódu a porušení separace síťové a protokolové vrstvy.

6 Testování

Vývoj aplikací probíhal iterativním způsobem – postupně byla doplňována funkcionalita jednotlivých částí protokolu. Testování probíhalo ručně, zkoumáním chování obou částí při provádění jednotlivých příkazů. Pro testování byl použit vývojový stroj s operačním systémem Windows 10 a virtuální stroj spuštěný v prostředí VMware s poskytnutým referenčním obrazem Ubuntu 20.04.

Síťový adaptér virtuálního počítače byl nakonfigurován v režimu „Bridged“ (přemostění). V tomto režimu se virtuální počítač chová, jako by byl zapojený přímo do sítě hostitelského počítače; *sdílí* s ním síťový adaptér, ale v rámci sítě má vlastní identitu[17].

Bylo otestováno, že se server korektně spouští na obou systémech a může se k němu připojit více klientů z obou systémů. Důraz byl kladen na otestování zároveň spuštěných přenosů souborů, kdy více než jeden klient přenášel soubory ze serveru a více než jeden klient přenášel soubory na server. Také bylo otestováno, že server správně zamyká a odemyká soubory – jakmile jeden z klientů začne soubor číst (přenášet příkazem `RETR`), může jej začít číst i jiný klient, ale nemůže do něj zapisovat ani žádný jiný klient, ani žádná třetí aplikace běžící na serveru. Po ukončení čtení všemi klienty je soubor odemknut a je do něj opět možné zapisovat.

Protože byl server implementován dříve než klient, k testování byl použit také nástroj *netcat*, který umožňuje komunikovat pomocí TCP spojení. Použitá implementace pro systém Windows[18] bohužel neumožňuje jednoduše zaslat na server nulový bajt, který protokol používá jako ukončovač zprávy. Řešením bylo dočasně provést triviální změnu kódu serveru – upravit hodnotu, kterou server hledá v přijatých datech jako ukončovač, na hodnotu `LF` (0x0A). Na referenčním linuxovém stroji je možné

[16] Tato hodnota je určena konstantou v kódu a byla takto zvolena kvůli typickému MTU Ethernetu – 1 500 B [8, str. 274]. Nebylo by ale obtížné udělat ji uživatelsky nastavitelnou.

[17] Hostitelský software do hostitelského síťového adaptéru přidává ovladač, který dovoluje pomocí adaptéru zasílat pakety s jinou MAC adresou a zasílat do virtuálního stroje zpět přijaté pakety s touto adresou.

[18] `https://eternallybored.org/misc/netcat/`

v terminálu zaslat nulový bajt klávesovou zkratkou Ctrl+Shift+2 a odeslat zprávu zasláním EOF (Ctrl+D), zde tedy použití k testování nic nebránilo ani se standardní implementací serveru.

Díky použitému návrhu kódu serverové aplikace by bylo možné část jeho funkcionality spolehlivě testovat pomocí unit testů, to však nebylo z časových důvodů provedeno.

Reference

1. KERRISK, Michael. *capabilities - overview of Linux capabilities* [online]. 2021 [cit. 2021-04-25]. Dostupné z: `https://man7.org/linux/man-pages/man7/capabilities.7.html`.
2. MICROSOFT CORPORATION. *Guid Struct* [online]. 2021 [cit. 2021-04-25]. Dostupné z: `https://docs.microsoft.com/en-gb/dotnet/api/system.guid?view=netcore-3.1`.
3. POSTEL, J. *Telnet Protocol specification* [Internet Requests for Comments]. RFC Editor, 1980-06. RFC, 764. Dostupné také z: `https://rfc-editor.org/rfc/rfc764.txt`.
4. ÚNMZ. *Datum a čas – Zobrazení pro výměnu informací – Část 1: Základní pravidla*. Praha, ČR, 2020-03. Česká technická norma, ČSN ISO 8601-1. Úřad pro technickou normalizaci, metrologii a státní zkušebnictví.
5. MICROSOFT CORPORATION. *Architectural principles* [online]. 2021 [cit. 2021-04-25]. Dostupné z: `https://docs.microsoft.com/en-gb/dotnet/architecture/modern-web-apps-azure/architectural-principles`.
6. FAITH, Rickard E.; KERRISK, Michael. *bind - bind a name to a socket* [online]. 2021 [cit. 2021-04-25]. Dostupné z: `https://man7.org/linux/man-pages/man2/bind.2.html`.
7. MICROSOFT CORPORATION. *Socket.Bind(EndPoint) Method* [online]. 2021 [cit. 2021-04-25]. Dostupné z: `https://docs.microsoft.com/en-gb/dotnet/api/system.net.sockets.socket.bind?view=netcore-3.1`.
8. KUROSE, James F.; ROSS, Keith W. *Computer networking: a top-down approach*. 7. vyd. Hoboken, New Jersey: Pearson, 2017. isbn 978-0-13-359414-0.