VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

PROGRAMOVÁNÍ SÍŤOVÝCH APLIKACÍ NA RŮZNÝCH PLATFORMÁCH

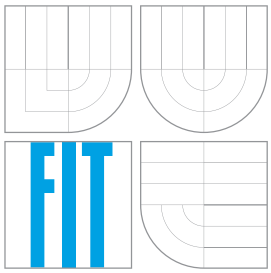
BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

MARTIN VACH

BRNO 2012

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

PROGRAMOVÁNÍ SÍŤOVÝCH APLIKACÍ NA RŮZNÝCH PLATFORMÁCH

PROGRAMMING OF NETWORKING APPLICATIONS ON DIFFERENT PLATFORMS

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

MARTIN VACH

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. PETR MATOUŠEK, Ph.D.

BRNO 2012

Abstrakt

Cílem práce je porovnat síťové komunikační protokoly a jejich implementace na různých platformách. V práci jsou popsány používané komunikační protokoly a možnosti jejich implementace v různých programovacích jazycích. Jsou zde popsány postupy implementace síťové komunikace na demonstrační aplikaci a testování těchto implementací. Na základě výsledků testování jsou komunikační protokoly a jejich implementace porovnány.

Abstract

The aim of this thesis is compare the network communication protocols and their implementations on different platforms. In the work are described the used communication protocols and the possibilities of their implementation in different programming languages. There are described procedures for implementation of network communication on the demonstration application and testing of these implementations. Based on the results of testing, are the communication protocols and their implementations compared.

Klíčová slova

síťová komunikace, konkurentní server, iterativní server, klient, schránka.

Keywords

network communication, concurrent server, iterative server, client, socket.

Citace

Martin Vach: Programování síťových aplikací na různých platformách, bakalářská práce, Brno, FIT VUT v Brně, 2012

Programování síťových aplikací na různých platformách

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Petra Matouška, Ph.D.

.....
Martin Vach
14. května 2012

Poděkování

Děkuji vedoucímu bakalářské práce Ing. Petru Matouškovi, Ph.D. za odborné vedení a za rady poskytnuté při zpracování bakalářské práce.

Obsah

Kapitola 1

Úvod

Tato práce se zabývá síťovou komunikací typu klient/server (iterativní i konkurentní) pro různé protokoly a na různých platformách. Konkrétně se jedná o programování síťových aplikací v prostředí Unixu (jazyk C/C++, Perl, Python) a Windows (C# v .NET, Java). Implementované jsou UDP, TCP, SCTP, IP a linkové protokoly.

1.1 Cíle práce

Cílem práce je porovnání prostředků, knihoven a postupů pro implementaci síťové komunikace typu klient/server. Tohoto porovnání bude dosaženo vytvořením souboru aplikací typu klient/server, které budou tuto komunikaci implementovat. Bude se jednat o aplikace, které budou reprezentovat klienty a servery. Servery budou konkurentní a iterativní, zvlášť pro protokoly TCP, UDP, SCTP, IP a protokoly linkové vrstvy. Na základě implementace těchto aplikací budou porovnány prostředky pro jejich implementaci.

1.2 Síťová komunikace

Počítačová síť se skládá z mnoha síťových zařízení. Jedná se především o koncová zařízení (osobní počítače, servery, a prakticky vše co je připojeno do sítě a je využíváno koncovými uživateli), přenosové médium (měděné kabely, optické kabely, bezdrátový přenos, ...) a směrovače (routery), které zajišťují směrování paketů od zdrojového k cílovému zařízení.

Aby bylo možné doručit paket od zdroje k cíly, je v počítačových sítích využita adresace. K určení cíle je využito IP adresy a portu. IP adresa slouží k adresování konkrétního síťového rozhraní v síti a port slouží k rozpoznání služby (aplikace) na cílové adrese (rozhraní), které je paket adresován. V současnosti se využívají dvě verze IP adres. Jedná se o starší IPv4 a novější IPv6. IPv4 adresa je 32 bitové číslo zapisované po bajtech oddělených tečkami. IPv6 adresa má 128 bitů. Zapisuje se do osmi skupin po čtyřech hexadecimálních číslech oddělených dvojtečkami. Porty sloužící k určení služby (aplikace, které je paket určen) mají rozsah od 0 až po 65535 a jsou rozděleny do skupin. První jsou tzv. well known ports (0-1023). Tyto porty jsou vyhrazené pro běžné služby. Druhou skupinou jsou registrované porty (1024-49151), jejichž použití by mělo být registrováno u ICANN. Poslední skupinou jsou dynamické a soukromé (49152-65535). Tyto porty jsou určeny pro soukromé využití a nejsou jim přiděleny žádné apllikace.

1.3 Klient/server

Klient/server je síťová architektura, skládající se ze dvou částí, které mezi sebou komunikují. Jsou jimy klienti a servery. Jedná se o dvě odlišné entity. Klient v počítačové komunikaci většinou požaduje nějakou službu a server tuto službu poskytuje. Inicializaci komunikace většinou provádí klient tím že pošle serveru, který je identifikován IP adresou, požadavek na nějakou jeho službu (identifikována číslem portu). Server přijme tento požadavek na daném portu a pokud může, vyhoví mu a odešle klientovi žádanou odpověď. Zpracování obvykle probíhá na straně serveru, zatímco klient posílá požadavky a přijímá výsledky (odpovědi).

Servery se dělí podle způsobu obsluhy klienta na iterativní a konkurentní. Toto dělení určuje počet klientů, které může server najednou obsloužit. Iterativní server může v jeden okamžik obsluhovat právě jednoho klienta (jeho požadavek). Oproti tomu konkurentní server může v jeden okamžik paralelně obsluhovat více klientů. Toto chování bývá implementováno jako obsluha jiným procesem nebo vláknem, které nezávisle na ostatní komunikaci vyřizuje požadavek jednoho klienta.

1.4 Prostředky komunikace

V síťové komunikaci se využívají schránky (socket). Schránky tvoří aplikační programové rozhraní (API) pro meziprocesovou komunikaci. Při komunikaci prostřednictvím schránek nemusí běžet oba procesy na stejném počítači. Procesy mohou běžet na různých počítačích, které jsou propojeny do počítačové sítě. Schránka je v této komunikaci pojmenování pro koncový bod komunikace. Představují abstraktní datovou strukturu obsahující údaje potřebné pro komunikaci. Schránky jsou v síti jednoznačně identifikovány IP adresou (rozhraní připojené do sítě) a portem (identifikujícím aplikaci ve které pracují). Krom schránek lze pro komunikaci využít i prostředky pro zachytávání paketů na síťovém rozhraní a pro jejich odesílání.

1.5 Postup práce

Postup práce lze rozdělit do několika etap. Tou první je nastudování dané problematiky (síťová komunikace a protokoly). Na základě těchto znalostí bude vytvořen konkrétní model komunikace pro každý protokol (testovací aplikace). Model bude nejprve implementován v jazyce C jako vzorová aplikace, na jejímž základě budou nalezeny vhodné prostředky v ostatních programovacích jazycích a tato komunikace v nich bude implementována. Výsledkem implementace bude soubor aplikací (klientů a serverů) implementujících komunikaci v daných jazycích a na daných protokolech. Tyto aplikace budou poté otestovány a na základě tohoto testování budou porovnány prostředky pro jejich implementaci a postupy implementace v jednotlivých jazycích.

Pro demonstraci implementace komunikace v daných protokolech jsem zvolil binární přenos souborů. Tento typ komunikace se skládá z:

1. Inicializace komunikace – vytvoření prostředku komunikace (např. schránky), navázání spojení s druhou stranou (pokud to protokol vyžaduje) a zahájení přenosu.
2. Přenos dat – data jsou vysílána a přijímána v cyklu po částech.
3. Ukončení komunikace – identifikace konce přenosu dat, přerušení komunikačního kanálu, pokud byl vytvořen a ukončení aplikace (na straně klienta)

Kapitola 2

Protokoly

Komunikace v počítačové síti musí mít jednotnou formu, aby obě komunikující strany rozuměly té druhé. K tomuto účelu slouží komunikační protokoly. Protokol definuje syntaktická a sémantická pravidla způsobu komunikace mezi dvěma místy v síti (mezi dvěma koncovými zařízeními). Protokoly definují způsob adresování, navázání komunikace obou koncových bodů, formát zpráv (které si obě strany posílají), zpracování chyb, které v průběhu komunikace mohou nastat (např. ztráta paketu) a způsob ukončení komunikace.

V počítačových sítích existují různé úrovně (vrstvy) komunikace. Těmito vrstvami se zabývá několik modelů síťové komunikace. Jedná se např. o model ISO/OSI [9]. Model ISO/OSI je referenčním modelem, definovaným Mezinárodní standardizační organizací ISO (International Standard Organization). Tento model je základem ostatních modelů a skládá se ze sedmi vrstev.

Vrstvy modelu ISO/OSI (viz Obrázek 2.1):

1. Fyzická vrstva – zabývá se přímo přenosem dat ve formě bitů, pracuje na úrovně hardware.
2. Linková vrstva – stará se o přenos dat mezi sousedními systémy, využívá rámce.
3. Síťová vrstva – zajišťuje směrování komunikace v sítích, využívá IP datagramy.
4. Transportní vrstva – poskytuje přenos dat (segmentů) mezi koncovými uzly.
5. Relační vrstva – řídící mechanismy pro komunikaci mezi aplikacemi.
6. Prezenční vrstva – transformace dat do vhodného formátu pro aplikace.
7. Aplikační vrstva – aplikace komunikující v síti.

Každá z vrstev OSI modelu využívá data z vyšší vrstvy a ta zapouzdří vlastní informací. První tři vrstvy pracují na úrovni dat. V transportní vrstvě jsou data rozdělena do segmentů (paketů). Síťová vrstva tyto segmenty zapouzdří do IP datagramů (přidání IP adresy). Linková vrstva přidá fyzickou adresu a takto vzniklé rámce předá na fyzickou vrstvu, ze které jsou data přenášena ve formě bitů. Na příjímací straně je tento postup opačný. Ne všechna zařízení v síti potřebují číst pro svoji potřebu data. To se týká např. směrovačů, které ke své činnosti (směrování) potřebují znát IP adresu, proto pracují jen v rámci nejnižších třech vrstev.

Obrázek 2.1: Modely síťové komunikace

Model ISO/OSI je modelem referenčním. Pro účely implementace jsou vhodnější jiné modely. Takovýmto modelem je model TCP/IP [4], který ISO/OSI model zjednodušuje. Toto zjednudušení spočívá ve spojení některých vrstev do jedné. První tři vrstvy modelu ISO/OSI (aplikační, prezenční a relační) jsou spojeny do jedné (aplikační vrstvy). Pod ní se nacházejí transportní a internetová vrstva. Linková a fyzická vrstva ISO/OSI modelu jsou spojeny do vrstvy síťového rozhraní.

Vrstvy modelu TCP/IP (viz Obrázek 2.1):

1. Vrstva síťového rozhraní (linková) – poskytuje přístup k přenosovému médiu.
2. Síťová vrstva – zajišťuje směrování a adresaci dat (IP datagramu) v síti.
3. Transportní vrstva – poskytuje přenos dat mezi koncovými body komunikace.
4. Aplikační vrstva – spojuje funkčnost nejvyšších tří vrstev ISO/OSI modelu. Zabývá se aplikační částí komunikace (aplikace využívající komunikace po síti).

Schránky pracují na rozhraní mezi aplikační a transportní vrstvou. Je to z toho důvodu, že se zde oddělují aplikační informace od komunikačních informací. Tedy data používaná aplikacemi a informace pro přenos těchto dat po síti.

Tato práce se zabývá komunikací na více vrstvách síťového modelu:

- Transportní – protokoly TCP, UDP, SCTP.
- Síťové – protokol IP.
- Linkové – protokoly linkové vrstvy.

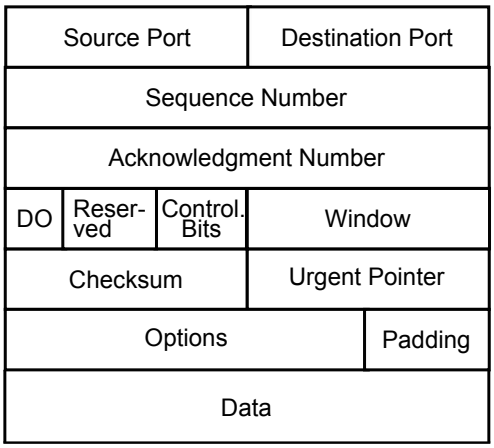
2.1 Protokol TCP

TCP (Transmission Control Protocol) [13] je protokolem transportní vrstvy ISO/OSI modelu sloužící k přenosu toku bajtů. Tento protokol je spojově orientovaný (connection-oriented). To znamená, že se jedná o spojově orientovanou službu. U tohoto typu komunikace dojde nejprve k vytvoření spojení mezi oběma komunikujícími stranami a až poté může

začít samotná komunikace (přenos informací). TCP poskytuje plně duplexní (full duplex) přenos, což znamená, obousměrný přenos dat. TCP zároveň poskytuje spolehlivé doručování. Po odeslání dat je od strany komunikace, která přijímá data požadováno potvrzení přijetí. Pokud odpověď není doručena v určitém čase, jsou data odeslána znovu. Pokud se data nepodaří doručit, TCP ukončí přenos jako neúspěšný. TCP k odhadu této doby využívá RTT (round-trip time), což je doba pro přenos dat k cíly a navrácení odpovědi. Výpočet této doby je dynamický, aby mohl reagovat na změny v síti.

Při TCP přenosu se zároveň využívá sekvenčních čísel. Pomocí sekvenčních čísel předchází protokol TCP doručení paketů v nesprávném pořadí, duplikaci nebo ztrátě dat. Při přenosu většího množství dat jsou tato data rozdělena do segmentů (podle maximální velikosti MTU) a jsou poslána zvlášť. MTU (maximum transmission unit) je hodnota daná hardwarem (linkovou vrstvou). MTU ethernetu je 1500 bajtů, u jiných technologií je tato hodnota odlišná. Příjemce poté na tyto přijaté pakety odpovídá prostřednictvím sekvenčních čísel. Tímto lze zjistit, jestli dorazila všechna data nebo se některé pakety ztratily. Zároveň je možnost, že data jsou přijata v nesprávném pořadí. V tomto případě může TCP na druhé straně tato data přeuspořádat. Sekvenční čísla také řeší příjem duplicit (např. vysílací straně není doručeno potvrzení a data jsou znovu poslána). Tyto duplicity jsou na straně příjemce zahozeny.

TCP také využívá kontrolu toku (flow control). TCP se dotazuje, kolik dat je možné přijmout v konkrétním čase. Tomu přizpůsobí velikost okénka (jedná se o množství dat, které může přijmout přijímací vyrovnávací paměť). Tato hodnota je doručena ve zprávě ACK (potvrzení doručení). To zajišťuje, že odesílací strana nemůže poslat nadbytek dat, při kterých by vyrovnávací paměť pro příjem na cílové straně komunikace přetekla.



Obrázek 2.2: TCP hlavička

2.1.1 Komunikace TCP

Jelikož je TCP spojově orientovaná služba, musí dojít k navázání komunikačního kanálu mezi serverem a klientem před samotným procesem komunikace.

Postup navázání komunikace (viz obr. 2.3):

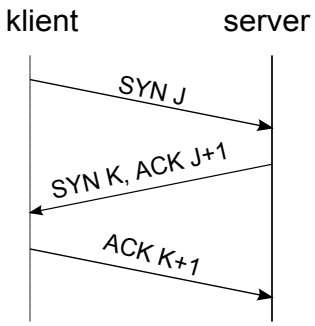
1. Server musí být připraven akceptovat příchozí spojení (od klienta).

2. Pokud se chce klient k serveru připojit, pošle serveru synchronizační zprávu (SYN). Tímto klient inciuje komunikaci se serverem (posílá mu počáteční sekvenční číslo – J).

3. Server musí klientovi na tuto zprávu odpovědět (ACK). Server posílá zrávu obsahující jeho SYN (součástí je počáteční sekvenční číslo serveru K) a zároveň odpověď (ACK) na zprávu klienta.

4. Na konec musí klient odpovědět na zprávu SYN, která mu přijde od serveru.

Tento proces se nazývá třícestná synchronizace (Three-way handshake) – dojde k přenosu tří zpráv.



Obrázek 2.3: Navázání TCP komunikace [17]

Součástí každé zprávy SYN mohou být volby TCP. Těmi jsou maximální velikost segmentu (množství dat akceptovatelných v každém segmentu TCP). Je to maximální velikost segmentu, kterou může daná strana komunikace odeslat. Maximální velikost okénka (velikost vyrovnávací paměti k příjmu), časové razítko, které zamezuje poškození dat starými, zpožděnými nebo duplicitními segmenty.

Ukončení komunikace (viz obr. 2.4):

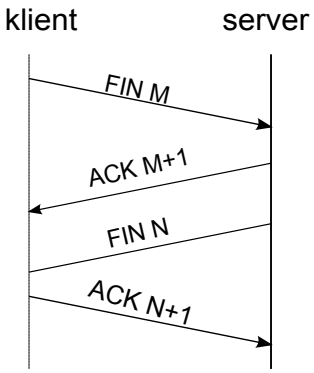
1. Pokud jedna z aplikací na jedné straně spojení ukončí komunikaci (aktivní uzavření), pošle zprávu FIN (chce ukončit posílání dat) se sekvenčním číslem (M).

