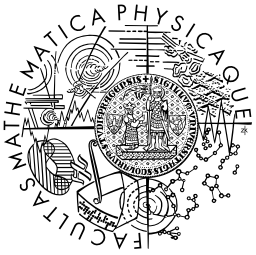


MATEMATICKO-FYZIKÁLNÍ FAKULTA
Univerzita Karlova

BAKALÁŘSKÁ PRÁCE

Martin Mirbauer

Peer-to-peer síť pro decentralizované uložení souborů a distribuované zpracování úloh

Katedra softwarového inženýrství

Vedoucí bakalářské práce: RNDr. Filip Zavoral, Ph.D.

Studijní program: Informatika

Studijní obor: Softwarové a datové inženýrství (ISDI)

Praha 2016

Prohlašuji, že jsem tuto bakalářskou práci vypracoval(a) samostatně a výhradně s použitím citovaných pramenů, literatury a dalších odborných zdrojů.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona v platném znění, zejména skutečnost, že Univerzita Karlova má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle §60 odst. 1 autorského zákona.

V ..... dne ..... Podpis autora

Název práce: Peer-to-peer síť pro decentralizované uložení souborů a distribuované zpracování úloh

Autor: Martin Mirbauer

Katedra: Katedra softwarového inženýrství

Vedoucí bakalářské práce: RNDr. Filip Zavoral, Ph.D., Katedra softwarového inženýrství

Abstrakt: Peer-to-peer sítě umožňují implementaci aplikací s vysokým výkonem, škálovatelností a odolností proti výpadkům. Aktuálně nejrozšířenější sítě jsou většinou jednoúčelové (např. file-sharing) a často postrádají zabezpečení proti zlým uzlům; systémy pro distribuci úloh jsou typicky centralizované, což omezuje jejich výkon. Proto jsme v této práci analyzovali praktičnost použití architektury peer-to-peer pro distribuci úloh a navrhli jsme potřebnou infrastrukturu (uložení souborů a jejich vyhledání) včetně možných bezpečnostních opatření. Pro otestování návrhu a ověření použitelnosti jsme implementovali klientský program (uzel) zajišťující chod sítě, umožňující přístup k souborům sdíleným ostatními uzly a distribuci úloh mezi uzly v síti. Výsledný program je multiplatformní a umožňuje rozšíření o další funkce.

Klíčová slova: peer-to-peer sítě, distribuované zpracování

Title: Peer-to-peer Network for Decentralized File Storage and Distributed Task Processing

Author: Martin Mirbauer

Department: Department of Software Engineering

Supervisor: RNDr. Filip Zavoral, Ph.D., Department of Software Engineering

Abstract: Peer-to-peer networks allow development of high-performance, scalable and fault-tolerant applications. The most widely used peer-to-peer networks are mostly single-purpose (e.g. file-sharing) and usually lack countermeasures against sybil nodes; typical task distribution systems are centralized, which limits their performance. In this work we have analyzed the usability of peer-to-peer architecture for task distribution and designed the required infrastructure (file storage and search) including evaluation of possible security features. In order to evaluate the proposed design's usability we have implemented a client application (node) participating in the network's function, allowing access to files shared by other nodes and task distribution within the network. The resulting application is multi-platform and can be extended with more functionality.

Keywords: peer-to-peer networks, distributed processing

Rád bych poděkoval svému vedoucímu práce RNDr. Filipu Zavoralovi, Ph.D. za přínosné rady a pomoc při práci. Také bych chtěl poděkovat své rodině za podporu a trpělivost.

Obsah

1. Úvod

Peer-to-peer síť je množina vzájemně propojených počítačů, které sdílejí své zdroje (typicky úložiště nebo výpočetní výkon) s ostatními. Díky rozdělení zátěže mezi více počítačů může síť uživatelům poskytovat vyšší výkon a odolnost proti selhání než centralizovaný systém.

Peer-to-peer sítě jsou především využívané pro sdílení souborů, vyhledání protistrany ke komunikaci a ke zprostředkování komunikace mezi uživateli.

1.1 Cíl práce

Cílem práce je navrhnout peer-to-peer síť pro decentralizované uložení souborů a distribuované zpracování úloh. Při návrhu sítě se zaměříme na její výkon a ochranu před zlými uzly.

Jako součást práce implementujeme program uzlu sítě umožňující decentralizované uložení souborů a distribuci úloh mezi uzly v síti. Výkon výsledného programu změříme a srovnáme s existujícími řešeními.

1.2 Související práce

Publikované materiály se zaměřují především na návrh vhodných topologií sítě (např. [48] a [49]), které i při velkém počtu uzlů mají dobré vlastnosti – umožňují rychlé nalezení uzlu odpovědného za určitý zdroj, a to s minimální režií (počtem navázaných spojení s ostatními uzly).

Jiné materiály se věnují zajištění důvěryhodnosti uzlů sítě, omezení počtu zlých uzlů nebo zvýšení odolnosti sítě proti případným útokům [65] nebo hodnotí bezpečnost již existujících peer-to-peer sítí [62].