2. Druhá strana přijme zprávu FIN (pasivní ukončení). Na přijatý FIN odpoví a informaci o přijetí FIN také pošle aplikaci jako informaci o ukončení přenosu (aplikace končí s příjmem s vyjímkou toho, co už má ve frontě).

3. Po dokončení příjmu aplikace uzavírá schránku i na druhé straně a je odeslán FIN (se sekvenčním číslem N) ke straně, která zahájila ukončení.

4. Druhá strana (aktivně uzavřená) přijme FIN a odpoví na něj.

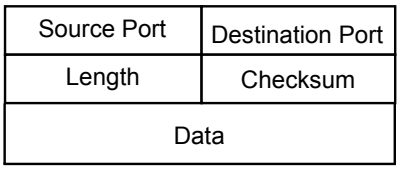
V průběhu ukončení může dojít k toku dat od pasivně ukončené strany komunikace k aktivně ukončené straně. Tomu se říká poloviční ukončení (half-close), nacházející se mezi kroky 2 a 3. Zasláním zprávy FIN dojde k uzavření schránky.



Obrázek 2.4: Ukončení TCP komunikace [17]

2.2 Protokol UDP

UDP (User Datagram Protocol) [10] je stejně jako TCP protokolem transportní vrstvy. UDP není spojově orientovaná služba, jedná se o transakčně orientovanou službu. UDP nevyužívá na rozdíl od TCP proud dat, ale UDP datagramy. Protokol negarantuje doručení UDP datagramu k cíly. Každý datagram má svoji velikost a tato velikost je spolu s datagramem doručena cílové aplikaci. Protokol je nespolehlivý, nedetekuje ztrátu paketů a proto není schopen ztracený paket znovu poslat. Pokud chceme využívat spolehlivosti v UDP přenosu, je nutné ji do aplikace doimplementovat. Jedná se zde především o mechanismy potvrzování doručení, přerušení po vypršení časovače (timeout) a znovu posílání datagramů, které nebyly doručeny. Jelikož není UDP spojově orientovaná služba, nedochází mezi komunikujícími stranami k navázání spojení. Aplikace si tedy nevytvoří komunikační kanál, po kterém by celá komunikace probíhala, ale zasílají si jen adresované datagramy. Z tohoto důvodu nemůže server jednoduše přijímat data od všech klientů na jednu UDP schránku.



Obrázek 2.5: UDP hlavička

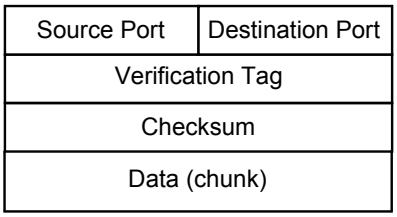
2.2.1 Komunikace UDP

Jak už bylo popsáno výše, nedochází k navázání spojení mezi oběma stranami komunikace. Komunikace pomocí UDP protokolu probíhá formou zasílání datagramů, které jsou adresované IP adresou cílového rozhraní a portem cílové služby, a zpět jako odpověď je odeslán datagram, jehož cílovou adresou je zdrojová adresa obsažená v přijeté zprávě (totéž platí i pro port).

2.3 Protokol SCTP

SCTP (Stream Control Transmission Protocol) [18] je dalším z protokolů transportní vrstvy. Tento protokol původně vznikl pro telefonní účely. Protokol narozdíl od protokolu TCP provádí tzv. asociaci mezi klienty a servery místo spojení. Tento pojem je zaveden z důvodu odlišení způsobu komunikace. Zatímco spojení se týká dvou koncových IP adres, asociace představuje komunikaci dvou systémů, které mohou zahrnovat více IP adres díky multihomingu. Multihoming umožňuje jednomu konci SCTP komunikace mít více IP adres. Tento přístup umožňuje větší odolnost proti poruchám v síti. To je způsobeno tím, že koncový bod komunikace je připojen do sítě více redundantními spojeními (proudy). Počet těchto proudů je určen během inicializace asociace dohodou obou komunikujících stran. Data jsou v těchto proudech přenášena paralelně a nezávisle na ostatních proudech. V každém z těchto proudů zaručuje SCTP doručení všech dat ve správném pořadí. V případě poruchy jednoho spojení může SCTP přepnout na jiné a pokračovat. Asociace umožňuje přenos přes všechny možné kombinace těchto seznamů zdrojových/cílových adres koncových bodů. Přenášená data jsou rozdělena na segmenty (chunks), které jsou identifikovány hlavičkou a jako SCTP pakety jsou odeslány paralelně nezávislými toky k cíly. SCTP poskytuje podobné služby jako TCP. Jsou to spolehlivost, sekvenční čísla, kontrola toku a plně duplexní přenos dat. Ovšem SCTP je na rozdíl od TCP, které je spojově orientovanou službou, službou orientovanou na zprávy (message-oriented). Komunikace může probíhat po několika na sobě nezávislých kanálech přenášejících data paralelně. Každý z těchto kanálů má vlastní spolehlivé doručování zpráv. Díky tomu ztracená zpráva v jednom toku neblokuje ostatní. Tímto se SCTP odlišuje od TCP, kde pokud dojde ke ztrátě, je přenos blokován, dokud tato situace není vyřešena.

Pro zajištění spolehlivosti využívá SCTP dva typy sekvenčních čísel. Jsou to sekvenční čísla proudů (Stream Sequence Number), která určují číslování v rámci jednoho proudu, přenosové sekvenční číslo (Transmission Sequence Number), což je číslo každé datové zprávy na úrovni asociace. Tato dvě sekvenční čísla jsou na sobě nezávislá. Tímto je zajištěna funkčnost přenosu i při zablokování komunikace v jednom nebo více proudech. Přijímací strana odpoví na přenosové sekvenční číslo, i když je v sekvenci těchto čísel mezera způsobená zablokováním některého z dalších proudů.



Obrázek 2.6: SCTP hlavička

2.3.1 Komunikace SCTP

Stejně jako v případě protokolu TCP předchází i u SCTP přenosu dat inicializace spojení mezi koncovými body komunikace.

Tento inicializační proces se nazývá asociace a skládá se z následujících kroků (viz obr. 2.7):

1. Server je připraven akceptovat příchozí asociaci. Tomuto stavu se také říká pasivní otevření.

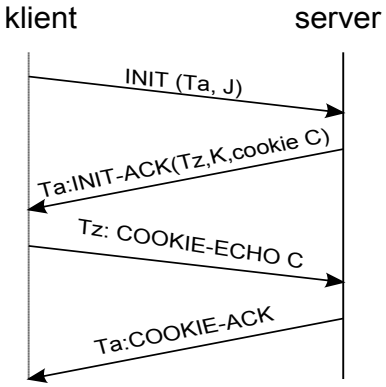
2. Klient zahájí aktivní otevření posláním zprávy, která implicitně otevře asociaci. Tímto krokem pošle serveru zprávu INIT, ve které pošle seznam svých IP adres, počáteční sekvenční číslo (J), příznak (Ta) sloužící k identifikaci všech paketů v této asociaci, počet požadovaných odchozích proudů a počet akceptovaných příchozích proudů.

3. Server odpoví na klientovu zprávu INIT svojí zprávou INIT-ACK, ve které předá klientovi svůj seznam IP adres, počáteční sekvenční číslo (K), příznak (Tz), počet odchozích a příchozích proudů a stavové cookies (C). Ta obsahuje vše, co server potřebuje k zajištění platnosti asociace. Stavové cookies je zde využito jako digitální podpis zpráv, aby nedocházelo při inicializaci asociace k podvržení spojení od případného útočníka.

4. Klient odpoví serveru na stavovou cookie zprávou COOKIE-ECHO, tato zpráva může už také obsahovat uživatelská data přidaná do stejného paketu.

5. Server odpoví zprávou COOKIE-ACK, že cookies je korektní a asociace byla ustanovena. S touto zprávou mohou být v paketu také poslána uživatelská data.

K navázání asociace jsou zapotřebí čtyři pakety (zprávy) a proto je tento proces nazván čtyřcestná synchronizace (Four-way handshake).

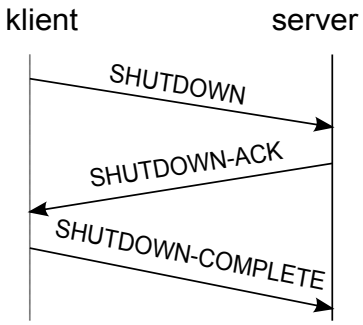


Obrázek 2.7: SCTP asociace komunikace [17]

Asociaci je možno ukončit dvěma způsoby. Tím prvním je přerušení (Abort). Tento způsob ukončení zahodí všechna data čekající na obou stranách komunikace. Strana, které se rozhodne pro toto ukončení pošle zprávu ABORT, přičemž nepotvrzená data už nebudou potvrzena. Spolu se zprávou ABORT pošle verifikační příznak. V tomto paketu už nejsou žádná data. Příjemce zprávy ABORT na ni nemusí odpovídat. Příjemce ověří verifikační příznak, odstraní asociaci a tuto skutečnost oznámí vyšší vrstvě komunikace (aplikaci).

Druhým, elegantnějším způsobem ukončení je zastavení (Shutdown) (viz obr. 2.8). Strana, která se rozhodne ukončit komunikaci, pošle po potvrzení všech nepotvrzených

dat zprávu SHUTDOWN a zastaví přijímání nových dat. Druhá strana komunikace přijme SHUTDOWN a po potvrzení nepotvrzených dat na něj odpoví (SHUTDOWN-ACK). Strana, která žádala o ukončení, přijme SHUTDOWN-ACK, pošle zprávu SHUTDOWN-COMPLETE, odstraní asociaci a ukončí komunikaci. Druhá strana přijme SHUTDOWN-COMPLETE a také odstraní asociaci a ukončí komunikaci.



Obrázek 2.8: SCTP ukončení komunikace [17]

2.4 Protokol IP

Protokol IP (Internet Protocol) [12] je základním protokolem síťové vrstvy. IP protokol je určen k přenosu datagramů (Internet datagram – IP datagram) od zdroje k cíle, který je identifikován IP adresou, přes propojené sítě. Jedná se o nespolehlivou, nespojovanou, datagramově orientovanou službu. Protokol neposkytuje žádné mechanismy zajišťující spolehlivost, řízení toku, sekvencování, nebo jiné služby, které poskytuje např. protokol transportní vrstvy TCP. Tyto služby mohou být v případě potřeby dodány protokoly transportní vrstvy, které tyto vlastnosti přenosu zajišťují.

Internetový modul využívá IP adresy datagramu k jeho přenosu na cílové rozhraní, které je touto adresou identifikováno. Tento proces se nazývá směrování. Internetový modul se nachází v každém počítači využívajícím nebo poskytujícím internetovou komunikaci a na každé bráně (gateway) propojující sítě. Všechny využívají stejná pravidla pro interpretaci adresy. Navíc mají k dispozici postupy, na základě kterých rozhodují o směrování datagramů. Každý IP datagram v síti je považován za nezávislý k ostatním datagramům.

Detekce chyb přenosu IP protokolu je reportována protokolem ICMP – Internet Control Message Protocol [11], který je v internetovém modulu implementován.

V dnešní době jsou využívány dvě verze internetového protokolu. Je to dnes už zastarávající a postupně nedostačující IPv4 a jeho náhrada – IPv6.

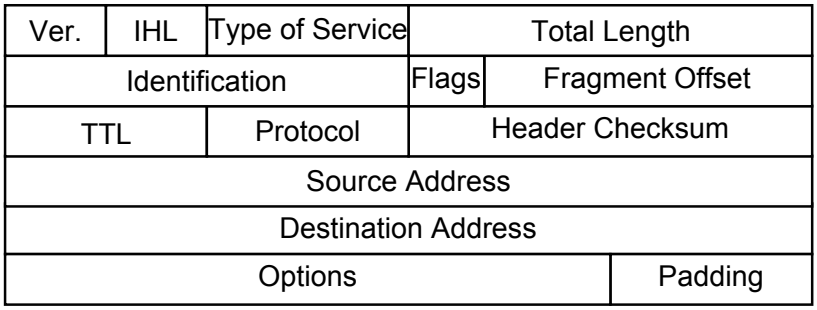
Hlavička IPv4 datagramu obsahuje položky (viz obr. 2.9):

- Verzi IP protokolu (Version).
- Délku hlavičky datagramu (IHL).

- Typ služby (Type of Service) – Původně určen k indikaci žádané kvality služby. Jednalo se o sadu parametrů charakterizující výběr parametrů služeb poskytovaných při přenosu konkrétní sítí (zpoždění, požadovaná šířka pásma, nejnižší cena přenosu). Postupně byla tato služba předefinována na tzv. diferencované služby (Differentiated Services)
- Celkovou délku datagramu (Total Length).
- Identifikátor (Identification) – přiřadí odesílatel a při fragmentaci mají všechny fragmenty datagramu tento identifikátor stejný.
- Příznaky (Flags) – týkají se fragmentace. Obsahuje informace o tom, zda může být datagram fragmentován a pokud je fragmentován, informuje o tom, zda za ním následuje další fragment nebo tento byl poslední.
- Offset fragmentu (Fragment Offset) – pozice části datagramu (fragmentu) v původním datagramu před fragmentací.
- TTL (Time to Live) – Indikuje maximální životnost IP datagramu v síti. TTL je nastaveno odesílatelem a na každém zařízením, kde je hlavička IP datagramu zpracovávána (obvykle směrovače) je tato hodnota snížena. Pokud TTL dosáhne nuly, datagram není doručen a je zahozen.
- Protokol vyšší vrstvy (Protocol) – jsou mu předána data po doručení.
- Kontrolní součet hlavičky (Header Checksum) – Slouží ke kontrole hlavičky IP datagramu po přenosu (netýká se dat datagramu). V případě chybného kontrolního součtu je tato hlavička zahozena zařízením, které tuto chybu odhalilo.
- Zdrojová IP adresa (Source Address).
- Cílová IP adresa (Destination Address).
- Volby (Options) – Zajišťují rozšiřující volby komunikace, které mohou být v některých situacích užitečné, ale nejsou nezbytně nutné pro běžnou komunikaci. Tyto volby se týkají časového razítka, bezpečnosti a speciálního směrování.
- Zarovnání (Padding) – zaokrouhlení délky hlavičky.

Za IP hlavičkou jsou připojena data. IPv4 adresa má délku 32 bitů (4 osmi-bitové skupiny odděleny tečkou). Protokol IP dále poskytuje možnost fragmentace, pro jejíž potřeby je vyhrazeno několik položek hlavičky (identifikátor, příznaky, offset fragmentu). Pokud je datagram přenášen přes síť s menší velikostí rámce (velikost MTU dané přenosové technologie), je fragmentován zařízením, na jehož výstupním rozhraní se tato síť nachází. Datagram je rozložen na menší části (podle velikosti MTU přenosové technologie v dané síti). Tyto části jsou poté znovu složeny na síťové vrstvě cílového rozhraní na základě informací ve hlavičce IP datagramu.

Protokol IPv4 postupem času přestal dostačovat potřebám internetu. Nedostatkem byl především maximální počet IP adres. Tento i jiné nedostatky řeší protokol IPv6 [6]. Tento protokol řeší nedostatek adres rozšířením adresového prostoru. Toho je dosaženo zvětšením velikosti IP adresy z 32 bitů u IPv4 na 128 bitů u IPv6. V protokolu IPv6 dochází také ke zjednodušení hlavičky IP datagramu. IPv6 hlavička (viz obr. 2.10) obsahuje jen základní



Obrázek 2.9: Hlavička IPv4 datagramu [12]

informace a oproti IPv4 hlavičce má pevnou délku. IPv6 také poskytuje možnosti autentizace a ochranu osobních údajů.

IPv6 hlavička obsahuje položky (viz obr. 2.10):

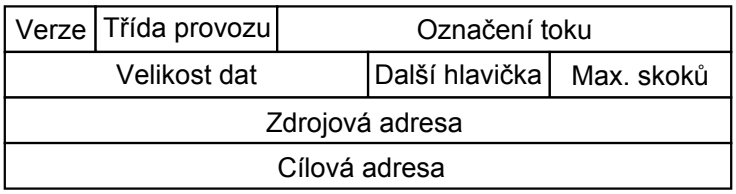
- Verze IP protokolu (Version).
- Třída provozu (Traffic Class) – zařazení datagramu do určité třídy provozu (např. pro určení priorit).
- Označení toku (Flow Label) – odesílatel označí pakety příslušející jedné komunikaci. Pomocí této značky směrovač rozpozná, ke kterému toku paket náleží.
- Velikost dat (Payload Length) – velikost obsahu.
- Další hlavička (Next Header) – identifikace následující rozšiřující hlavičky (jejího typu).
- Maximum skoků (Hop Limit) – obdoba TTL u IPv4 datagramu. Jedná se o číslo identifikující maximální počet směrovačů, kterými může datagram projít než vyprší jeho životnost. Tato hodnota se na každém směrovači sníží. Pokud dosáhne nuly, je datagram zahozen a odesílatel je o tom informován ICMP zprávou.
- Zdrojová IP adresa (Source Address).
- Cílová IP adresa (Destination Address) .

IPv6 adresa má délku 128 bitů rozdělených do osmi skupin až čtyř hexadecimálních čísel. Skupiny jsou odděleny dvojtečkou.

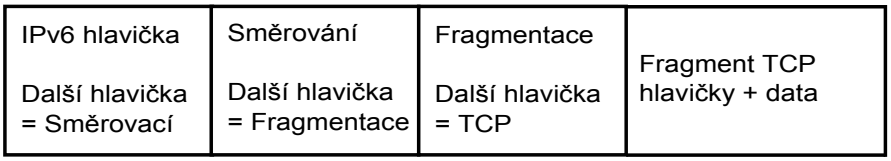
Jelikož byla optimalizována délka IPv6 hlavičky (některé položky z ní byly vyjmuty), zavádí IPv6 tzv. rozšiřující hlavičky (viz obr. 2.11). Tyto hlavičky se nacházejí mezi IPv6 hlavičkou a hlavičkou vyšší vrstvy komunikace (obvykle transportní). Každá následující hlavička je odkázána předchozí a pořadí hlaviček je uzpůsobeno požadavkům při přenosu. To znamená, že hlavičky, které musí být přečteny na každém směrovači jsou před těmi, které náleží až cílovému rozhraní. Tímto uspořádáním je zvýšena rychlost přenosu (směrovače čtou jen informace, které jsou pro ně určené).

Hlavičky IPv6 datagramu mají následující pořadí (viz obr. 2.11):

1. IPv6 hlavička.
2. Volby pro všechny uzly (Hop-by-Hop Options) – Tyto volby jsou určeny všem uzlům, kterými datagram prochází během přenosu (je přes ně směrován).
3. Volby cílového rozhraní – v případě, že je datagram směrován přes více uzlů (viz. hlavička směrování).
4. Směrování (Routing header) – Umožňuje směrovat nejen na základě cílové adresy, ale i přes určené uzly. V hlavičce směrování je uveden seznam adres, přes které musí datagram projít.
5. Fragmentace (Fragment header) – Fragmentace IPv6 se liší od IPv4. Zatímco u IPv4 může fragmentaci provést libovolný uzel, na jehož výstupním rozhraní je nižší MTU než velikost paketu, u IPv6 provádí fragmentaci jen odesílatel. Odesílatel odešle datagram a když na cestě narazí na linku s nižším MTU, je zahozen a odesílatel je o tomto informován ICMP zprávou obsahující velikost MTU. Na základě této zprávy provede odesílatel fragmentaci a fragmenty znovu odešle. Paket má dvě části (fragmentovatelnou a nefragmentovatelnou). Nefragmentovatelná část obsahuje IPv6 hlavičku a všechny rozšiřující hlavičky potřebné ke směrování. Tato část nesmí být fragmentována a je součástí každého fragmentu. Do této části patří i hlavička fragmentace.
6. Autentizační hlavička (Authentication header).
7. Zapouzdření bezpečnostního užitečného zatížení (Encapsulating Security Payload).
8. Volby cílového rozhraní – Jedná se o volby určené pro rozhraní příjemce.
9. Hlavička vyšší vrstvy (protokolů transportní vrstvy).



Obrázek 2.10: Hlavička IPv6 datagramu [6]

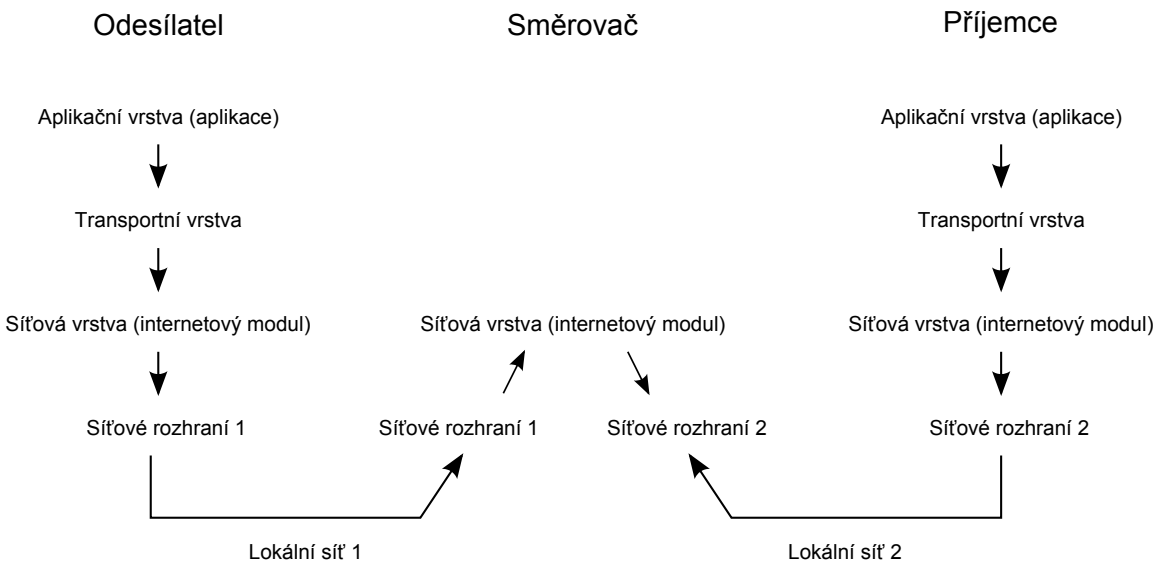


Obrázek 2.11: Příklad řazení rozšiřujících hlaviček IPv6 datagramu [6]

2.4.1 Komunikace

Účelem IP protokolu je přenos IP datagramu propojenými sítěmi od zdroje k cíly. Datagram je posílán od jednoho síťového modulu k dalšímu, dokud nedosáhne cíle. Posílání je založeno na IP adrese. Tento proces je nazván směrování (viz obr. 2.12).

Přenos datagramu začíná předáním segmentu dat z vyšší (transportní) vrstvy síťové vrstvě (internetovému modulu) k odeslání na cílovou adresu. Na síťové vrstvě je k tomuto segmentu přidána hlavička (obsahuje IP adresu cíle a další parametry přenosu). Internetový modul určí L2 adresu (v lokální síti), na kterou pošle takto vzniklý datagram. Pokud je datagram posílán do jiné sítě, tato adresa náleží bráně ze sítě, ve které se nachází. Datagram s L2 adresou předá nižší vrstvě, která vytvoří rámec a pošle na cílovou adresu lokální sítě (cílové rozhraní). Zde je L2 adresa odstraněna a na základě IP adresy ve hlavičce IP datagramu je rozhudnuto, kam poslat datagram dále (do jaké další sítě). Takto datagram postupuje přes sítě, dokud nedorazí na síťové rozhraní identifikované cílovou IP adresou datagramu. Internetový modul na tomto rozhraní rozhodne, že datagram náleží jemu a data z datagramu předá vyšší vrstvě [12].



Obrázek 2.12: Hlavička IPv6 datagramu [6]

2.5 Komunikace na linkové vrstvě

Linková vrstva se zabývá přenosem dat na konkrétní lince mezi dvěma rozhraními v jedné síti. Poskytuje spojení mezi dvěma sousedními systémy. Přenos probíhá prostřednictvím rámců. Datagram z vyšší (síťové) vrstvy je zabalen do rámce a je mu přidána lokální adresa (MAC adresa), která náleží síťovému rozhraní v lokální síti.

Tato vrstva zajišťuje také spolehlivost doručení informace, kontrolu chyb při komunikaci

mezi dvěma uzly a synchronizaci rámců. Linková vrstva je rozdělena do dvou podvrstev: vrstva řízení přístupu k médiu (Media Access Control – MAC) a vrstva řízení logického spoje (Logical Link Control – LLC). MAC kontroluje jak počítač v síti získává přístup k datům a svolení k jejich vysílání. LLC kontroluje synchronizaci rámců, kontrolu toku a kontrolu chyb [3].

Na linkové vrstvě jsou pro přenos využity různé technologie (Ethernet, PPP, Token Ring, FDDI, Frame Relay, ...). Tyto technologie se mimo jiné od sebe odlišují různou velikostí MTU (maximální velikost rámce, který může být po lokální síti dané technologie přenášen).

Na vyšších vrstvách lze jako prostředek komunikace použít schránky. Na linkové vrstvě k tomuto účelu slouží knihovny k tomu určené. Tyto knihovny poskytují prostředky pro tvorbu paketů, přidávání hlaviček a jejich přímý zápis na síťové rozhraní. Zároveň je možné pakety ze síťového rozhraní číst.

Jedná se například o:

- C – libnet [8] a libpcap [1] pro vytváření paketů, jejich vysílání na síťové rozhraní a pro jejich odchytávání a čtení.
- Perl – modul Net::Pcap [14] a kolekce modulů libnet [2]
- Python – modul pcap [16] a pylibnet [7]
- Java – jNetPcap [15] – wrapper pro knihovny libpcap a WinPcap – odesílání i odchytávání paketů
- C# – Pcap.Net [5] – wrapper pro WinPcap – umožňuje odesílání i odchytávání paketů.

Kapitola 3

Implementace protokolů

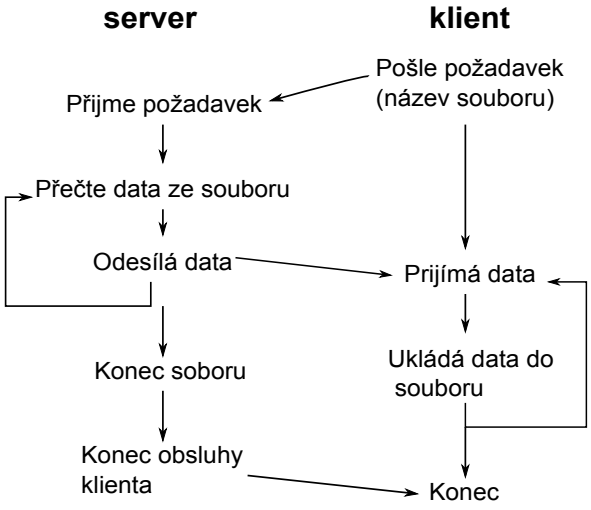
Základním prostředkem implementace síťové komunikace je schránka (socket). Tyto schránky jsou využívány protokoly transportní vrstvy jako vstupně/výstupní bod komunikace. Na úrovni nižších vrstev (síťová, linková) se používají schránky typu raw. Krom schránek lze také využít knihovny určené k vytváření paketů, jejich vysílání na síťové rozhraní a pro jejich odchytávání a čtení.

Implementovaná komunikace se týká dvou typů aplikací. Jednou z těchto aplikací je server (iterativní/konkurentní), který běží nepřetržitě. Server naslouchá na zadané adrese, případně portu, čeká na požadavky klienta a obsluhuje je. Druhou aplikací je klient, který od serveru požaduje služby (v tomto případě data). Pokud tato data server má, odešle je klientovi. Klient tato data přijme a poté ukončí svoji činnost.

Konkurentní server je třeba implementovat jako paralelní komunikaci s více klienty najednou. Toho je dosaženo prostředky jednotlivých programovacích jazyků pro tyto potřeby. Těmito prostředky jsou mechanismy jednotlivých jazyků pro paralelní programování. Jedná se o vytvoření nového procesu nebo vlákna, které komunikaci s jedním konkrétním klientem obstará bez ovlivnění ostatních.

3.1 Popis implementované aplikace

Implementovaná komunikace (viz Obrázek 3.1) se týká binárního přenosu souboru mezi serverem a klientem. Obě tyto strany komunikace vytvoří schránku nebo jiný prostředek vzájemné komunikace. Komunikaci (po navázání spojení mezi oběma stranami) zahajuje klient odesláním žádosti o soubor na serveru (odešle název souboru). Server tuto žádost přijme a pokud má k souboru přístup, otevře ho a začne v cyklu odesílat čtená data ze souboru. Klient otevře soubor pro zápis a data přijímaná od serveru do něj zapisuje. Po skončení přenosu klient uzavře schránku/přeruší spojení se serverem (signalizace ukončení přenosu závisí na protokolu). Zároveň jsou uzavřeny soubory jak pro čtení na straně serveru, tak i pro zápis na straně klienta. Poté může server obsluhovat dalšího klienta, pokud je iterativní. Pokud je server konkurentní, může obsluhovat více klientů najednou (zvláštním procesem nebo vláknem).



Obrázek 3.1: Implementovaná aplikace

3.2 Protokol TCP

Při TCP komunikaci (viz Obrázek 3.2) je nejdříve vytvořena schránka serveru a klienta – `socket()`. Tato schránka je pojmenována (je jí přiřazena adresa a port) – `bind()`. Schránka serveru poté na zadané adrese naslouchá příchozí komunikaci – `listen()`. Schránka na straně klienta se připojí k serveru – `connect()`. Server, pokud může, toto připojení klienta přijme – `accept()`. Tímto dojde k navázání spojení mezi klientem a serverem a může být zahájena samotná komunikace (přenos souboru). V rámci komunikace nejprve klient požádá server o soubor (pošle mu jeho název) – `write(filename)`. Server tento požadavek přijme – `read(filename)`. Poté server v cyklu posílá data (`write(data)`) z požadovaného souboru klientovi a ten tato data přijímá – `read(data)`. Přijatá data klient zapisuje do souboru. Po dokončení této komunikace (soubor je přenesen) je klient odpojen od serveru a ukončí svou činnost uzavřením schránky – `close()`.

U protokolu TCP není třeba se zabývat při implementaci spolehlivostí. Spolehlivost doručení dat (případně oznámení chyby) je součástí protokolu TCP.

Jazyk/funkce	socket()	bind()	listen()	accept()	close()
C/C++	socket()	bind()	listen()	accept()	close()
Perl	IO:Socket:INET->new()			accept()	close()
Python	socket.socket()	bind()	listen()	accept()	close()
Java	new ServerSocket()			accept()	close()
C#	new TcpListener()			AcceptSocket()	Stop()

Tabulka 3.1: Funkce a metody použité pro implementaci TCP serveru

Jazyk/funkce	socket()	connect()	close()
C/C++	socket()	connect()	close()
Perl	IO:Socket:INET->new()		close()
Python	socket.socket()	connect()	close()
Java	new Socket()		close()
C#	new TcpClient		Close()

Tabulka 3.2: Funkce a metody použité pro implementaci TCP klienta

Jazyk/funkce	read()	write()
C/C++	read()	write()
Perl	operátor <>	print()
Python	recv()	send()
Java	DataInputStream.read()	DataOutputStream.write()
C#	NetworkStream.Read()	NetworkStream.Write

Tabulka 3.3: Funkce a metody použité pro implementaci přenosu dat protokolem TCP

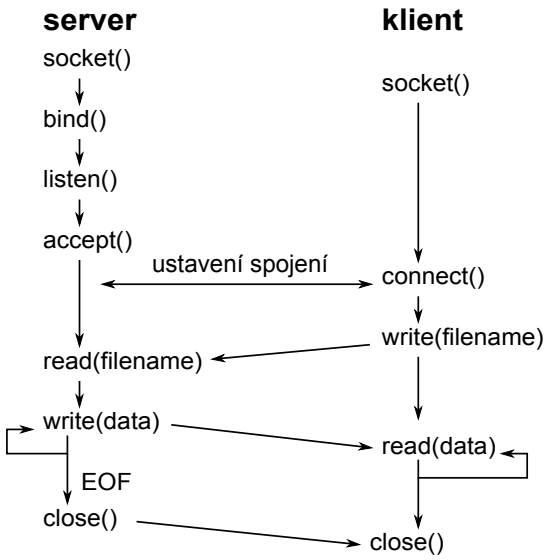
Jazyk C/C++

V jazyce C byly pro implementaci TCP komunikace pomocí schránek využity následující hlavičkové soubory:

- `<netinet/in.h>` – obsahuje adresovou strukturu schránky `sockaddr_in` (slouží pro uložení protokolové rodiny, portu a adresy schránky).
- `<arpa/inet.h>` – obsahuje funkce pro konverze adresy (převede IP adresu z řetězcové reprezentace na formát odpovídající formátu uložení IP adresy ve struktuře `in_addr)`.
- `<sys/socket.h>`[1] – funkce a konstanty využité pro práci se schránkami.