Problematikou peer-to-peer sítí se zabývá velké množství prací. Z vybraných (citovaných) prací jsme čerpali mj. při výběru vhodných přístupů během návrhu vlastní sítě. Při analýze řešení obecných problémů, které nesouvisí pouze s peer-to-peer sítěmi, jsme čerpali také z dokumentů RFC, dokumentací knihoven zvažovaných k použití a výsledků měření chování reálných sítí.

1.3 Struktura práce

V kapitole 2 detailně analyzujeme a navrhneme řešení problémů, které je nutné vyřešit při implementaci vlastní peer-to-peer sítě.

Kapitola 3 se zabývá využitím navržené peer-to-peer sítě pro sdílení souborů se zajištěním jejich spolehlivého uložení, s distribucí úloh mezi uzly a návrhem použití sítě pro jiné komplexnější aplikace.

V kapitole 4 popíšeme metodiku měření výkonu a chování implementované sítě a zhodnotíme výsledky měření.

Kapitoly 5 a 6 obsahují dokumentaci vzniklé implementace klienta peer-to-peer sítě pro programátory a pro uživatele.

Kapitola 7 shrnuje výsledky celé této práce a popisuje její další možná rozšíření.

2. Analýza

2.1 Programovací jazyk a architektura aplikace

Výsledná aplikace má být přenositelná a díky efektivnímu využití hardwarových prostředků by měla poskytovat dobrý výkon, proto jsme zvolili programovací jazyk C++ ve verzi C++11.

Výsledkem překladu z C++ je typicky strojový kód, který je prováděn nativně, není tedy nutné (ve srovnání např. s jazykem C#) provádět překlad programu za běhu (just-in-time) na potenciálně slabším hardwaru ani běh virtual machine (Common Language Runtime nebo Mono Runtime). Důsledkem je úspora použité operační paměti i času procesoru, ale i absence automatické správy paměti, která by mohla usnadnit práci při implementaci. Podobně standardní knihovna jazyka C++ neposkytuje tolik funkcionality jako knihovny, které jsou součástí .NET frameworku (např. síťovou komunikaci, serializaci a kryptografii), proto je nutné použít knihovny třetích stran nebo funkce implementovat samostatně.

Protože jeden požadavek (např. na vyhledání souboru) může být předáván mezi více uzly, než se najde uzel, který ho dokáže vyřešit, je důležité minimalizovat dobu zpracování požadavku na jednom uzlu. Každý požadavek je proto nutné zpracovat ihned po jeho přijetí, program musí být řízen událostmi – paradigma *event-driven programming*. Z hlediska zátěže jednotlivého uzlu by nemělo být nutné zpracovávat velké množství požadavků současně, jako je potřeba v serverových aplikacích, protože by se zátěž měla při běžném fungování sítě rozložit rovnoměrně mezi uzly. Nemělo být tedy být nutné používat více pracovních vláken (*thread pool*), ale stačí události (např. přijetí zprávy) zpracovávat asynchronně v jednom vlákně, jako to například provádí runtime programovacího jazyka ECMAScript.

2.2 Struktura popisu problému

Pro zjednodušení analýzy je použit referenční model ISO/OSI jako kostra, protože problémy, které je potřeba při návrhu a implementaci řešit, rozděluje přehledně do vrstev.

2.3 Síťová vrstva (L3)

Nejnižší vrstvou, kterou bude síť přímo používat, je síťová vrstva. Poskytuje adresování počítačů, na kterých poběží klientský program, a směrování paketů, které si budou uzly (instance klientského programu) vzájemně posílat.

2.3.1 První připojení do sítě

Pro připojení do peer-to-peer sítě musí každý uzel znát adresu alespoň jednoho uzlu – počítače, který je již v síti připojený. Tyto adresy tzv. *bootstrap uzlů* mohou být veřejně známé, pokud „provozovatel“ peer-to-peer sítě zajišťuje dostupnost uzlů na daných adresách. Pak mohou být *bootstrap* adresy součástí programu nebo jeho konfigurace. V opačném případě se může uzel snažit navázat spojení

naslepo – zkoušet náhodné adresy, nebo posílat *multicast* žádosti o připojení. To však lze bez problémů provozovat pouze v lokálních sítích, v sítí Internet by totiž mohl být takový provoz považovaný za útok tzv. *port scanning*.

Pokud se programu podaří připojit k peer-to-peer síti (ať přes *bootstrap uzly* nebo vyhledáváním), může si program zapamatovat adresy počítačů, se kterými komunikoval, a pro navázání prvního spojení po restartu programu použít místo *bootstrap uzlů* adresy počítačů, se kterými komunikoval před vypnutím. Tím se zamezí zbytečnému zatížení *bootstrap uzlů* a případné nefunkčnosti celé sítě, pokud by selhaly všechny *bootstrap uzly*.

2.3.2 Získání adresy druhé strany a navázání spojení

Pokud je už uzel připojený do sítě (tj. je připojený k alespoň jednomu dalšímu uzlu), může potřebovat spojit se s dalším uzlem nebo může být požádán o spojení, např. pokud potřebuje soubor uložený na jiném uzlu, resp. jiný uzel chce lokální uzel požádat o soubor. Aby mohlo toto spojení nastat, musí alespoň jeden z uzlů znát adresu druhého, uzly proto musí zveřejňovat vlastní adresy. Je proto nutné, aby každý uzel zjistil adresu počítače, na kterém sám běží.