Na straně TCP serveru (viz Obrázek 3.3) je vytvořena schránka pro naslouchání požadavkům klienta. Schránka je vytvořena funkcí `socket()` s parametry určujícími protokolovou rodinu (`AF_INET`), typ schránky (`SOCK_STREAM` – jedná se o stream socket pro protokol TCP) a protokol použitý pro komunikaci (`IPPROTO_TCP` – jedná se o TCP komunikaci). Funkce vrací deskriptor schránky. Poté je vytvořena adresová struktura `sockaddr_in` a vyplněna údaji, jimiž jsou protokolová rodina, IP adresa a port. Pomocí této adresové struktury je schránka pojmenována funkcí `bind()`, jejímiž parametry jsou krom této struktury také velikost této struktury a deskriptor schránky. Tato schránka je nastavena pro naslouchání (funkce `listen()` s parametry deskriptor schránky a velikost fronty klientů, kteří mohou čekat na obsluhu). Pokud se klient připojí k serveru, je toto připojení přijato funkcí `accept()`, která vrací deskriptor schránky pro komunikaci s klientem. Parametry této funkce jsou deskriptor schránky, struktura `sockaddr_in` pro adresní informace od klienta a velikost této struktury. Po přijetí klienta je funkcí `read()` s parametry deskriptor schránky klienta, vyrovnávací paměť pro uložení přijatých dat a velikost přijímaných dat,

[1] http://pubs.opengroup.org/onlinepubs/7908799/xns/syssocket.h.html



Obrázek 3.2: Implementace TCP komunikace

přijato jméno souboru, který klient od serveru požaduje. Data přečtená ze souboru jsou odesílána klientovi funkcí `write()` s parametry desriptor schránky, data a velikost odesílaných dat. Po odeslání všech dat je schránka pro obsluhu klienta uzavřena funkcí `close()`.

Na straně klienta (viz Obrázek 3.4) je vytvořena schránka funkcí `socket()`. Funkce zde má stejné parametry jako v případě serveru. Schránka je připojena funkcí `connect()` k serveru. Tato funkce je volána s parametry, kterými jsou deskriptor schránky, adresní strukturou pro připojení `sockaddr_in`, obsahující informace pro připojení k serveru (IP adresu a port), a její velikost. Po navázání spojení je odeslán název souboru funkcí `write()`. Příchozí data jsou poté přijímána funkcí `read()` a zapsána do výstupního souboru. Po ukončení komunikace je uzavřena schránka klienta funkcí `close()` a klient je ukončen.

Pro implementaci konkurentního serveru lze využít funkce `fork()` implementované v knihovně `<unistd.h>`.

```
int main(int argc, char **argv){
    int listenfd = socket(AF_INET, SOCK_STREAM, 0);

    servaddr.sin_family = PF_INET;
    servaddr.sin_port = htons(port);
    servaddr.sin_addr.s_addr  = INADDR_ANY;
    bind(listenfd,  (struct sockaddr *) &servaddr, sizeof(servaddr);
    listen(listenfd, 10);

    while (1){
      int connfd = accept(listenfd,  (struct sockaddr *) &cliaddr, &clilen);
      read(connfd, &filename, sizeof(filename));
      write(connfd, data, sizeof(data));
      close(connfd);}}
```

Obrázek 3.3: Prostředky komunikace TCP serveru v jazyce C

```
int main(int argc, char **argv){

    servaddr.sin_family = AF_INET;
    servaddr.sin_port = htons(port);
    servaddr.sin_addr.s_addr = address;

    int sockfd = socket(AF_INET, SOCK_STREAM, 0);

    connect(sockfd, (struct sockaddr *) &servaddr, sizeof(servaddr));

    write(sockfd, filename, sizeof(filename));
    read(sockfd, temp, sizeof(temp));

    close(sockfd);
}
```

Obrázek 3.4: Prostředky komunikace TCP klienta v jazyce C

Jazyk Perl

V jazyce Perl je TCP server (viz. Obrázek 3.5) implementován třídou `IO::Socket::INET` stejnojmenného modulu[2]. Novou instancí této třídy metodou `new()` je vytvořena schránka serveru. Parametry této metody jsou protokol (v tomto případě "tcp"), port na kterém server naslouchá, Listen určující počet klientů a Reuse pro uvolnění portu po ukončení programu. Tato schránka naslouchá požadavkům klientů. Klient je serverem přijat metodou `accept()` třídy `IO::Socket::INET`. Tato metoda vrací deskriptor schránky pro obsluhu klienta. Na této schránce je nejprve přijat název souboru pomocí operátoru `<>`, který pošle klient. Tento operátor čte data ze schránky prostřednictvím jejího deskriptoru – `<deskriptor_schránky>`. Tento soubor je otevřen (`open()`). Funkcí `sysread()` jsou ze souboru čtena dat. Přečtená data jsou funkcí `print` zapsána na schránku. Po přenesení celého souboru jsou soubor a schránka pro komunikaci s klientem uzavřeny funkcí `close`.

Na straně klienta (viz. Obrázek 3.6) je nejprve vytvořena schránka pro komunikaci se serverem pomocí třídy `IO::Socket::INET`. Narozdíl od serveru je zde kromě čísla portu zadána i IP adresa serveru (chybí položky Listen a Reuse). Zadáním IP adresy je schránka po svém vytvoření připojena k serveru (není zde třeba funkce `connect()`). Prostřednictvím této schránky je odeslán název požadovaného souboru funkcí `print` na schránku. Data od serveru jsou přijímána pomocí operátoru `<deskriptor_schránky>` a jsou zapisována do souboru. Po dokončení přenosu je schránka uzavřena funkcí `close`.

Stejně jako v jazyce C lze i v Perlu implementovat konkurentní server. Konkurentní server je realizován funkcí `fork()`, která vytvoří nezávislý proces pro obsluhu klienta.

Jazyk Python

TCP komunikace je v jazyce Python implementována pomocí modulu `socket`[3]. Pomocí tohoto modulu je nejdříve na straně serveru (viz Obrázek 3.7) vytvořen objekt schránky pomocí metody `socket()`, jejímiž parametry jsou `AF_INET` a `SOCK_STREAM`, identifikující

[2] http://search.cpan.org/ gbarr/IO-1.25/lib/IO/Socket/INET.pm
[3] http://docs.python.org/library/socket.html

```
my $server = IO::Socket::INET->new( Proto => "tcp",
                                    LocalPort => $port,
                                    Listen => 1, Reuse => 1);

while(my $client = $server->accept()){
  $filename= <$client>;
  print $client $data;

  close $client;}
```

Obrázek 3.5: Prostředky komunikace TCP serveru v jazyce Perl

```
my $server = IO::Socket::INET->new(Proto => "tcp",
                                   PeerAddr => inet_ntoa($ip),
                                   PeerPort => $port );

print $server $filename_send; #odeslani nazvu souboru
$data = <$server>; #cteni dat od serveru

close $server;
```

Obrázek 3.6: Prostředky komunikace TCP klienta v jazyce Perl

protokolovou rodinu a typ schránky. Tato schránka je pojmenována metodou `bind()` s parametrem, kterým je prázdný řetězec zastupující adresu (`INADDR_ANY`) a číslo portu. Na tomto portu začne server naslouchat (metoda `listen()` s parametrem délka fronty čekajících klientů). Klient je přijat metodou `accept()` objektu schránky. Tato metoda vrací deskriptor schránky prostřednictvím které bude server s daným klientem komunikovat. Na této schránce je přijat požadavek na soubor od klienta metodou `recv()`, jejímž parametrem je velikost přijímaných dat. Tento soubor je otevřen funkcí `open()` pro čtení. Ze souboru jsou čtena data a odesílána pomocí metody `send()` objektu schránky. Po ukončení čtení dat ze souboru je schránka pro obsluhu klienta uzavřena metodou `close()`.

Na straně klienta (viz Obrázek 3.8) je vytvořena schránka pro komunikaci se serverem. Stejně jako v případě serveru se jedná o objekt modulu `socket`. Tato schránka je připojena k serveru metodou `connect()`, jejímiž parametry jsou adresa serveru a port na kterém server naslouchá. Metodou `send()` je serveru odeslán název požadovaného souboru. Metodou `recv()` objektu schránky jsou přijímána data. Velikost přijímaných dat je ovlivněna parametrem této metody. Přijatá data jsou zapsána do souboru. Po ukončení přenosu je schránka klienta uzavřena metodou `close()` a klient je ukončen.

Konkurentní server lze v případě jazyka Python implementovat funkcí `fork()`, která je součástí modulu `os`.

Jazyk Java

V jazyce Java jsou schránky implementovány pomocí balíčku `java.net.*`[4]. Schránka serveru (viz Obrázek 3.9) je vytvořena jako instance třídy `ServerSocket` metodou `new()` s parametrem, kterým je číslo portu, na kterém bude server naslouchat. Server čeká na příchozí

[4] http://docs.oracle.com/javase/1.5.0/docs/api/index.html

```
def main(argv=None):
  s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
  s.bind(("", port))
  s.listen(1)

  while(1):
    conn, addr = s.accept() #pripojeni klienta

    filename = conn.recv(1024) #jmeno souboru od klienta
    conn.send(data) #odeslani dat klientovi

    conn.close()
```

Obrázek 3.7: Prostředky komunikace TCP serveru v jazyce Python

```
def main(argv=None):
  s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
  s.connect((addr, port))

  send(filename) #odeslani nazvu souboru
  data = s.recv(1024) #prijeti dat

  s.close()
```

Obrázek 3.8: Prostředky komunikace TCP klienta v jazyce Python

požadavek od klienta a metodou `accept()` ho přijme. Tato metoda vrací novou schránku (objekt třídy `Socket`) pro komunikaci s konkrétní klientem. Pro komunikaci prostřednictvím schránek je třeba vytvořit vstupní a výstupní tok (`InputStream` a `OutputStream`) a z nich datové toky pro vstup a výstup dat. Třídy těchto toků obsahuje balíček `java.io.*`. Na vstupním datovém toku je nejprve přijat požadavek na soubor od klienta (metoda `readUTF()`). Z tohoto souboru jsou čtena data a odeslána metodou `write()` výstupního datového toku. Po dokončení přenosu jsou datové toky a schránka pro obsluhu klienta uzavřeny metodou `close()`.

Na straně klienta (viz Obrázek 3.10) je metodou `new()` třídy `Socket` vytořena schránka pro komunikaci se serverem. Parametry metody jsou IP adresa serveru a port, na kterém naslouchá. Jako v případě serveru jsou i zde vytvořeny vstupně/výstupní datové toky. Metodou `writeUTF()` výstupního datového toku je odeslán název souboru serveru a následně jsou metodou `read()` vstupního datového toku čtena data přijímaná od serveru. Vyrovnávací paměť pro příjem těchto dat je parametrem metody `read()`. Po ukončení přenosu jsou všechny datové toky a schránka klienta uzavřeny metodou `close()` a klient je ukončen.

Konkurentní server je implementován pomocí třídy `Thread`. Pomocí této třídy je vytvořeno nové vláknu, které v metodě `run()` obsahuje popis svého chování (komunikace serveru s klientem).

```
class server{
    public static void main(String args[]) throws IOException{
        ServerSocket s = new ServerSocket(Integer.parseInt(port));

        while (true){
            Socket s1=s.accept();

            InputStream s1In = s1.getInputStream();
            DataInputStream dis = new DataInputStream(s1In);
            String filename = dis.readUTF();
            OutputStream s1out = s1.getOutputStream();
            DataOutputStream dos = new DataOutputStream (s1out);

            dos.write(data);
            s1.close();}}}
```

Obrázek 3.9: TCP server v jazyce Java

```
class client{
    public static void main(String args[]) throws IOException{
        Socket s1 = new Socket(address,Integer.parseInt(port));

        OutputStream s1out = s1.getOutputStream();
        DataOutputStream dos = new DataOutputStream (s1out);
        dos.writeUTF(filename);  //odeslat jmeno souboru

        InputStream s1In = s1.getInputStream();
        DataInputStream dis = new DataInputStream(s1In);
        data_size = dis.read(data);
        s1.close();}}
```

Obrázek 3.10: TCP klient v jazyce Java

Jazyk C#

Síťová komunikace protokolu TCP je v jazyce C# implementována prostřednictvím jmenného prostoru `System.Net.Sockets`[5]. Schránka serveru (viz Obrázek 3.11) je vytvořen jako instance třídy `TcpListener` – vytvoří schránku, která naslouchá na zadaném portu (tento port je zadán parametrem metody `new()`). Schránka je metodou `Start()` spuštěna. Metodou `AcceptSocket()` je přijat klient, který se chce k serveru připojit. Tato metoda vrací schránku pro komunikaci s klientem (`Socket()`). Pro komunikaci prostřednictvým této schránky je vytvořen, instancí třídy `networkStream`, metodou `new()` s parametrem deskriptor schránky klienta, stream pro komunikaci s klientem. Z tohoto streamu je metodou `ReadLine()` třídy `StreamReader` přečten požadavek od klienta. Soubor, který klient požaduje je odeslán metodou `Write()` streamu. Po ukončení přenosu jsou stream i schránka pro obsluhu klienta uzavřeny metodou `Close()`.

Na straně klienta (viz Obrázek 3.12) je schránka instancí třídy `TcpClient`. Metodě `new()` této třídy jsou jako parametry předány IP adresa serveru a port na kterém na-

[5] http://msdn.microsoft.com/en-us/library/system.net.sockets.aspx

slouchá. Z této schránky je metodou `GetStream()` získán `NetworkStream`. Pomocí třídy `StreamWriter()` je na něj metodou `WriteLine()` zapsán název souboru. Data od serveru jsou čtena metodou `Read()` třídy `NetworkStream`. Po dokončení přenosu je `NetworkStream` uzavřen stejně jako schránka metodou `Close()` a klient je ukončen.

Pro potřeby komunikace konkurentního serveru je využito třídy `Thread`, která metodou `Start()` spustí nové vlákno pro komunikaci klienta. Třída `Thread` se nachází ve jmenném prostoru `System.Threading`.

```
public class Server{
    public static void Main(string[] args){
        TcpListener tcpListener =
            new TcpListener(IPAddress.Any ,Convert.ToInt32(port));
        tcpListener.Start();

        while (true){
            Socket socketForClient = tcpListener.AcceptSocket();
            if (socketForClient.Connected){
                NetworkStream networkStream =
                    new NetworkStream(socketForClient);
                System.IO.StreamReader streamReader =
                    new System.IO.StreamReader(networkStream);
                string filename = streamReader.ReadLine();

                networkStream.Write(data, 0, data.Length);
                socketForClient.Close();}}}}
```

Obrázek 3.11: TCP server v jazyce C#

```
public class Client{
    static public void Main( string[] args ){
        TcpClient  socketForServer =
            new TcpClient(address, Convert.ToInt32(port));

        NetworkStream networkStream = socketForServer.GetStream();
        System.IO.StreamWriter streamWriter =
            new System.IO.StreamWriter(networkStream);

        streamWriter.WriteLine(filename);
        int i = networkStream.Read(data, 0, data.Length);
        socketForServer.Close();}}
```

Obrázek 3.12: TCP klient v jazyce C#

3.3 Protokol UDP

Jelikož je UDP narozdíl od TCP nespojová služba, je nutné pro potřeby implementace přenosu dat doimplementovat k základní implementaci komunikace prvky, které budou zajišťovat spolehlivé doručování dat. Toho jsem dosáhl zavedením sekvenčních čísel při komunikaci a potvrzováním přijetí dat. Dále je schránce nastaven časovač, pro případ, že by došlo ke ztrátě dat.

Komunikace UDP (viz Obrázek 3.13) je zahájena vytvořením schránky serveru a klienta – `socket()`. Tato schránka je na straně serveru pojmenována – `bind()`. Narozdíl od protokolu TCP zde nedochází k navázání spojení, ale obě strany komunikují prostřednictvím adresovaných zpráv. Ty jsou odesílány funkcí `sendto()` a přijímány funkcí `recvfrom()`. Pro potřeby vzorové implementace jsem potřeboval odlišit zprávy od jednotlivých klientů a tudíž bylo třeba zavést prvek synchronizace mezi klientem a serverem. Pokud chce klient komunikovat se serverem, zašle mu nejprve synchronizační zprávu – `sendto(START)`. Server tuto zprávu přijme (`recvfrom(START)`) a rozpozná. Pro tohoto nového klienta vytvoří novou schránku, která bude mít stejnou adresu, ale jiný port než původní schránka. Prostřednictvím této schránky odpoví server klientovi na synchronizační zprávu – `sendto(ACK)`. Klient přijetím této zprávy získá nový port serveru, se kterým bude dále komunikovat. Pošle na něj název souboru – `sendto(filename)`. Server přijme název souboru a pokud k němu má přístup, začne ho číst a odesílat serveru – `sendto(data)`. Součástí dat je i sekvenční číslo. Klient tato data přijímá – `recvfrom(data)` a na přijatá data odpoví – `sendto(ack)`. Tato odpověď obsahuje sekvenční číslo dat, která byla naposledy správně přijata. Na základě této odpovědi sever odešle další data. Pokud odpověď není přijata před vypršením časovače, jsou znovu odeslána data, která nebyla potvrzena. Pokud klient neobdrží data před vypršením jeho časovače, odešle znovu odpověď se sekvenčním číslem posledních korektně doručených dat (tímto informuje server, která data vyžaduje). O ukončení přenosu informuje server klienta zprávou obsahující sekvenční číslo s hodnotou -1. Po odeslání této zprávy server ukončí obsluhu tohoto klienta. Klient po obdržení této zprávy ukončí svou činnost uzavřením schránky – `close()`.

Jazyk/funkce	socket()	bind()	close()
C/C++	socket()	bind()	close()
Perl	socket()	bind()	close()
Python	socket.socket()	bind()	close()
Java	new DatagramSocket()		close()
C#	new Socket()	Bind()	Close()

Tabulka 3.4: Funkce a metody použité pro implementaci UDP serveru a klienta

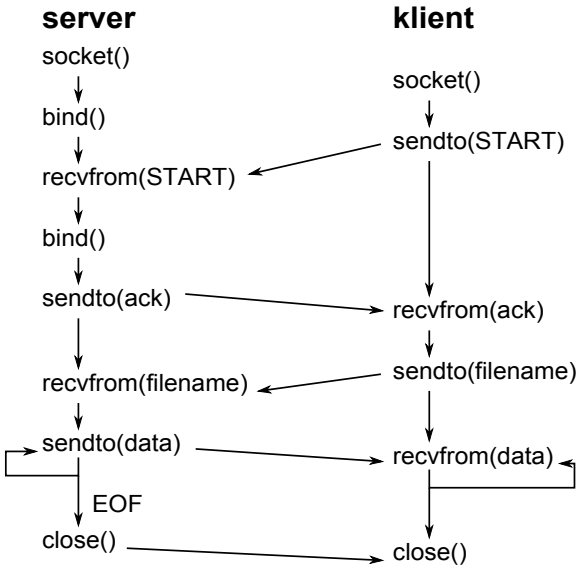
Jazyk C/C++

Pro implementaci komunikace v jazyce C je využit hlavičkový soubor `<sys/socket.h>` obsahující funkce a konstanty pro práci se schránkami. Dále jsou využity také hlavičkové soubory obsahující struktury a funkce pro práci s adresami (`<netinet/in.h>` a `<arpa/inet.h>`).

Na straně UDP serveru (viz Obrázek 3.14) je vytvořena schránka pro naslouchání požadavkům klienta. Schránka je vytvořena funkcí `socket()` s parametry určujícími protokolovou rodinu (`AF_INET`), typ schránky (`SOCK_DGRAM` – jedná se o datagramovou schránku pro

Jazyk/funkce	recvfrom()	sendto()
C/C++	recvfrom()	sendto()
Perl	recv()	send()
Python	recvfrom()	sendto()
Java	DatagramPacket.receive()	DatagramPacket.send()
C#	ReciveFrom()	SendTo()

Tabulka 3.5: Funkce a metody použité pro implementaci přenosu dat protokolem UDP



Obrázek 3.13: Implementace UDP komunikace

protokol UDP) a protokol použitý pro komunikaci (`IPPROTO_UDP` – jedná se o UDP komunikaci). Funkce vrací deskriptor schránky. Poté je vytvořena adresová struktura `sockaddr_in` a vyplněna údaji, jimiž jsou protokolová rodina, IP adresa a port. Pomocí této adresové struktury je schránka pojmenována funkcí `bind()`, jejímiž parametry jsou deskriptor schránky, adresní struktura serveru a její velikost. Po provedení synchronizace je nutné nastavit schránce časovač (pro případ, že by došlu k výpadku komunikace). To lze provést funkcí `setsockopt()`, jejímiž parametry jsou deskriptor schránky, úroveň volby `SOL_SOCKET`, jméno volby (`SO_RCVTIMEO` – maximální doba čekání na přijetí dat), hodnota volby. Poté je od klienta přijat požadavek funkcí `recvfrom()`. Ta má jako parametry deskriptor schránky, vyrovnávací paměť pro přijímaná data, její velikost, příznaky, strukturu `sockaddr` pro uložení informací o klientovi a její velikost. Data požadovaného souboru jsou klientovi odesílána funkcí `sendto()`, jejímiž parametry jsou deskriptor schránky, odesílaná data, jejich velikost, příznak, adresová struktura klienta a její velikost. Po ukončení přenosu je uzavřena schránka určená pro komunikaci s daným klientem funkcí `close()`.

Na straně klienta (viz Obrázek 3.15) je vytvořena schránka funkcí `socket()`. Funkce zde má stejné parametry jako v případě serveru. Schránce je stejně jako v případě serveru nastaven časovač funkcí `setsockopt()`. Dále je vytvořena adresní struktura `sockaddr_in`, obsahující informace pro připojení k serveru (IP adresu a port). Prostřednictvím této struktury je komunikováno se serverem. Po synchronizaci se serverem mu je odeslán název poža-

dovaného souboru funkcí `sendto()`. Poté jsou funkcí `recvfrom()` od serveru přijata data a uložena do souboru. Po ukončení přenosu je schránka klienta zavřena funkcí `close()`.

```
int main(int argc, char **argv){
    int sock = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);

   //vyplneni struktury servaddr

    struct timeval tv;  //struktura pro timeout
    tv.tv_sec = 0.5;
    setsockopt (sockfd, SOL_SOCKET, SO_RCVTIMEO, (char *)&tv, sizeof(tv));

    while(1){
      //synchronizace
      ...
      recvfrom(sockfd, &filename, sizeof(filename),
          0, &cliaddr, sizeof(cliaddr));
      sendto(sockfd, &Data, sizeof(Data), 0, &cliaddr,
          sizeof(cliaddr));}}
```

Obrázek 3.14: Prostředky komunikace UDP serveru v jazyce C

```
int main(int argc, char **argv){
    int sockfd = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP) ;

     //vyplneni struktury servaddr pro pripojeni k serveru

    struct timeval tv;  //strruktura pro timeout
    tv.tv_sec = 0.5;
    setsockopt (sockfd, SOL_SOCKET, SO_RCVTIMEO, (char *)&tv, sizeof(tv));
    //synchronizace
      ...
    sendto(sockfd, filename, BUFF_SIZE, 0,
        (struct sockaddr *) servaddr, len);
    recvfrom(sockfd, &data, sizeof(data), 0, &servaddr, sizeof(cliaddr));

    close(sockfd);}
```

Obrázek 3.15: Prostředky komunikace UDP klienta v jazyce C

Jazyk Perl

UDP komunikaci v jazyce Perl lze implementovat pomocí modulu `IO::Socket::INET`. Pomocí funkce `socket()` lze vytvořit schránku serveru (viz Obrázek 3.16), jejíž parametry jsou protokolová rodina (`AF_INET`), typ schránky (`SOCK_DGRAM`), a protokol (UDP). schránka je pojmenována funkcí `bind()` s parametry deskriptor schránky a adresní strukturou `sockaddr_in` serveru, jejímiž parametry jsou IP adresa (`INADDR_ANY`) a port. Po

synchronizaci s klientem je schránce nastaven časovač funkcí `setsockopt()`, jejímiž parametry jsou deskriptor schránky, úroveň volby (`SOL_SOCKET`), jméno volby (`SO_RCVTIMEO`), hodnota volby). Poté je funkcí `recv()` přijat název souboru požadovaného klientem. Tato funkce má jako parametry deskriptor schránky, vyrovnávací paměť pro přijímaná data, velikost přijímaných dat a příznak. Zároveň vrací adresní strukturu klienta. Té je využito funkcí `send()`, která se stará o odeslání dat požadovaného souboru. Parametry jsou dále schránka, odesílaná data a příznak.

Na straně klienta je vytvořena schránka jako v případě serveru (viz Obrázek 3.17). Této schránce je funkcí `setsockopt()` nastaven časovač a po synchronizaci se serverem je funkcí `send()` s využitím adresní struktury `sockaddr_in()` pro komunikaci se serverem (obsahuje IP adresu a port, na kterém server naslouchá) odeslán požadavek na soubor. Data přijímaná od serveru jsou přijímána funkcí `recv()` a uložena do souboru. Po dokončení přenosu je schránka klienta uzavřena funkcí `close()`.

```
socket(SOCK, PF_INET, SOCK_DGRAM, getprotobyname("udp"));

$serv_sock = sockaddr_in($port, INADDR_ANY);
bind(SOCK, $serv_sock);

while(1){
    #synchronizace
    ...
    setsockopt( SOCK, SOL_SOCKET, SO_RCVTIMEO, pack('L!L', 1, 0) );

    $cl_sock = recv(SOCK,$filename,1024,0) or die "recv(): $!";
    send(SOCK, $data, 0, $cl_sock);
}
```

Obrázek 3.16: Prostředky komunikace UDP serveru v jazyce Perl

```
socket(SOCK, PF_INET, SOCK_DGRAM, getprotobyname("udp"));
setsockopt( SOCK, SOL_SOCKET, SO_RCVTIMEO, pack('L!L', 1, 0) );
#synchronizace
  ...
$serv_sock = sockaddr_in($port, $ip_serveru);

send(SOCK, $filename, 0, $serv_sock);
recv(SOCK,$x,1032,0);
close(SOCK);
```

Obrázek 3.17: Prostředky komunikace UDP klienta v jazyce Perl

Jazyk Python

Pro implementaci UDP komunikace v jazyce Python jsem využil stejně jako v případě TCP modul `socket`. Funkcí `socket()` s parametry `AF_INET` a `SOCK_DGRAM` je vytvořen objekt schránky serveru (viz Obrázek 3.18). Parametry představují protokolovou rodinu a typ schránky. Schránka serveru je pojmenována metodou `bind()`, je jí přiřazen parametrem

metody port a IP adresa. Po synchronizaci s klientem je schránce nastaven časovač metodou `settimeout()`. Poté je od klienta přijat název souboru metodou `recvfrom()` schránky. Parametrem metody je velikost přijímaných dat. Tato metoda vrací data (zde název souboru) a adresu odesílatele (klienta). Soubor je poté odeslán metodou `sendto()`. Parametry jsou odesílaná data a adresová struktura klienta přijatá s požadavkem.

Na straně klienta je stejně jako v případě serveru vytvořena UDP schránka (viz Obrázek 3.19). Té je také nastaven časovač metodou `settimeout()` schránky. Metodou `sendto()` je serveru odeslán název požadovaného souboru a data od serveru jsou přijímána metodou `recvfrom()`. Přijatá data jsou uložena do souboru. Po ukončení přenosu je schránka klienta uzavřena a klient je ukončen.

```
def main(argv=None):
  s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
  s.bind(("", port))

  while 1:
  #synchronizace
  ...
  s.settimeout(1)

  filename, addr = s.recvfrom(1024);
  s.sendto(struct.pack('i i 1024s', counter, len(data), data), addr)
```

Obrázek 3.18: Prostředky komunikace UDP serveru v jazyce Python

```
def main(argv=None):
  s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
  s.settimeout(1)

  s.sendto(filename, 0,addr)
  data, addr = s.recvfrom(1032);

  s.close()
```

Obrázek 3.19: Prostředky komunikace UDP klienta v jazyce Python

Jazyk Java

Prostředky pro UDP komunikaci jsou jako v případě TCP implementovány v balíčku `java.net.*`. Schránka serveru je vytvořena jako instance třídy `DatagramSocket` metodou `new()` s parametrem, kterým je číslo portu, na kterém bude server naslouchat (viz Obrázek 3.20). Této schránce je nastaven časovač metodou `setSoTimeout()`, jejímž parametrem je čas v milisekundách. Dále jsou přijímány požadavky od klientů (dojde k synchronizaci mezi serverem a klientem). Poté je od klienta přijat název požadovaného souboru metodou `receive()` objektu schránky. Parametrem této metody je struktura pro příjem paketu reprezentovaná třídou `DatagramPacket`. Data z požadovaného souboru jsou klientovi odeslána metodou `send()` schránky, jejímž parametrem jsou odesílaná data.

Na straně klienta je vytvořena schránka jako instance třídy `DatagramSocket` (viz Obrázek 3.21). Zde narozdíl od serveru bez parametru. Prostřednictvím této schránky je po synchronizaci se serverem odeslán požadavek na soubor metodou `send()` schránky. Schránce je nastaven časovač pro příjem metodou `setSoTimeout()`. Poté jsou na schránce metodou `receive()` přijímána data od serveru. Po dokončení přenosu je schránka metodou `close()` uzavřena.

```
class server{
    public static void main(String args[]) throws Exception{
        DatagramSocket serverSocket =
            new DatagramSocket(Integer.parseInt(port));

        serverSocket.setSoTimeout(500);

        while (true){
            //synchronizace s klientem
            ...
            DatagramPacket receivePacket =
                new DatagramPacket(receiveData, receiveData.length);

            serverSocket.receive(receivePacket);

            byte[] receiveData = new byte[1024];

            DatagramPacket receivePacket =
                new DatagramPacket(receiveData, receiveData.length);

            serverSocket.receive(receivePacket);
            String filename = new String(receivePacket.getData());

            DatagramPacket sendPacket =
                new DatagramPacket(data, data.length, IPAddress,port);

            serverSocket.send(sendPacket);
        }
    }
}
```

Obrázek 3.20: Prostředky komunikace UDP serveru v jazyce Java

```
class client{
    public static void main(String args[]) throws Exception{
        DatagramSocket clientSocket = new DatagramSocket();
        //synchronizace se serverem
          ...

        sendPacket = new DatagramPacket(filename.getBytes(),
           filenam.length, IPAddress, port);

        receivePacket = new DatagramPacket(data, data.length);
        clientSocket.setSoTimeout(500);

        clientSocket.receive(receivePacket);
        clientSocket.close();}}
```

Obrázek 3.21: Prostředky komunikace UDP klienta v jazyce Java

Jazyk C#

Stejně jako u protokolu TCP se nacházejí prostředky pro implementaci UDP komunikace v jazyce C# ve jmenném prostoru `System.Net.Sockets`. Schránka serveru je vytvořena jako nová instance třídy `Socket()` (viz Obrázek 3.22). Jedná se o datagramovou schránku (parametr `SocketType.Dgram` metody `new()`). Typem protokolu je UDP (`ProtocolType.Udp`). Schránka serveru je pojmenována metodou `Bind()`, jejímž parametrem je adresová struktura reprezentovaná třídou `IPEndPoint` s parametry, kterými jsou adresa (`IPAddress.Any`) a port. Schránce je nastaven časovač přiřazením hodnoty vlastnosti `ReceiveTimeout`. Na schránce je po synchronizaci s klientem přijat požadavek metodou `ReceiveFrom()` schránky. Parametry metody jsou vyrovnávací paměť pro data a struktura pro uložení údajů klienta. Data ze souboru jsou odeslána metodou `SendTo()` s parametry, kterými jsou odesílaná data a adresová struktura klienta.

Na straně klienta je také jako v případě serveru vytvořena schránka (`Socket`) (viz Obrázek 3.23), které je nastaven časovač. Na tuto schránku je metodou `Sendto()` odeslán název souboru a poté jsou metodou `ReceiveFrom()` přijímána data od klienta. Po dokončení přenosu je schránka klienta metodou `Close()` uzavřena a činnost klienta je ukončena.

```
public class Server{
    public static void Main(string[] args){
        Socket s = new Socket(AddressFamily.InterNetwork,
             SocketType.Dgram, ProtocolType.Udp);
        s.Bind(new IPEndPoint(IPAddress.Any, Convert.ToInt32(port)));
        s.ReceiveTimeout = 500;
        while (true){
            //synchronizace s klientem
            ...
            s.ReceiveFrom(data, ref host);
            s.SendTo(data, host);}}}
```

Obrázek 3.22: Prostředky komunikace UDP serveru v jazyce C#

```
class client{
    public static void Main(string[] args){
        Socket s = new Socket(AddressFamily.InterNetwork,
            SocketType.Dgram, ProtocolType.Udp);
        s.ReceiveTimeout = 500;

        IPEndPoint serverEndPoint = new IPEndPoint
          (Dns.GetHostEntry(address).AddressList[1], Convert.ToInt32(port));
         //synchronizace se serverem
          ...
        s.SendTo(System.Text.Encoding.ASCII.GetBytes(filename),
               serverEndPoint);
        s.ReceiveFrom(data, ref host);
        s.Close();}}
```

Obrázek 3.23: Prostředky komunikace UDP klienta v jazyce C#

3.4 Protokol SCTP

Existují dva typy implementace SCTP komunikace [17]. Jedním je komunikace typu „One-to-One“ (podobný styl jako u TCP), kde konkrétní schránka serveru obsluhuje jen jednoho klienta. Druhým typem je „One-to-Many“ (podobné UDP), kde schránka serveru obsluhuje více klientů. Pro vzorovou implementaci jsem si zvolil „One-to-One“ metodu implementace. Zároveň existují dva typy rozhraní pro implementaci této komunikace [19]. Rozdíly v těchto implementacích představuje především implementace funkcí `bind()` a `connect()`, které v této základní verzi umožňují připojení jen jedné adresy ke schránce. K těmto funkcím existují alternativy, kterými jsou funkce `bindx()` a `connectx()`, které umožňují připojení více adres na schránku tzv. multihoming. Funkce `bindx()` zároveň umožňuje měnit adresy v rámci asociace a to pomocí příznaků `SCTP_BINDX_ADD_ADDR` pro přidání adres a `SCTP_BINDX_REM_ADDR` pro odebrání zadaných adres.