Program může bez komunikace s okolím získat pouze adresu svého síťového rozhraní, pokud je však počítač za směrovačem, který provádí překlad adres/portů [1, *NAPT*, viz kap. 4.1.2], nezná adresu, pod kterou je vidět zvenku své sítě, musí tedy kontaktovat nějaký server mimo lokální síť, aby zjistil svoji veřejnou adresu. Pokud však ostatním uzlům v peer-to-peer síti zveřejní pouze svoji vnější adresu, pak jiný uzel, který může ležet ve stejné síti, se bude pokoušet o navázání spojení se společným *NAPT* směrovačem, což buď selže (v závislosti na nastavení firewallu), nebo bude spojení neefektivní – zbytečně bude zatěžovat síťové prvky přeposílající pakety na cestě spojení. Proto je vhodné ostatním uzlům zveřejnit více adres, aby mohly zkusit navázat spojení od co nejpřímějšího (v lokální síti) po globální (přes veřejné adresy v rámci Internetu).

2.3.3 Problém navázání přímého spojení

Pro funkčnost peer-to-peer sítě je nutné, aby spolu zúčastněné počítače mohly přímo komunikovat. V případě, že má každý počítač unikátní veřejnou adresu, je navázání spojení mezi počítači triviální. V opačném případě, např. pokud více počítačů sdílí jednu veřejnou IPv4 adresu a směrovač provádí překlad adres a portů, je potřeba zajistit, aby se libovolné dva počítače v síti mohly spojit. Podle měření [2] z roku 2008 leží za *NAPT* směrovačem nebo firewallem téměř 90% uzlů v peer-to-peer síti Tribler [3].

Pokud leží za *NAPT* směrovačem pouze jeden z dvojice počítačů, které se chtějí spojit, stačí se pokusit o navázání spojení v obou směrech – oba počítače zkusí vytvořit spojení s druhou stranou. V jednom směru komunikace selže, protože *NAPT* data vyhodnotí jako nevyžádaná a zahodí je, ale v druhém směru bude úspěšná: počítač za *NAPT* směrovačem bude při navazování spojení v roli klienta, druhý (s veřejnou adresou) v roli serveru.

Pokud jsou však oba počítače za *NAPT* směrovači, je potřeba nějakým způsobem spojení *otevřít*, nebo *prorazit* či *obejít*:

- alespoň na jedné straně *otevřít* možnost přímého spojení – ručně nastavit *port forwarding*, nebo automaticky, pokud to směrovač podporuje,
- zkusit *prorazit NAPT* směrovače, přes které má procházet komunikace, tzv. metodou *hole punching*, nebo
- veškerou komunikaci posílat přes třetí počítač, který je dosažitelný z obou uzlů.

Všechna tato řešení zasahují do vyšších vrstev (transportní nebo aplikační), řeší však problém způsobený nedostatkem adres síťové vrstvy, proto je popíšeme zde.

2.3.4 Možná řešení

Problém navázání spojení mezi libovolnými dvěma uzly bylo nutné vyřešit již dříve především v oblasti multimediálních aplikací, např. pro IP telefonii, kde je důležité, aby zpoždění (*latency* a *jitter*) paketů bylo minimální. Proto byly navrženy a implementovány následující protokoly, které lze použít i pro naši aplikaci.

2.3.4.1 UPnP-IGD, PCP

Protokol IGD [18] (*Internet Gateway Device Standardized Device Control Protocol*) ze sady protokolů UPnP (*Universal Plug and Play*) a protokol PCP (*Port Control Protocol*) [19] umožňují nastavit dočasný *port forwarding*.

Port forwarding je záznam v tabulce překladu adres – pakety přijaté *NAPT* směrovačem na určitém portu jsou předány uzlu ve vnitřní síti, který o *otevření* portu požádal. Poté může libovolný uzel navázat spojení na veřejnou adresu *NAPT* směrovače na tento otevřený port, čímž se spojí s daným uzlem. Kromě vytvoření umožňují protokoly IGD a PCP také *port forwarding* obnovit (zamezit vypršení platnosti rezervovaného portu) nebo zrušit.

Protože se jedná o (dočasnou) změnu nastavení směrovače, musí *NAPT* směrovač některý z těchto protokolů podporovat a být nastaven, aby požadavkům uzlů ve vnitřní síti vyhověl.

2.3.4.2 STUN

Pro zjištění vnější adresy *NAPT* směrovače slouží protokol STUN (*Session Traversal Utilities for NAT*) [4].

Jedná se o protokol typu klient-server – server leží ve veřejném Internetu, klient se může serveru dotázat na svoji veřejnou adresu a zjistit, jak se mění číslo portu, ze kterého byl dotaz vyslán. Tím může program detekovat, jestli k překladu adres dochází, určit typ překladu a získat adresu, pod kterou je zvenku viditelný.

2.3.4.3 TURN

Protokol TURN (*Traversal Using Relays around NAT*) [5] umožňuje přemostění spojení přes třetí stranu s veřejnou adresou, tzv. *TURN server*.

TURN klient u TURN serveru zarezervuje port, poté každý paket (po zapouzdření v TURN zprávě a uložení adresy a portu příjemce v hlavičce TURN zprávy) posílá klient TURN serveru, ten jej odešle jako UDP datagram adresátovi. Podobně pokud přijde UDP datagram na TURN server na port, který má

nějaký klient zarezervovaný, je datagram přeposlán (v TURN zprávě s hlavičkou obsahující adresu a port odesílatele) danému TURN klientovi.

2.3.4.4 ICE

Protokol ICE (*Interactive Connectivity Establishment*) [6] zajišťuje navázání co nejpřímějšího spojení, k čemuž používá protokol STUN, případně i TURN.

Před navázáním spojení zjistí obě strany vlastní *seznam kandidátů* – seznam dvojic $(adresa, port)$, na kterých může být daný uzel dostupný. *Seznam kandidátů* obvykle obsahuje lokální adresy síťových rozhraní s lokálním portem, *vnější* adresu *NAPT* směrovače a jím přiřazený port (pokud směrovač podporuje protokol IGD nebo PCP), *vnější* adresu s portem získané ze STUN serveru a adresu TURN serveru a port na něm rezervovaný. Poté si uzly navzájem vymění *seznamy kandidátů* a oba současně se pokouší navázat spojení s *kandidáty* druhé strany v pořadí od lokálních adres po veřejné, dokud úspěšně nenaváží první spojení.

Protože se uzly snaží navzájem kontaktovat na adresách získaných ze STUN serveru, může v případě, že oba uzly leží za *NAPT* směrovači (dále NAT), proběhnout úspěšný *hole punching*. Při navazování spojení pošle jeden uzel (A) paket na adresu NAT druhého uzlu (B), ale paket je zahozen, protože pochází od neznámého zdroje (a NAT uzlu B neví, kterému počítači ve vnitřní síti ho má předat), ale NAT A si zapamatuje, že poslal na adresu B nějaký paket. Když pak druhý uzel (B) pošle paket prvnímu (A), NAT uzlu A považuje přijatý paket za odpověď od uzlu B a předá jej uzlu A, navíc NAT B si zapamatuje, že uzel B poslal paket na adresu uzlu A, čímž vznikne obousměrný komunikační kanál. Tento postup funguje za předpokladu, že *NAPT* směrovače přiřazují uzlům stejnou dvojici $(adresa, port)$ při vzájemné komunikaci (mezi uzly A a B) i při komunikaci jednotlivých uzlů se STUN serverem, nebo alespoň je překlad adres předvídatelný (např. stejná adresa, zvyšování čísla portu po 1) – pak stačí do seznamu kandidátů vložit předpovězené dvojice.

Pokud je v alespoň jednom *seznamu kandidátů* obsažen i TURN server, je úspěch navázání spojení zaručen. Pokud je ale možné navázat spojení přímějším způsobem, tato cesta se najde dříve.

2.3.4.5 Existující implementace

Popsané protokoly STUN, TURN a ICE jsou implementované v následujících knihovnách: PJNATH [7], libjingle [9] a libnice [8]. Knihovna PJNATH je zveřejněna pod licencí GPL, její použití by vyžadovalo zveřejnění zdrojového kódu celé aplikace pod touto licencí. Zaměříme se proto na zbývající knihovny.

Knihovna libjingle je napsána v programovacím jazyce C++, nabízí rozhraní v jazyce C a je zveřejněna pod neomezující licencí 3-Clause BSD. Knihovna je určena primárně pro *real-time* přenosy (audio/video hovory), ale lze použít i k *proražení NAPT* směrovače a přenosu obecných dat. Dříve byla použita v instant messengeru Google Talk [10], nyní je používána např. programem Steam [11] pro navázání přímého spojení mezi hráči. Knihovna je nyní součástí [12] kódu WebRTC [13] webového prohlížeče Chromium [14].

Knihovna libnice je napsaná v jazyce C, zveřejněná pod LGPL – nevyžaduje zveřejnění zdrojového kódu, stačí poskytnout již přeložené `.o` soubory zbytku

aplikace, aby bylo možné program znovu slinkovat se samostatně přeloženou knihovnou. Knihovna je použita například v instant messaging klientech Pidgin [15] a Empathy [16].

2.3.5 Shrnutí

Pro funkčnost sítě nad síťovým protokolem IPv4 bude potřeba alespoň jeden *bootstrap uzel* se statickou veřejnou IPv4 adresou (v případě scanování nemusí nutně být statická), STUN server a TURN server. Všechny tyto služby mohou být provozovány na stejném počítači, případně mohou být součástí každého uzlu.

Díky stálému rozšiřování protokolu IPv6, který díky dostatku adres pro každý počítač nenutí ke sdílení jedné síťové adresy více počítači, lze očekávat, že v budoucnosti nebude navázání přímého spojení mezi počítači problémem jako v případě IPv4, proto implementaci řešení tohoto problému ponecháme jako možné rozšíření.

2.4 Transportní vrstva (L4)

Transportní vrstva zajišťuje *end-to-end* přenos dat.

Protože je jedním z cílů této práce multiplatformnost výsledného programu, je možné používat pouze protokoly, které podporují všechny cílové operační systémy, tedy TCP nebo UDP.