Protokol SCTP (viz Obrázek 3.24) je protokolem transportní vrstvy a proto stejně jako protokoly TCP a UDP komunikuje prostřednictvím schránek, které jsou vytvořeny na straně serveru i klienta funkcí `socket()`. Schránka serveru je pojmenována – `bind()`. Na straně serveru i klienta lze pomocí zprávy `SCTP_INIT` nastavit počet proudů, které daná strana vyžaduje v rámci asociace. Server poté na zadané adrese začne naslouchat – `listen()`. Pokud se klient chce připojit k serveru, provede to funkcí `connect()`. Server toto spojení přijme funkcí `accept()`. Tímto krokem vznikne mezi serverem a klientem tzv. asociace. Po navázání asociace zažádá klient server o soubor odesláním jeho názvu – `sctp_sendmsg(filename)`. Server tuto žádost přijme – `sctp_recvmsg(filename)`. Poté tento soubor začne číst (pokud k němu má přístup) a odesílá ho klientovi `sctp_sendmsg(data)`. Klient tato data přijímá funkcí `sctp_recvmsg(data)` a ukládá do souboru. Komunikace je ukončena uzavřením schránky funkcí `close()`. Tato funkce vyvolá ukončení asociace (zaslání zpráv SHUTDOWN mezi oběma koncovými body komunikace).

Druhým typem komunikace je ”One-to-Many”. Tato komunikace se podobá komunikaci UDP. Z toho vyplývá, že z předešlého popisu komunikace nebudou použity funkce `listen()`, `accept()` a `connect()` jako v případě implementace komunikace protokolem UDP.

Obdobně jako u protokolu TCP a UDP, lze i zde implementovat konkurentní server.

Jazyk/funkce	socket()	bind()	listen()	accept()	close()
C/C++	socket()	bind()	listen()	accept()	close()
Perl	socket()	bind()	listen()	accept()	close()
Python	sctp.sctpsocket_tcp()	bind()	listen()	accept()	close()

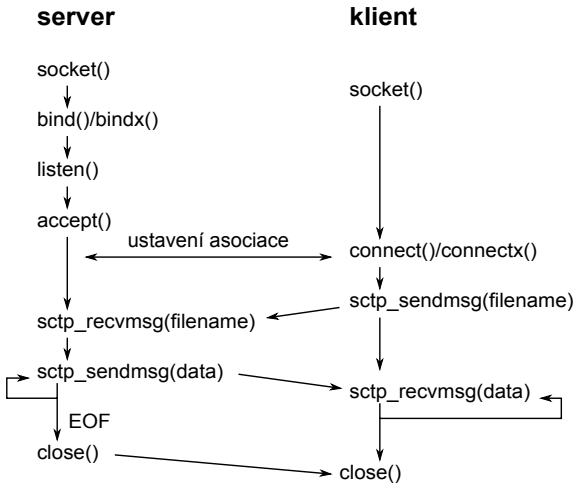
Tabulka 3.6: Funkce a metody použité pro implementaci SCTP serveru

Jazyk/funkce	socket()	connect()	close()
C/C++	socket()	connect()	close()
Perl	socket()	connect()	close()
Python	sctp.sctpsocket_tcp()	connect()	close()

Tabulka 3.7: Funkce a metody použité pro implementaci SCTP klienta

Jazyk/funkce	sctp_recvmsg(data)	sctp_sendmsg()
C/C++	sctp_recvmsg()	sctp_sendmsg()
Perl	recv()	send()
Python	recv()	sctp_send()

Tabulka 3.8: Funkce a metody použité pro implementaci přenosu dat protokolem SCTP



Obrázek 3.24: Implementace SCTP komunikace

Jazyk C/C++

K implementaci SCTP komunikace v jazyce C jsem použil stejné knihovny jako v případě protokolů TCP a UDP. Přibyla knihovna `libsctp` a hlavičkový soubor `<netinet/sctp.h>`, obstarávající prostředky určené pro SCTP implementaci.

Na straně serveru jsem vytvořil schránku funkcí `socket()` (viz Obrázek 3.25). Podle parametů je schránka typu `SOCK_STREAM` a jejím protokolem je SCTP (`IPPROTO_SCTP`). Schránka je stejně jako v případě TCP a UDP pojmenována funkcí `bind()`, jíž je parametrem předána adresní struktura schránky `sockaddr`. Funkcí `setsockopt()` je pomocí volby `SCTP_INITMSG` nastavena struktura inicializační zprávy (`struck sctp_initmsg`), která slouží k nastavení počtu proudů pro komunikaci v rámci asociace. Funkcí `accept()` je přijato spojení ze strany klienta. S klientem je komunikováno prostřednictvím funkce `sctp_recvmsg()` s parametry, jimiž jsou deskriptor schránky, vyrovnávací paměť pro přijímaná data s její velikostí, struktura `sctp_sndrcvinfo` pro řídíci informace a příznaky. Data jsou odesílána funkcí `sctp_sendmsg()`. Po dokončení přenosu je schránka uzavřena fukcí `close()`. Tímto dojde k odeslání zprávy `SHUTDOWN` a je ukončena asociace mezi komunikujícími stranami.

Na straně klienta je vytvořena schránka stejným způsobem, jako v případě serveru (viz Obrázek 3.26). Taktéž je jí nastavena funkcí `setsockopt()` inicializační zpráva. Schránka je připojena funkcí `connect()`, které je předána jako parametr adresní struktura schránky pro připojení k serveru. Komunikace probíhá obdobně jako v případě serveru pomocí funkcí `sctp_sendmsg()` a `sctp_recvmsg()`. Po dokončení přenosu je schránka zavřena funkcí `close()`.

```
int main(int argc, char **argv){
    int sockfd = socket(AF_INET, SOCK_STREAM, IPPROTO_SCTP);
    .... //vyplneni struktury servaddr
    bind(sockfd, (struct sockaddr *) &servaddr, sizeof(servaddr));

    struct sctp_initmsg initmsg;  //nastaveni I/O streamu
    ... //nastaveni poctu streamu ve strukture initmsg
    setsockopt(sockfd, IPPROTO_SCTP, SCTP_INITMSG,
                    &initmsg, sizeof(initmsg));
    listen(sockfd, 5);

    while (1){
        connfd = accept(sockfd,  (struct sockaddr *) &cliaddr, &clilen)

        sctp_recvmsg(sockfd, filename, sizeof(filename),
                     (struct sockaddr *)NULL, 0, &sndrcvinfo, &flags);
        sctp_sendmsg( sockfd, temp, x,
                          NULL, 0, 0, 0, 0, 0, 0 ;
        close(connfd);
    }
}
```

Obrázek 3.25: Prostředky komunikace SCTP serveru v jazyce C

```
int main(int argc, char **argv){
    int sockfd = socket(AF_INET,SOCK_STREAM, IPPROTO_SCTP);
    struct sctp_initmsg initmsg;
    ... //nastaveni poctu streamu ve strukture initmsg
    setsockopt(sockfd, IPPROTO_SCTP, SCTP_INITMSG,
                    &initmsg, sizeof(initmsg));
    connect(sockfd, (struct sockaddr *) &servaddr, sizeof(servaddr));

    sctp_sendmsg(sockfd, filename, strlen(filename),
                      (struct sockaddr *)NULL, 0, 0, 0, 0, 0, 0);
    sctp_recvmsg(sockfd, (void *)temp, sizeof(temp),
                      (struct sockaddr *)NULL, 0, &sndrcvinfo, &flags);
   close(sockfd);}
```

Obrázek 3.26: Prostředky komunikace SCTP klienta v jazyce C

Jazyk Perl

Při implementaci protokolu SCTP lze v jazyce Perl využít modulu `IO::Socket::INET`, který obsahuje prostředky pro implementaci SCTP komunikace. Na straně serveru je vytvořena schránka funkcí `socket()` (viz Obrázek 3.27). Tato schránka je typu `SOCK_STREAM` a je jí nastaven protokol `sctp`. Schránka je pojmenována funkcí `bind()`, která má jako parametry deskriptor schránky a adresní strukturu schránky `sockaddr_in`. Tato struktura obsahuje adresu serveru a port, na kterém naslouchá (`listen()` s parametry deskriptor schránky a délka fronty čekajících klientů). Klient je přijat funkcí `accept()` volané s parametry, kterými jsou deskriptor schránky pro komunikaci s klientem a schránky, na které server naslouchá. Komunikace je realizována funkcemi `recv()` (pro příjem dat) a `send()` (pro odeslání dat). Asociace mezi serverem a klientem je ukončena funkcí `shutdown()`. Parametry této funkce jsou deskriptor schránky a číslo reprezentující způsob ukončení (v tomto případě 2 – okamžité ukončení). Na straně klienta je vytvořena schránka stejným způsoben, jako na straně serveru (viz Obrázek 3.28). Tato schránka je k serveru připojena funkcí `connect()`, jejímž parametrem je adresní struktura schránky `sockaddr_in`. Komunikace je opět realizována funkcemi `recv()` a `send()`. Po dokončení přenosu je schánka klienta uzavřena funkcí `shutdown()` a jeho činnost je ukončena.

```
socket(SOCK, PF_INET, SOCK_STREAM, getprotobyname("sctp"));
$serv_sock = sockaddr_in($port, INADDR_ANY);
bind(SOCK, $serv_sock);
listen(SOCK, 10);

while(1){
    accept ($client,SOCK);
    recv($client, $filename, 1024, 0);
    send ($connfd,$data,0);
    shutdown($connfd, 2);
}
```

Obrázek 3.27: Prostředky komunikace SCTP serveru v jazyce Perl

```
socket(SOCK, PF_INET, SOCK_STREAM, getprotobyname("sctp"));
$serv_sock = sockaddr_in($port, $ip);
connect(SOCK,  $serv_sock);

send (SOCK,$filename,0);
recv(SOCK, $data, 1024, 0);
shutdown(SOCK, 2);
```

Obrázek 3.28: Prostředky komunikace SCTP klienta v jazyce Perl

Jazyk Python

V případě SCTP komunikace v jazyce Python jsem využil modul `sctp`, který poskytuje plnou funkcionalitu pro implementaci. Základem komunikace je objekt `sctpsocket_tcp()` (případně `sctpsocket_udp()`) z modulu `sctp` (viz Obrázek 3.29). Navázání asociace v jazyce Python probíhá pomocí funkcí obdobných jako při TCP komunikaci. Na straně serveru je to metoda `bind()`. K přijetí klienta dochází metodou `accept()`, která vrací schránku pro obsluhu klienta. Pro potřeby odesílání a příjmu dat jsem využil funkcí `sctp_send()` a `recv()`. I zde lze pro nastavení počtu vstupně/výstupních proudů požadovaných daným koncovým bodem komunikace využít `initmsg()`. Na straně klienta (viz Obrázek 3.30) je vytvořená schránka připojena k serveru metodou `connect()`, mající jako parametry adresu serveru a port na kterém naslouchá. Data jsou odesílána metodou `sctp_send()` a přijímána metodou `recv()` schránky. Po dokončení přenosu je schránka klienta uzavřena metodou `close()`.

```
def main(argv=None):
  s = sctp.sctpsocket_tcp(socket.AF_INET)
  s.bind(("", port))
  s.listen(1)

  while 1:
    conn, addr = s.accept() #pripojeni klienta
    filename = conn.sctp_recv(1024)

    conn.sctp_send(data)
    conn.close()
```

Obrázek 3.29: Prostředky komunikace SCTP serveru v jazyce Python

```
def main(argv=None):
  s = sctp.sctpsocket_tcp(socket.AF_INET)
  s.connect((addr, port))

  s.sctp_send(filename)
  data = s.recv(1024)
  s.close()
```

Obrázek 3.30: Prostředky komunikace SCTP klienta v jazyce Python

3.5 Protokol IP

V případě implementace komunikace protokolu IP používám schránky typu raw určené ke komunikaci na síťové vrstvě. Raw schránky zde využívám k odesílání dat. Na druhé straně komunikace (příjemce dat) využívám k odchytávání paketů knihovnu `libpcap`. Tato knihovna slouží k odchytávání komunikace na zadáném síťovém rozhraní. Při odesílání dat přes raw schránku protokolem IP je nejdříve potřeba vytvořit samotnou schránku – `socket()`. Typ schránky je `SOCK_RAW` a typ protokolu je nastaven na `IPPROTO_RAW`. Poté je vytvořena IP hlavička, jejíž položky je třeba vyplnit. Za takto vytvořenou hlavičku jsou poté připojena přenášená data. Tato data (s připojenou hlavičkou) jsou přes raw schránku odeslána příjemci (`sendto()`). Na straně příjemce jsou tato data odchytávána na zvoleném rozhraní. Pro příjem dat je nejprve použita funkce `pcap_open_live()`, která vytvoří sezení pro odchytávání paketů na zadaném rozhraní. Pakety lze poté odchytávat různými způsoby. Buď v cyklu voláním funkce `pcap_next()`, která přijme další paket. Nebo funkcí `pcap_loop()`, která postupně do vypršení časovače, odměřujícího dobu mezi příchody paketů, odchytává pakety a předává je zadané funkci, která řeší jejich zpracování. V případě implementované komunikace je zpracováním myšleno získání dat z paketu a jejich další zpracování. V případě příjmu názvu souboru, který má být přenášen jsou data názvem tohoto souboru a tento soubor je otevřen pro čtení. V případě, že jsou v paketu data přenášeného souboru, jsou tato data na straně klienta uložena do cílového souboru.

Jazyk C/C++

V jazyce C je vytvořena raw schránka s nastaveným typem protokolu `IPPROTO_RAW` (viz Obrázek 3.31). Této schránce je nastavena volba IP hlavičky funkcí `setsockopt()`, které je předána jako parametr volba `IP_HDRINCL`. Tato hlavička je vytvořena pomocí struktury `struct ip*`. Struktura se nachází v hlavičkovém souboru `<netinet/ip.h>`. Tato struktura je vyplněna včetně protokolu vyšší vrstvy. Jelikož komunikace probíhá jen na síťové vrstvě byl zadán nedefinovaný protokol. Za takto vytvořenou hlavičku jsou funkcí `memcpy()` připojena přenášená data. Datová struktura vytvořená tímto způsobem je odeslána funkcí `sendto()`. Na straně příjemce je komunikace zachytávána pomocí knihovny libpcap (`<pcap/pcap.h>`). Nejdříve je zvoleno rozhraní, na kterém bude zachytávána síťová komunikace funkcí `pcap_findalldevs()`, které načte seznam dostupných rozhraní, ze kterých je zvoleno rozhraní pro komunikaci (`char *dev`). Toto rozhraní je otevřeno funkcí `pcap_open_live()`. Na otevřeném rozhraní jsou pakety zachytávány funkcí `pcap_next()`. Zachycené pakety jsou dále zpracovány (jsou z nich vybrána data pro další zpracování).

Jazyk Perl

K odesílání dat v jazyce Perl jsem využil modulu `Net::RawIP` (viz Obrázek 3.32), který slouží ke tvorbě raw schránek. Tyto schránky jsou vytvořeny jako nová instance třídy `Net::RawIP` (metoda `new()`), přičemž je součástí konstruktoru i vytvoření IP hlavičky pomocí položky `ip{}` a připojení dat. Tato data jsou odeslána pomocí metody `send()`. Pro příjem dat je využito modulu `Net::Pcap`. Pomocí `Net::Pcap::findalldevs()` (vrací seznam dostupných zařízení) je nalezeno rozhraní pro odchytávání paketů. Na zvoleném rozhraní je poté otevřeno sezení (`Net::Pcap::open_live()`) pro zachytávání paketů. Pakety jsou zachytávány ve smyčce (do vypršení časovače mezi pakety, který byl nastaven při otevírání sezení) pomocí `Net::Pcap::loop()`. Tato metoda předává zachycené pakety funkci ve

```
int main(int argc, char **argv){
    //odeslani paketu
    char buffer[1044]="";
    struct ip *IPheader = (struct ip *) buffer;
    IPheader->ip_hl = 5;
    //vyplneni IP hlavicky
    ....
     memcpy(((char *)IPheader) + SIZE_IP, body, 0);
    int sockfd = socket(PF_INET, SOCK_RAW, IPPROTO_RAW);
    sendto(sockfd,buffer, IPheader->ip_len,0,
      (struct sockaddr *)&din,sizeof(din));

    //prijem paketu
    struct pcap_pkthdr header;  //hlavicka ziskana pcap
    pcap_if_t *alldevs, *d2;

    pcap_t *pcap_findalldevs(&alldevs, errbuf);

    handle = pcap_open_live(dev, BUFSIZ, 1, 3000, errbuf);

    const u_char *packet = pcap_next(handle, &header); //dalsi packet

    const struct ethernet *ethernet;
    const struct ip *sip;
    ethernet = (struct ethernet*)(packet);
    sip = (struct ip*)(packet + SIZE_ETHERNET);
    data = (char *)(packet + SIZE_ETHERNET + SIZE_IP);}
```

Obrázek 3.31: Odchytávání paketů v jazyce C

svém parametru. Ta získá z paketu ethernetový rámec (`NetPacket::Ethernet::strip()`), z tohoto rámce ip datagram (`NetPacket::IP->decode()`) a z ip datagramu samotná přenášená data (`NetPacket::IP::strip()`). Po ukončení přenosu je sezení uzavřeno metodou `Net::Pcap::close()`, jejímž parametrem je uzavírané sezení.

Jazyk Python

V jazyce Python jsem pro odesílání dat využil raw schránky z modulu `socket` (viz Obrázek 3.33). Typ protokolu schránky je `IPPROTO_RAW`. Pro vytvoření IP hlavičky jsem využil modul `dpkt`. Jedná se o modul pro vytváření paketů, který obsahuje definice datových struktur. K vytvoření IP hlavičky je využit konstruktor `dpkt.ip.IP()`. Položky hlavičky jsou vyplněny zadanými hodnotami. Data určená k přenosu jsou vloženy do položky `data` IP hlavičky. Výsledná datová struktura je převedena na řetězec funkcí `str()` a odeslána metodou `send()` schránky. Na přijímací straně je pro odchytávání paketů využito modulu `pcapy`. Pomocí tohoto modulu je otevřeno sezení (`pcapy.open_live()`) pro zadané rozhraní. Rozhraní je získáno metodou `findalldevs()`, která vrací seznam dostupných rozhraní. Z tohoto rozhraní jsou odchytávány pakety metodou `next()`, která vrací hlavičku a přenášená data (payload).

```
#odeslani dat
$n = Net::RawIP->new({ip  => {version => '4', ...},
                      generic => { data =>$data}})
$n->send;

#prijem dat
@devs = Net::Pcap::findalldevs(\$err) or die 'Net::Pcap::findalldevs();
$object = Net::Pcap::open_live($dev, 1500, 0, -1, \$err);

while(1){
  Net::Pcap::loop($object, 1, \&send_data, '') ;
}
Net::Pcap::close($object);

sub send_data {
    my ($user_data, $header, $packet) = @_;
    my $ether_data = NetPacket::Ethernet::strip($packet);
    my $ip = NetPacket::IP->decode($ether_data);
    $data=$ip->{'data'};}
```

Obrázek 3.32: Odchytávání paketů v jazyce Perl

```
def main(argv=None):
  pcapy.findalldevs()
  cap = pcapy.open_live(dev.strip(), 1500, 1, 0)
  cap.setfilter("ip")

  while 1:
    (header, payload) = cap.next()
    eth=dpkt.ethernet.Ethernet(str(payload))
    ipt = eth.data
    data=ipt.data

  s = socket.socket(socket.AF_INET, socket.SOCK_RAW, socket.IPPROTO_RAW )
  ip = dpkt.ip.IP(src=src_addr, dst=dst_addr, p=143, ...)

  send_ip=struct.unpack("BBBB",dst_addr)
  send_ip_str="%d.%d.%d.%d" %(send_ip[0],send_ip[1],send_ip[2],send_ip[3])

  s.connect((send_ip_str, 1))
  s.send(str(data))
  s.close()
```

Obrázek 3.33: Odchytávání paketů v jazyce Python

Jazyk Java

V programovacím jazyce Java využívám knihovnu `JNetPcap`[6], která poskytuje prostředky pro zachytávání i odesílání paketů (viz Obrázek 3.34). Pro odeslání dat je nejdříve nutné otevřít sezení metodou `openLive()` třídy `Pcap`. Metoda `openLive()` má jako parametr identifikaci síťového rozhraní zvoleného ze seznamu vygenerovaného metodou `findAllDevs()`. Paket je takto otevřeným sezením odeslán metodou `sendPacket()`. Datovou strukturu paketu lze implementovat pomocí třídy `JMemoryPacket`. Z tohoto paketu lze metodou `getHeader()` získat strukturu IP hlavičky (instance třídy `Ip4`) a tu vyplnit. Pakety jsou na otevřeném sezení zachytávány metodou `nextPacket()` třídy `PcapPacketHandler`. Metoda vrací hlavičku paketu a data.

```
public class server{
    public static void main(String[] args) throws IOException{
        StringBuilder errbuf = new StringBuilder();
        List<PcapIf> ifs = new ArrayList<>();
        Pcap.findAllDevs(ifs, errbuf);
        Integer dev = Integer.valueOf(1);
        PcapIf netInterface = ifs.get(dev);

        final Pcap pcap = Pcap.openLive(netInterface.getName(),
             1500, Pcap.MODE_NON_PROMISCUOUS, 3000, errbuf);

        JMemoryPacket packet = new JMemoryPacket(34);
        Ip4 ip4 = packet.getHeader( new Ip4() );
        packet.scan(JProtocol.IP4_ID);

        ip4.ttl(64);
        //vyplneni IP hlavicky
        ...
        pcap.sendPacket(ByteBuffer.wrap( packet.....));

        PcapPacketHandler<String> jpacketHandler =
          new PcapPacketHandler<String>(){
            public void nextPacket(PcapPacket packet, String user){
                //Zpracování paketu
                ......
            }};
        pcap.loop(0, jpacketHandler, "");
    }
}
```

Obrázek 3.34: Odchytávání paketů v jazyce Java

[6] http://jnetpcap.com

Jazyk C#

V jazyce C# jsem pro implementaci komunikace protokolem IP využil knihovny `PcapDotNet`[7]. Tato knihovna obsahuje třídy využitelné jak pro tvorbu a odesílání paketů, tak pro jejich odchytávání na síťovém rozhraní a následné dekódování. Síťové rozhraní je zde identifikováno třídou `PacketDevice` (získáno jako položka seznamu rozhraní získaných metodou `AllLocalMachine` třídy `LivePacketDevice` (viz Obrázek 3.35). Na tomto rozhraní je otevřeno sezení reprezentované třídou `PacketCommunicator`. Toto sezení je otevřeno metodou `Open()`. Pomocí třídy `IpV4Layer` je vytvořena IP hlavička. Pomocí třídy `PayloadLayer` je vytvořena datová vrstva, které jsou předána data určená k přenosu. Všechny výše popsané prvky jsou poté zabaleny pomocí třídy `PacketBuilder` a vloženy do paketu (identifikován třídou `Packet`). Tento paket je odeslán metodou `SendPacket()` třídy `PacketCommunicator`. Na přijímací straně komunikace je otevřeno sezení metodou `Open()` třídy `PacketDevice`. Sezení je instance třídy `PacketCommunicator`. Pomocí metody `ReceivePacket`, která je metodou třídy `PacketCommunicator` jsou na daném rozhraní odchytávány pakety. Tato metoda vrací jako výsledek `PacketCommunicatorReceiveResult`. Pomocí tohoto výsledku je určeno, zda byl paket přijat a může dojít k jeho zpracování. V průběhu zpracování jsem z paketu odstranil data hlavičky. Výsledkem této operace jsou přenášená data, která jsou poté uložena do souboru.

3.6 Komunikace na linkové vrstvě

Pro implementaci komunikace na linkové vrstvě využívám prostředky jednotlivých jazyků pro vytváření paketů na straně odesílatele a pro jejich zachytávání na straně příjemce.

Na straně serveru jsou zachytávány pakety od klientů. Jedná se o požadavky na soubor. Po zachycení požadavku je tento požadavek vyřízen. Je otevřen soubor, ke čteným datům je přidána ethernetová hlavička a takto vytvořený rámec je odeslán na zvolené síťové rozhraní. Na straně klienta jsou po odeslání požadavku (názvu souboru) zachytávány pakety na zvoleném síťovém rozhraní. Přijaté pakety jsou dekódovány a data v nich obsažená jsou uložena do souboru.

Jazyk C/C++

Příjem paketů probíhá stejně jako v případě protokolu IP pomocí knihovny libpcap (hlavičkový soubor `pcap.h`). Pro odesílání paketů je využito knihovny libnet. Nejdříve je vytvořen takzvaný libnet kontext funkcí `libnet_init()` na zvoleném rozhraní. Ten slouží k odesílání paketů na toto rozhraní. Poté je vytvořena ethernetová hlavička. Toho je dosaženo funkcí `libnet_build_ethernet()`, jejímiž parametry jsou MAC adresy (zdrojová a cílová), typ, ukazatel na přenášená data, velikost přenášených dat a libnet kontext (rozhraní na které bude rámec odeslán). Takto vytvořený rámec je odeslán funkcí `libnet_write()`. Po dokončení odesílání je kontext uzavřen funkcí `libnet_destroy()`.

Jazyk Perl

V komunikaci na linkové vrstvě jsem v jazyce Perl využil modul `Net::Pcap`. Ten slouží stejně jako v případě protokolu IP k odchytávání paketů na síťovám rozhraní (`Net::Pcap::loop()`). Zde jsem prostředků tohoto modulu využil i pro odesílání dat. Data jsem nejprve zabalil

[7] http://pcapdotnet.codeplex.com/

```
static void Main(string[] args){
    IList<LivePacketDevice> allDevices =
        LivePacketDevice.AllLocalMachine;
    PacketDevice selectedDevice = allDevices[deviceIndex];

    using (PacketCommunicator communicator = selectedDevice.Open(65536,
            PacketDeviceOpenAttributes.None,3000)){
        EthernetLayer eth = new EthernetLayer {};
        IpV4Layer ipV4 = new IpV4Layer{
            Source = new IpV4Address(address), ...}

        PayloadLayer data = new PayloadLayer(){
            Data = new Datagram(data),};

        PacketBuilder builder = new PacketBuilder(eth, ipV4, data);

        Packet packet = builder.Build(DateTime.Now);
        communicator.SendPacket(packet);

        //prijem paketu
        PacketCommunicatorReceiveResult result =
            communicator.ReceivePacket(out packet2);
        switch (result){
        case PacketCommunicatorReceiveResult.Timeout:
            continue;
        case PacketCommunicatorReceiveResult.Ok:
            /*zpracovani paketu*/
            break;
        default:}}}}
```

Obrázek 3.35: Odchytávání paketů v jazyce C#

funkcí `pack()`, které jsem předal jako parametry položky ethernetové hlavičky včetně jejich formátování. Takto zabalená data jsem poslal na rozhraní, které bylo otevřené funkcí `Net::Pcap::open_live()`. To jsem provedl funkcí `Net::Pcap::pcap_sendpacket()`, jejímiž parametry jsou identifikátor otevřeného sezení a odesílaná data.

Jazyk Python

V jazyce Python jsem k odchytávání dat využil modul `pcapy`. Odchytávání zde probíhá stejně, jako v případě protokolu IP. Data odesílám prostřednictvím raw schránky vytvořené pomocí třídy `socket` a parametrem `SOCK_RAW`, která je připojena na rozhraní (`bind((dev.strip(), eth_type))`). Na tuto schránku je metodou `send()` odeslán rámec vytvořený pomocí modulu `dpkt` (`dpkt.ethernet.Ethernet(src=src_addr, dst=dst_addr, type=eth_type)`). Po odeslání dat je schránka uzavřena metodou `close()`.

Jazyk Java

Stejně jako u protokolu IP je i zde využito knihovny `JNetPcap` pro odchytávání a odesílání paketů na síťové rozhraní. Pro odesílaná dat je využito instance třídy `JMemoryPacket`, která je naplněna daty. Tato data jsou reprezentována třídou `Ethernet`, obsahující položky ethernetové hlavičky. Za hlavičku jsou připojena přenášená data. Vše je odesláno metodou `sendPacket()` třídy `Pcap` knihovny `JNetPcap`.

Jazyk C#

Jako v ostatních jazycích je i v případě jazyka C# využito stejné knihovny, jako v případě protokolu IP. Jedná se o knihovnu `PcapDotNet`. Rozdíl od implementace protokolu IP je především v jiném struktuře odesílaných dat (komunikace na jiné vrstvě). Zde se využívá třídy `EthernetLayer` pro reprezentaci ethernetové hlavičky a `PayloadLayer` pro reprezentaci přenášených dat. Jinak komunikace probíhá stejně jako v případě protokolu IP prostřednictvím instance třídy `PacketCommunicator`.

Shrnutí

V rámci práce jsem implementoval ukázky serverů a klientů na výše zmíněných protokolech a v daných jazycích (viz Obrázek 3.9). Všechny servery mohou být spuštěny jako iterativní nebo konkurentní.

jazyk	implementace serveru	implementace klienta
C/C++	TCP, UDP, SCTP, IP, Ethernet	TCP, UDP, SCTP, IP, Ethernet
Perl	TCP, UDP, SCTP, IP, Ethernet	TCP, UDP, SCTP, IP, Ethernet
Python	TCP, UDP, SCTP, IP, Ethernet	TCP, UDP, SCTP, IP, Ethernet
Java	TCP, UDP, IP, Ethernet	TCP, UDP, IP, Ethernet
C#	TCP, UDP, IP, Ethernet	TCP, UDP, IP, Ethernet

Tabulka 3.9: Přehled implementované komunikace

Kapitola 4

Testování

V rámci této bakalářské práce je účelem testování ověření správnosti a spolehlivosti implementované síťové komunikace. Zabýval jsem se především spolehlivostí komunikace a také správností implementace konkurentního serveru.

4.1 Překlad a použité knihovny

Jazyk C

V jazyce C jsou ke všem implementacím přiloženy soubory Makefile pro překlad. Překlad probíhá prostřednictvím překladače g++ (použitá verze 4.4).
Použité knihovny:

- libc
- libsctp (ver. 1.0.11) – implementace protokolu SCTP
- libpcap (ver. 1.1.1) a libnet (ver. 1.1.4) – implementace na síťové a linkové vrstvě

Jazyk Perl

Implementace v jazyce Perl lze spustit prostřednictvím interpretu jazyka Perl (použitá verze 5.10.1).
Použité moduly:

- IO::Socket::Net – implementace protokolů TCP, UDP, SCTP
- Net::RawIP (ver. 0.25) – implementace raw schránek
- Net::Pcap (ver. 0.16) – implementace odchytávání paketů
- NetPacket::Ethernet a NetPacket::IP (ver. 1.3.1) – dekódování paketů

Jazyk Python

Implementace v jazyce Python jsou spouštěny prostřednictvím jeho interpretu (použitá verze 2.6.6).
Použité moduly:

- socket – implementace schránek

- sctp[1] – implementace sctp komunikace (wraper libsctp)
- pcapy (ver. 0.10.6) – implementace odchytávání paketů
- dpkt – dekódování a vytváření hlaviček paketů

Jazyk Java

Pro implementaci komunikace v jazyce Java bylo využito Java JDK ve verzi 7u3[2]. Dále bylo využito pro implementaci komunikace na síťové a linkové vrstvě knihovny JNetPcap[3] (ver. 1.3.0).

Implementace komunikace na transportní vrstvě je překládána pomocí javac (parametrem je zdrojový soubor). Pro implementace na síťové a linkové vrstvě je při překladu i spuštění přidán odkaz na knihovnu JNetPcap (javac -cp .;jnetpcap.jar soubor.java)

Jazyk C#

Pro implementaci komunikace v jazyce C# byl využit Microsoft .NET Framework 4[4]. Překlad aplikací probíhá pomocí překladače C# (csc.exe) – csc soubor.cs.

Pro implementaci komunikace na síťové a linkové vrstvě byla přidána knihovna PcapDotNet[5] (ver. 0.10.0). Při překladu musí být uvedena reference na knihovny obsažené v PcapDotNet (csc /reference:(PcapDotNet knihovny) soubor.cs).
PcapDotNet knihovnami jsou:

- PcapDotNet.Analysis.dll
- PcapDotNet.Base.dll
- PcapDotNet.Core.dll
- PcapDotNet.Core.Extensions.dll
- PcapDotNet.Packets.dll

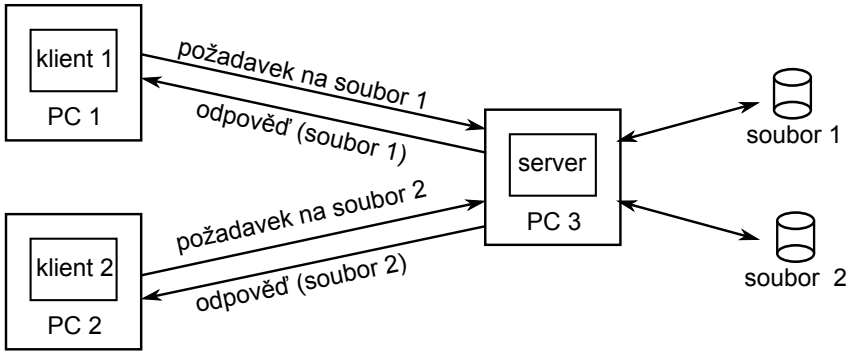
4.2 Testování implementované komunikace

Během testování jsem se zabýval spolehlivostí komunikace (přenosu dat). Toto testování bylo realizováno přenosem různých souborů různé velikosti (účelem testu bylo určit spolehlivost přenosu různých souborů). Jednalo se o tři soubory.

- soubor 1 – binární soubor (9290 B)
- soubor 2 – pdf soubor (94022 B)
- soubor 3 – binární soubor (přibližně 21,7 MB)

[1] https://github.com/philpraxis/pysctp
[2] http://www.oracle.com/technetwork/java/javase/downloads/jdk-7u3-download-1501626.html
[3] http://jnetpcap.com
[4] http://www.microsoft.com/net/
[5] http://pcapdotnet.codeplex.com/

Dalším testem bylo ověření správné funkčnosti implementace konkurentního serveru. Toho bylo dosaženo „souběžným“ spuštěním dvou klientů s různými požadavky (klienti požadovali různé soubory). Nejprve byl spuštěn klient požadující od serveru „větší“ soubor a poté klient požadující „menší“ soubor. Pokud byla konkurentnost serveru implementována správně, byl dříve přenesen menší soubor, tedy soubor, na který přišel požadavek později. Pro větší názornost tohoto testu bylo vloženo do zpracování serveru čekání danými prostředky jednotlivých jazyků (obvykle se jednalo o funkci sleep() nebo podobnou funkci/metodu). Tímto čekáním bylo dosaženo zpomalení činnosti serveru a zvětšení rozdílu výsledných časů přenosu jednotlivých klientů.