2.4.1 Srovnání TCP a UDP pro potřeby aplikace

Pro přenos souborů potřebujeme spolehlivý přenos dat, vyhovoval by nám tedy protokol TCP. Ale protože TCP používá spojovaný typ přenosu a vyžaduje *three-way handshake* na začátku spojení, ztěžuje navázání spojení za *NAPT* směrovačem. Naopak UDP používá nespojovaný typ přenosu, což usnadňuje začátek komunikace, neposkytuje však potřebnou spolehlivost, kterou je nutné implementovat na vyšší (aplikační) vrstvě. Protože ale zajištění spolehlivosti přenosového kanálu logicky spadá do transportní vrstvy, je popsáno v této kapitole (2.4.3).

2.4.2 Výběr transportního protokolu podle úspěšnosti navázání přímého spojení

Podle výsledků měření *NAT-Analyzer* [17] lze přímé spojení navázat v přibližně v 62 % případů. Při použití *hole punching*u je úspěšnost navázání přímého spojení pouze 19 % pro protokol TCP a 49 % pro UDP. Pokud *NAPT* směrovač podporuje protokol IGD, lze navázat přímé spojení v 39 % resp. 61 % případů.

Pro nejvyšší úspěšnost je tedy vhodné použít transportní protokol UDP, implementovat *hole punching*, případně přidat podporu IGD. Spojení, která se tímto způsoben nepodaří navázat, je nutné přemostit přes ostatní uzly (např. za použití protokolu TURN).

2.4.3 Zajištění spolehlivosti nad UDP

Protože jsme zvolili transportní protokol UDP, který je nespolehlivý, musíme zajistit spolehlivost sami. Je nutné provádět kontrolu poškození paketů, opravu pořadí doručování, implementovat řízení toku (*flow control*), předcházet zahlcení sítě (*congestion control*) a detekovat případné přerušení spojení. Pokud bychom navrhovali vlastní transportní protokol, mohli bychom vytvořit protokol „na míru“, který by měl (např. oproti TCP) minimální režii (neobsahoval by nevyužitá pole *reserved* a *urgent pointer* v každém paketu), umožňoval by rychlé navázání spojení a díky vlastnímu algoritmu pro řízení toku a předcházení zahlcení bychom mohli dosáhnout vyšší propustnosti ve srovnání s TCP.

Pro dosažení maximálního výkonu sítě by bylo vhodné používat dva transportní protokoly – jeden pro posílání řídicích zpráv, druhý pro přenos dat. Protokol pro řídicí zprávy by měl umožňovat rychlé navázání spojení a minimální přenosové zpoždění, protokol pro data by měl efektivně využít propustnost linky, ale dát přednost řídicím zprávám. Pro zjednodušení implementace budeme používat stejný transportní protokol pro přenos zpráv i dat, případnou optimalizaci přenosů ponecháme jako možné rozšíření.

Protože návrh protokolu a jeho implementace je netriviální a testování v různých prostředích (např. v sítích používajících technologie Ethernet, PowerLine, xDSL, Wi-Fi, EDGE, 3G, LTE, nebo jejich kombinaci) je časově náročné, použijeme již existující implementaci vhodného protokolu.

2.4.4 Existující protokoly zajišťující spolehlivost nad UDP

Pro zajištění spolehlivého přenosu nad UDP lze použít následující protokoly: RBUDP (*Reliable Blast UDP*) [20], TSUNAMI [21], PA-UDP (*Performance Adaptive UDP*) [22], *SCTP over UDP* [23], μTP (*Micro Transport Protocol/uTorrent Transport Protocol*) [24] a UDT (*UDP-based Data Transfer Protocol*) [25].

Přestože u všech těchto protokolů jsou data posílána protokolem UDP, protokoly RBUDP a TSUNAMI interně používají protokol TCP pro řízení datových přenosů. Navíc RBUDP je podle autorů určen spíše do lokálních sítí nebo do sítí, kde je jediným provozem, protože jeho algoritmus pro řízení toku a předcházení zahlcení sítě je příliš agresivní. K protokolu PA-UDP jsme nenalezli funkční implementaci – referenční implementace odkazovaná z [22] je nedostupná, knihovní implementace protokolu *SCTP over UDP* byla sice nalezena, ale pouze v jazyce Java. K analýze tedy zbývají protokoly μTP a UDT.

2.4.5 μTP

Protokol μTP byl navržen jako náhrada protokolu TCP v síti *BitTorrent* [36]. Použitý *congestion control* algoritmus LEDBAT (*Low Extra Delay Background Transport*) [26] je navržen tak, aby dal přednost probíhajícím přenosům protokolem TCP, proto je protokol μTP vhodný spíše na přenosy na pozadí. Pokud bychom zvolili μTP, museli bychom protokol částečně upravit, protože používáním původního *congestion control* algoritmu LEDBAT bychom omezili výkon výsledné peer-to-peer sítě v rušných síťových prostředích.

μTP je implementován v knihovně libutp [27], která je zveřejněna pod volnou MIT licencí, nepodmiňující použití knihovny zveřejněním zdrojového kódu celé aplikace. Knihovna je napsána v jazyce C++ a nabízí API pro jazyk C. Knihovna je v praxi používaná v *BitTorrent*ovém klientu *uTorrent*, který patří mezi nejvíce rozšířené [28].

2.4.6 UDT

Protokol UDT byl navržen pro dosažení co nejvyšší propustnosti v sítích s vysokým součinem *propustnost* $*$ *odezva*. Kromě vestavěného *congestion control* algoritmu, který dokáže koexistovat s TCP přenosy [29, graf „UDT's TCP friendliness index"], nabízí možnost implementovat vlastní *congestion control* algoritmus, který lze zvolit pro každý socket zvlášť, čímž lze komunikaci přizpůsobit různým typům připojení – např. datové a řídicí, viz kapitolu 2.5.1.

Protokol UDT je používán v praxi pro multimediální přenosy (aplikace Movie2Me a NiFTy TV), migraci virtuálních strojů (VMware vCloud Connector, pro kopírování virtuálních strojů mezi instancemi hypervisoru vSphere) [30] i na obecný přenos souborů – v high performance computing (GridFTP) i pro synchronizaci dat programem PowerFolder. Referenční implementace protokolu UDT – knihovna udt [31] je zveřejněna pod licencí 3-Clause BSD, lze tedy použít bez nutnosti zveřejňování zdrojového kódu celé aplikace a podle porovnání transportních protokolů nad UDP [32, Figure 6] je z porovnávaných implementací nejméně náročná na výpočetní výkon.

Při testování referenční implementace jsme zjistili, že knihovna má vysokou režii pro udržování spojení (asi 3 pakety/s pro každé spojení). Protože v našem programu bude potřeba udržovat řádově stovky spojení s ostatními uzly (viz kap. 2.5.3), bylo nutné knihovnu udt upravit, aby bylo možné nastavit *keep-alive* interval podle potřeby.

2.5 Relační vrstva (L5)

Relační vrstva řídí komunikaci mezi uzly: určuje, které uzly mají být spojeny, zajišťuje nalezení potřebného uzlu a navázání a udržování spojení.

2.5.1 Počet spojení mezi dvojicí uzlů

Aby byla co nejlépe využita přenosová kapacita linky, je vhodné oddělit řídicí zprávy, které by měly být doručeny co nejdříve, od dat, jejichž opožděné doručení není kritické.

Pokud řídicí zprávy i data sdílejí jedno spojení, je vhodné dát řídicím zprávám přednost a zajistit jejich rychlé odeslání – nezahlcovat výstupní buffer – místo přenášení celých souborů je dobré dělit je na menší části a odesílat je teprve poté, co se odesílací buffer vyprázdní pod určitou hranici, aby řídicí zprávy dlouho nečekaly na odeslání. Velikost bufferu, příp. hranici jeho naplnění lze odvodit z propustnosti linky a maximálního přípustného zpoždění.

Pro datové přenosy je však vhodné vytvořit samostatné spojení, v ideálním případě s nižší prioritou, např. za použití méně agresivního *congestion control* algoritmu než v případě řídicího spojení, který v případě detekce ztráty paketu

sníží rychlost odesílání více, než *congestion control* algoritmus řídicího spojení. V tomto případě lze buď pro každý přenášený soubor vytvořit nové datové spojení (a zajistit, aby druhá strana věděla, který soubor je přenášen ve kterém spojení), nebo soubory v rámci jednoho datového spojení multiplexovat, jinak by přenos jednoho velkého souboru blokoval další přenosy souborů.

Pro implementaci odděleného datového spojení, resp. více datových spojení by bylo možné použít multiplexing v knihovně udt umožňující použít jedno „UDP spojení“ (stejný zdrojový i cílový port) pro více UDT spojení a použít funkce `sendfile` a `recvfile` pro přenos jednotlivých souborů. Pro vyloučení nutnosti navázání vícenásobného spojení mezi dvojicí uzlů, které by vyžadovalo přiřazení dat k jednotlivým spojením, a pro možnost přechodu na jiný transportní protokol (bez podpory multiplexingu) budeme používat pro komunikaci mezi dvojicí uzlů pouze jedno spojení společné pro data i řídicí zprávy.

2.5.2 Topologie sítě

Důležitou částí návrhu peer-to-peer sítě je volba topologie, tedy určení „kdo s kým má být spojen“

Pro analýzu existujících topologií je použito dělení typů peer-to-peer sítí podle [33].

2.5.2.1 Nestrukturované, centralizované

Implementačně nejjednodušším řešením by bylo použít síťovou architekturu klient-server po vzoru sítě *Napster* [34]: vytvořit centrální databázi pro ukládání metadat o souborech a pro vyhledávání uzlů poskytujících soubor, který chce některý z uzlů stáhnout. Taková síť by však měla *single point of failure* – v případě výpadku serveru by celá přestala fungovat.

Závislost na jednom centrálním serveru lze omezit možností volby mezi více servery poskytujícími metadata o uzlech a souborech. Tímto způsobem fungují sítě *Direct Connect* [35] a *eDonkey* [37] – uživatel si může zvolit tzv. *hub*, ke kterému je připojen. Přestože tento přístup snižuje pravděpodobnost přerušení funkce sítě, stále je nutné hostovat *hub* servery odděleně od uzlů. Vhodnější by bylo, aby metadata byla uložena společně se samotnými daty přímo na uzlech.