Obrázek 4.1: Testování komunikace

Test komunikace na transportní vrstvě v prostředí Unixu

První částí bylo testování komunikace na transportní vrstvě v prostředí Unixu. Toto testování bylo provedeno na serverech merlin.fit.vutbr.cz (CentOS 5.5) a eva.fit.vutbr.cz(FreeBSD 9.0).

V testovaných případech byl server spuštěn na serveru merlin.fit.vutbr.cz a klient na eva.fit.vutbr.cz.
Spuštění aplikace pro protokoly TCP a UDP v prostředí Unixu:

- Jazyk C/C++:
  - ./server -p=12345 (-i)
  - ./client -a=merlin.fit.vutbr.cz -p=12345 -f=nazev_souboru
- Jazyk Perl:
  - perl server.pl -p=12345 (-i)
  - perl client.pl -a=merlin.fit.vutbr.cz -p=12345 -f=nazev_souboru
- Jazyk Python:
  - python server.py -p=12345 (-i)
  - python client.py -a=merlin.fit.vutbr.cz -p=12345 -f=nazev_souboru

Komunikace pomocí protokolu SCTP byla testována na systému Debian 6 z důvodu podpory protokolu na serveru merlin.fit.vutbr.cz (s dodanou knihovnou lze přeložit). Aplikace byly spouštěny se stejnými parametry jako v případě testování komunikace protokolů TCP a UDP (s vyjímkou adresy serveru).

Jazyk/test	soubor 1	soubor 2	soubor 3	test konkurentnosti
C/C++	ano	ano	ano	ano
Perl	ano	ano	ano	ano
Python	ano	ano	ano	ano

Tabulka 4.1: Testování TCP, UDP a SCTP komunikace v Unixu

Test komunikace na transportní vrstvě v prostředí Windows

Komunikace byla testována na operačním systému Windows XP Pro SP3 a Windows 7. Spuštění aplikace pro protokoly TCP a UDP v prostředí Windows:

- Jazyk Java:
  - java server -p=12345 (-i)
  - java client -a=adresa_serveru -p=12345 -f=nazev_souboru
- Jazyk C#:
  - server -p=12345 (-i)
  - client -a=adresa_serveru -p=12345 -f=nazev_souboru

Jazyk/test	soubor 1	soubor 2	soubor 3	test konkurentnosti
Java	ano	ano	ano	ano
C#	ano	ano	ano	ano

Tabulka 4.2: Testování TCP a UDP komunikace ve Windows

Protokol SCTP jsem v prostředí Windows neimplementoval. Důvodem byla neoficiální podpora protokolu a nefunkčnost nalezených knihoven třetí strany.

Test komunikace na síťové a linkové vrstvě v prostředí Unixu

Testování komunikace na síťové a linkové vrstvě jsem provedl na operačním systému Debian 6 z důvodu nutnosti administrátorského oprávnění pro přistupování k síťovému rozhraní.

Spuštění aplikace pro síťovou a linkovou vrstvu v prostředí Unixu:

- Jazyk C/C++:
  - ./server (-i)
  - ./client -a=adresa_serveru -f=nazev_souboru
- Jazyk Perl:
  - perl server.pl (-i)
  - perl client.pl -a=adresa_serveru -f=nazev_souboru
- Jazyk Python:

– python server.py (-i)
– python client.py -a=adresa_serveru -f=nazev_souboru

Adresa serveru je v případě komunikace na síťové vrstvě IP adresa serveru a v případě komunikace na linkové vrstvě je to MAC adresa serveru.

Jazyk/test	soubor 1	soubor 2	soubor 3	test konkurentnosti
C/C++	ano	ano	ano	ano
Perl	ano	ano	ano	ano
Python	ano	ano	ano	ano

Tabulka 4.3: Testování komunikace na síťové a linkové vrstvě v Unixu

Test komunikace na síťové a linkové vrstvě v prostředí Windows

Testování implementací komunikace v jazycích Java a C# na síťové a linkové vrstvě jsem provedl na počítačích s operačním systémem Windows 7 a Windows XP SP3. Komunikace byla testována obousměrně (server spuštěn na Windows 7 i XP).
Spuštění aplikace pro síťovou a linkovou vrstvu v prostředí Windows:

- Jazyk Java:
  – java -cp .;jnetpcap.jar server (-i)
  – java -cp .;jnetpcap.jar client -a=adresa_serveru -f=nazev_souboru
- Jazyk C#:
  – server (-i)
  – client -a=adresa_serveru -f=nazev_souboru

Adresou serveru je zde v případě komunikace na síťové vrstvě IP adresa serveru a v případě komunikace na linkové vrstvě MAC adresa serveru.

Jazyk/test	soubor 1	soubor 2	soubor 3	test konkurentnosti
Java	ano	ano	ano	ano
C#	ano	ano	ano	ano

Tabulka 4.4: Testování komunikace na síťové a linkové vrstvě ve Windows

Křížové testy

Kromě testování komunikace v rámci jednotlivých jazyků jsem otestoval i komunikaci mezi klientem a serverem v různých jazycích. Nejdříve v prostředí Unixu (viz Tabulka 4.5). Poté jsem testoval komunikaci mezi prostředími Windows a Unix (viz Tabulka 4.6).

Dále jsem otestoval komunikaci protokolu IP mezi implementací v jazyce Java (ve Windows) a v jazycích C, Perl, Python (v Unixu – Debian 6). V tomto testu jsem nejprve spustil ve Windows Java server a postupně jsem odesílal požadavky Unixových klientů (Debian). Poté jsem testoval komunikaci druhým směrem (server na Debianu a klient na Windows – Java).

Server	Klient
C	Perl, Python
Perl	C, Python
Python	C, Perl

Tabulka 4.5: Testování komunikace na transportní vrstvě mezi jednotlivými jazyky v prostředí Unixu (servery merlin.fit.vutbr.cz a eva.fit.vutbr.cz)

Server	Klient
Java	C#, C, Perl, Python
C#	Java, C, Perl, Python
C	C#, Java
Perl	C#, Java
Python	C#, Java

Tabulka 4.6: Testování komunikace na linkové vrstvě

4.3 Omezení

V rámci implementace jazyka a protokolu jsem úspěšně otestoval všechny implementované aplikace. Implementace komunikace na transportní vrstvě jsem v prostředí Unixu testoval spuštěním serveru (postupně v každém jazyce) a připojováním klientů (C, Perl, Python). Podobně jsem tuto komunikaci testoval v prostředí Windows.

U imlementace komunikace protokolem IP jsem testoval komunikaci mezi operačními systémy (Windows a Debian 6). V tomto případě implementace v jazyce C# nekomunikuje s ostatními. Je to způsobeno tím, že jsem u implementace protokolu v položce IP hlavičky, kterou je protokol vyšší vrstvy, zvolil nedefinovaný protokol (jedná se implementaci komunikace protokolu IP, tudíž na vyšší vrstvě se nenachází komunikační protokol, ale data). Zvolená knihovny pro jazyk C# (PcapDotNet) tento prototol nepodporuje, tudíž zde byl zvolen jiný.

4.4 Shrnutí

V rámci testování implementací jsem nejprve otestoval implementace v rámci jazyků, ve kterých byly implementovány a poté i napříč jazyky.

Protokoly TCP a UDP v prostředí Unixu jsem testoval na serverech merlin.fit.vutbr.cz a eva.fit.vutbr.cz (server se nacházel na serveru merlin). Implementace prokolu SCTP a komunikace na síťové a linkové vrstvě v prostředí Unixu jsem z důvodu podpory a nutnosti administrátorského oprávnění testoval na operačním systému Debian 6. Testování bylo provedeno s výše uvedenými knihovnami.

Implementace v prostředí Windows jsem testoval na Windows XP (SP3) a Windows 7 (64 bit). V obou systémech jsem doinstaloval .Net Framowork 4 (pro jazyk C#) a Java JDK 7 (pro jazyk Java). U komunikace na síťové a linkové vrstvě jsem využil výše zmíněných knihoven.

Kapitola 5

Závěr

Každý jazyk používá pro implementaci síťové komunikace prostředky pro něj vhodné. Často je základem pro implementace v ostatních jazycích implementace v jazyce C. V těchto jazycích je poté jen dodáno rozhraní ke knihovnám jazyka C (např. implementace protokolu SCTP v jazyce Python nebo implementace JNetPcap v jazyce Java).

Jazyk C má z použitých jazyků nejrozsáhlejší podporu. Tato implementační podpora je často základem pro implementace v ostatních jazycích. Jazyk C poskytuje mnoho datových typů a struktur určených pro implementaci síťové komunikace. Poskytuje také velké množství funkcí a konstant (velikosti hlaviček, typy protokolů, ...).

Podpora implementace síťové komunikace interpretovaných jazyků Perl a Python je realizována prostřednictvím řady modulů. U interpretovaných jazyků občas chybí implementace různých struktur a datových typů pro uložení různých typů adres (kromě datového typu řetězec). Týká se to například implementace dekódování ethernetové hlavičky. Mnoho těchto problémů lze řešit pomocí regulárních výrazů a funkcí pack/unpack.

V prostředí Windows v jazycích Java a C# existuje mnoho tříd určených pro implementaci síťové komunikace (např. odlišné třídy pro implementaci schránek protokolu TCP a UDP). Třídy mají velké množství metod pro jejich manipulaci. Knihovny pro implementaci komunikace na síťové a linkové vrstvě využívají knihovny libpcap (JNetPcap) a winpcap (PcapDotNet).

V této práci bylo mým úkolem nastudovat postupy při implementaci komunikace prostřednictvím různých komunikačních protokolů v různých jazycích. Tyto poznatky jsem využil k implementaci této komunikace. Výsledkem implementace je soubor aplikací (klientů a serverů) implementovaných v různých jazycích (v prostředích Unixu a Windows) a komunikujících na různých vrstvách síťové komunikace (transportní, síťová a linková). Prostředky využité pro implementaci síťové komunikace byly popsány v kapitole Implementace protokolů a otestovány. Při testování jsem se zaměřil na funkčnost přenosu a konkurentnost v rámci komunikace v jednom programovacím jazyce, ale také jsem testoval vybrané příklady komunikace mezi jednotlivými jazyky (v rámci stejného protokolu). Na závěr jsem na základě postupu při tvorbě implementací porovnal prostředky využité pro implementaci v jednotlivých jazycích.

Přínosem této práce je shrnutí možných postupů implementace síťové komunikace na různých platformách. Pomocí příkladů je ukázáno využití postupů pro implementaci této komunikace v různých jazycích a prostředích. V práci jsou shrnuty postupy a knihovny pro tuto implementaci. Tyto postupy jsou předvedeny prostřednictvím sady vzorových implementacích, které byly na základě nich vytvořeny.

Tato práce se nezaobírá celou problematikou síťové komunikace a dále by bylo možné

ji rozšířit například o další programovací jazyky nebo o využití podpory protokolu IPv6, kterým se práce zabývala jen okrajově teoreticky. Protokolu IPv6 lze využít jak v případě implementace schránek, tak v případě implementace komunikace na síťové vrstvě (komunikace prostřednictvím protokolu IPv6).

Literatura

[1] TCPDUMP/LIBPCAP public repository [online]. http://www.tcpdump.org/, 2010 [cit. 2012-01-22].

[2] BARR, G.: Graham Barr / libnet [online]. http://search.cpan.org/dist/libnet/, 2010-05-31 [cit. 2012-02-26].

[3] BATES, R.; GREGORY, D.: *Voice & Data Communication Handbook*. Boston: McGraw-Hill, páté vydání, 2006, ISBN 9780071661874.
URL http://books.google.cz/books?id=APNlQxoNDRIC

[4] BRADEN, R.: Requirements for Internet Hosts – Communication Layers. RFC 1122, říjen 1989.

[5] BRICKNER, B.: Pcap.Net [online]. http://pcapdotnet.codeplex.com/.

[6] DEERING, S.; HINDEN, R.: Internet Protocol, Version 6 (IPv6) Specification. RFC 2460, prosinec 1998.

[7] DOUBA, N.: pylibnet: python module for libnet [online].
http://pylibnet.sourceforge.net/.

[8] FOOT, G.; CATLETT, C.: Libnet home page [online]. http://libnet.sourceforge.net/, 2003-02-18 [cit. 2012-01-22].

[9] International Organization for Standardization/International Electrotechnical Commission: *Information technology – Open Systems Interconnection – Basic Reference Model: The Basic Model*. listopad 1994.
URL http://www.ecma-international.org/activities/Communications/TG11/s020269e.pdf

[10] POSTEL, J.: User Datagram Protocol. RFC 768, srpen 1980.

[11] POSTEL, J.: Internet Control Message Protocol - DARPA Inernet Programm, Protocol Specification. RFC 792, září 1981.

[12] POSTEL, J.: Internet Protocol - DARPA Inernet Programm, Protocol Specification. RFC 791, září 1981.

[13] POSTEL, J.: Transmission Control Protocol - DARPA Inernet Programm, Protocol Specification. RFC 793, září 1981.

[14] POTTER, T.: Net::Pcap [online].
http://search.cpan.org/ kcarnut/Net-Pcap-0.05/Pcap.pm.

[15] Sly Technologies, Inc.: *jNetPcap [online].*

[16] SONG, D.: pcap 1.1: Python Package Index [online].
http://pypi.python.org/pypi/pcap.

[17] STEVENS, W. R.; FENNER, B.; RUDOFF, A. M.: *Unix network programming.*
Boston: Addison-Wesley, třetí vydání, 2004, ISBN 0-13-141155-1.

[18] STEWART, R.: Stream Control Transmission Protocol. RFC 4960, září 2007.

[19] STEWART, R.; TUEXEN, M.; POON, K.: Sockets API Extensions for the Stream
Control Transmission Protocol (SCTP). RFC 6458, prosinec 2011.

Příloha A

Obsah CD

Na přiloženém CD se nachází:

- Zpráva BP:
  - ve formě zdrojového textu (latex)
  - ve formě výsledného pdf souboru
- Implementace:
  - zdrojové kódy implementací
  - knihovny přiložené u zdrojových kódů využité při implementaci
  - Readme soubory (překlad a spuštění aplikací)
  - Sada použitých testovacích souborů

Příloha B

Manual

Tato příloha ukazuje příklady překladu a spuštění vzorových aplikací v jednotlivých jazycích.

Ve všech jazycích se aplikace spouští s následujícími parametry:

- -a=adresa_serveru (Pro klienty na všech protokolech – typ adresy se liší podle protokolu)
- -p=číslo_portu (pro klienty a servery na transportní vrstvě)
- -i (volba iterativního serveru, jinak je konkurentní - všechny protokoly)
- -f=název_požadovaného_souboru (klienti na všech implementovaných protokolech)

Jazyk C

Překlad implementací v jazyce C probíhá pomocí přiloženého Makefile.

Spuštění:

- server: ./server (parametry)
- klient: ./client (parametry)

Příklad (TCP):

- server: ./server -p=12345 -i
- klient: ./client -a=merlin.fit.vutbr.cz -p=12345 -f=soubor

Jazyk Perl

Spuštění:

- server: perl server.pl (parametry)
- klient: perl client.pl (parametry)

Příklad (UDP):

- server: perl server.pl -p=12345 -i
- klient: perl client.pl -a=merlin.fit.vutbr.cz -p=12345 -f=soubor

Jazyk Python

Spuštění:

- server: python server.py (parametry)
- klient: python client.py (parametry)

Příklad (SCTP):

- server: python server.py -p=12345 -i
- klient: python client.py -a=merlin.fit.vutbr.cz -p=12345 -f=soubor

Jazyk Java

Překlad:

- komunikace transportní vrstvy
  - javac server.java
  - javac client.java
- komunikace na síťové a linkové vrstvě
  - javac -cp .;jnetpcap.jar server.java
  - javac -cp .;jnetpcap.jar client.java

Spuštění:

- komunikace transportní vrstvy
  - java server (parametry)
  - java client (parametry)
- komunikace na síťové a linkové vrstvě
  - java -cp .;jnetpcap.jar server (parametry)
  - java -cp .;jnetpcap.jar client (parametry)

Příklad (IP):

- server: java -cp .;jnetpcap.jar server
- klient: java -cp .;jnetpcap.jar client -a=147.229.176.9 -f=soubor

Jazyk C#

Překlad:

- komunikace transportní vrstvy
  - csc server.cs
  - csc client.cs
- komunikace na síťové a linkové vrstvě
  - csc /reference:PcapDotNet.Analysis.dll,PcapDotNet.Base.dll,PcapDotNet.Core.dll, PcapDotNet.Core.Extensions.dll,PcapDotNet.Packets.dll client.cs
  - csc /reference:PcapDotNet.Analysis.dll,PcapDotNet.Base.dll,PcapDotNet.Core.dll, PcapDotNet.Core.Extensions.dll,PcapDotNet.Packets.dll server.cs

Spuštění:

- server (parametry)
- client (parametry)

Příklad (Ethernet):

- server: server -i
- klient: client -a=AA:BB:CC:DD:EE:FF -f=soubor