2.5.2.2 Nestrukturované, decentralizované

Abychom mohli odstranit závislost sítě na centrální databázi, musíme její úlohu předat samotným uzlům. V (implementačně) nejjednodušším případě nebude žádný uzel znát jiné informace, než své dostupné zdroje (např. soubory v případě file-sharing sítí) a seznam svých sousedů – pro vyhledávání konkrétního zdroje je nutné provádět náhodnou procházku po uzlech a dotazovat se navštěvovaných uzlů. V případě velké sítě nebude proveditelné v rozumném čase navštívit všechny uzly, není proto při požadavku na soubor zaručeno nalezení hledaného souboru v síti i přesto, že v dané peer-to-peer síti existuje.

Aby nebylo nutné připojovat se ke každému uzlu kvůli vyhledávání nějakého zdroje, lze požadavek na vyhledání přeposílat. Implementačně nejjednodušším řešením je *flooding* – rekurentní rozesílání požadavku všem sousedním uzlům.

Aby nedošlo k zahlcení sítě jediným požadavkem přeposílaným mezi uzly donekonečna, musí obsahovat každý požadavek TTL (*time-to-live*) – počet povolených přeposlání. Tak funguje například peer-to-peer síť *Gnutella* [38], která navíc TTL postupně zvyšuje, čímž snižuje globální zatížení sítě – nejdříve pošle požadavek s nízkým TTL, pokud není zdroj nalezen, pošle další požadavek s vyšším TTL, a to až do okamžiku, dokud požadovaný zdroj nenajde, nebo dokud nedosáhne maximálního povoleného TTL.

Flooding lze provádět také pouze na části sítě, například mezi tzv. *super-peers*, jako v peer-to-peer síti *KaZaA*. *Super-peers* jsou uzly s dostatečnou konektivitou, kapacitou úložiště a dostatečným výkonem, které jsou ostatními uzly zvoleny, aby udržovaly cache dostupných zdrojů v síti. Díky nižšímu počtu takových uzlů je možné, aby byl pro vyhledávání v rámci *super-peers* použit *flooding*. Síť tak může dosáhnout většího celkového počtu uzlů, ale vyhledávání zdrojů je stále neefektivní a zátěž uzlů není rovnoměrná.

2.5.2.3 Strukturované, decentralizované

Lepší škálovatelnosti lze dosáhnout i v případě, že jsou všechny uzly rovnocenné. Pokud zpřísníme pravidla pro spojování uzlů a vhodně a konzistentně rozdělíme odpovědnost za každý zdroj mezi uzly (tj. existuje prostá funkce $id \rightarrow uzel$, která pro každý zdroj určí odpovědný uzel; každý uzel ji umí vyhodnotit, příp. s pomocí ostatních uzlů), mohou si uzly předávat požadavky efektivněji – tak, že pro každý zdroj lze zaručit nalezení uzlu, který má daný zdroj na starosti, a to v počtu kroků (tj. přeposlání požadavku) shora omezeném nějakou funkcí celkového počtu uzlů v síti. Takové vlastnosti nabízí například topologie *CAN* [46], *Pastry* [47], *Chord* [48] a *Kademlia* [49].

Všechny tyto topologie pracují na principu distribuované hashovací tabulky (DHT). Pro DHT budeme předpokládat, že každý zdroj a každý uzel má jednoznačný identifikátor (m-bitové číslo). V případě přiřazování identifikátorů souborům lze identifikátor získat například jako hash obsahu souboru; lze použít např. hashovací funkci SHA-1, jejímž výsledkem je 160-bitové číslo. Topologie přiřadí každému zdroji (podle jeho identifikátoru) právě jeden odpovědný uzel a určí pravidla pro směrování dotazů, čímž také určují, který uzel má být se kterým spojen.

Topologie Kademlia je použita například v peer-to-peer sítích Kad [50] (primárně používaná file-sharing programem eMule [51]) a Mainline DHT [52] sloužící jako distribuovaný tracker v síti BitTorrent. V praxi používané sítě s topologií Chord, Pastry nebo CAN jsme nenalezli.

Pro intuitivní strukturu sítě a dobré vlastnosti [53, Table 2. Summary of structured P2P solutions.] ($\mathcal{O}(\log_2 N)$ kroků k nalezení daného uzlu při $\mathcal{O}(\log_2 N)$ spojeních s ostatními uzly, kde N je celkový počet uzlů) jsme zvolili topologii Chord.

2.5.3 Chord

V topologii Chord jsou identifikátory umístěny na pomyslné kružnici – po čísle `0xFF..FF` následuje `0x00..00`. Tímto očíslováním zdrojů a uzlů získáme jejich adresy v síti (vznikne tzv. *overlay network*). Chord přiřazuje každý zdroj takovému uzlu, jehož identifikátor následuje po identifikátoru zdroje. S topologií

musíme zajistit vhodné směrování požadavků (v rámci *overlay network*), případně zajistit spolehlivost – pokud na nějaký požadavek dotazovaný uzel neodpoví, nebo odpovědět neumí (např. neobsahuje hledaný soubor), lze se dotázat jiného uzlu.

Aby byl každý zdroj dosažitelný z každého uzlu, musí k němu existovat nějaká cesta (přes ostatní uzly). Chord existenci této cesty zaručuje tak, že každý uzel má udržovat spojení se svým *následníkem* (tj. uzlem, jehož identifikátor následuje po identifikátoru daného uzlu). Každý uzel je tedy dosažitelný po cestě „po směru hodinových ručiček“ po pomyslné kružnici.

Kvůli efektivnímu vyhledávání musí být kromě svého následníka každý uzel spojený s dalšími uzly (tzv. *fingers* spojení) – následníky identifikátorů $(n + 2^{k+1}) \bmod 2^m$, kde $1 \leq k \leq m$ a kde n je identifikátor lokálního uzlu. Díky těmto spojením lze pomocí distribuovaného binárního vyhledávání najít uzel zodpovědný za libovolný zdroj v nejvýše $\log_2 N$ krocích, kde N je celkový počet uzlů v síti [48, Theorem IV.2, s. 6].

Topologie Chord definuje funkce zajišťující nalezení požadovaného zdroje a pomocné funkce pro udržování správné struktury sítě při příchodu a odchodu uzlů. Komunikace mezi uzly probíhá pomocí zpráv.

Jediná funkce používaná vyššími vrstvami – `find_successor` – vyhledává uzel odpovědný za daný zdroj (tj. vyhledání následníka daného identifikátoru v síti), ostatní funkce pouze pomáhají udržovat topologii sítě při připojování a odpojování uzlů.

Referenční implementace Chord sice existuje [54], ale knihovna SFS/SFSlite, na které je závislá, by kvůli licenci GPL vyžadovala zveřejnění zdrojového kódu celé aplikace a navíc není kompatibilní s operačním systémem Windows. Proto byla vyvinuta implementace vlastní, mj. také kvůli provedení následujících změn v protokolu.

Topologie Chord počítá s nespojovaným typem přenosu, např. provádí kontrolu dostupnosti uzlů ve funkci `check_predecessor`. Topologii je proto vhodné přizpůsobit zvolenému typu přenosu.

Díky spojovanému typu přenosu nemusíme ručně kontrolovat dostupnost sousedních uzlů ani periodicky provádět stabilizaci (udržovat spojení s předepsanými uzly), protože o každém připojení nebo odpojení uzlu se dozvíme od transportní vrstvy. Tím se navíc sníží počet řídicích zpráv na nezbytné minimum.

Pro zajištění automatické stabilizace sítě musíme ošetřit následující události:

- v případě připojení nového následníka, resp. předchůdce informovat původního následníka, resp. předchůdce, kterým uzlem byl nahrazen, a
- v případě odpojení následníka nalézt nového – požádat nejbližší živý uzel o nalezení našeho správného následníka „proti směru hodinových ručiček“.

Změny topologie jsou detailněji popsány v příloze ve vygenerované dokumentaci v kapitole *Chord overlay network*.

2.5.4 RPC

Pro zjednodušení implementace *overlay network* i funkcionality ve vyšších vrstvách můžeme implementovat mechanismus vzdáleného volání procedur *RPC* (*remote procedure call*) s využitím řídicích zpráv pro komunikaci mezi uzly (posílání požadavků a odpovědí).

Vzdálené volání procedur umožňuje zavolat proceduru/funkci na vzdáleném počítači, ne nutně se stejnou volací konvencí, architekturou a operačním systémem, případně předat výsledek (návratovou hodnotu) volajícímu. Obvykle také zajišťuje základní spolehlivost – informuje volajícího v případě neúspěchu nebo timeoutu.

RPC může být použito k zajištění spolehlivého posílání řídicích zpráv v rámci sítě (např. stabilizačních zpráv pro udržování topologie) a zpracování případných odpovědí, důležité je i pro použití ve vyšších vrstvách. Především usnadňuje práci aplikační vrstvě – při volání vzdálené funkce stačí definovat akci, která má být provedena při úspěšném zavolání a akci při chybě. Například při navazování *fingers* spojení (funkce `fix_fingers` [48]) lze zavolat funkci `find_successor(`$n+2^i$`)` na vzdáleném uzlu a po přijetí návratové hodnoty – obsahující kontaktní informace uzlu – se k tomuto uzlu připojit. Případnou chybu je možné např. zapsat do logu.

V praxi jsou často používané protokoly *XML-RPC* [55], *JSON-RPC* [56] a *ONC/RPC* [57]. Protokoly *XML-RPC* a *JSON-RPC* používají pro zápis posílaných zpráv formát *XML*, resp. *JSON*, které sice nabízejí dobrou interoperabilitu, ale kvůli textovému formátu dat plýtvají přenosovou kapacitou, proto jsou vhodné spíše pro *API* pro komunikaci s jinými aplikacemi, kde zpoždění přenášených dat není kritické. Protokol *JSON-RPC* je použit v síti *Bitcoin* [58] pro ovládání referenční implementace klientského programu uzlu (na přenášení dat mezi uzly je použit vlastní binární protokol) a protokol *SOAP* [59], který vychází z *XML-RPC*, je běžně používaný pro webové služby. *ONC/RPC* používá binární formát *XDR* [60] pro přenos dat a proto jej lze použít i ve výkonově náročnějších případech, například pro sdílení souborů protokolem *NFS* [61].

Existující implementace *RPC* typicky podle zadaného rozhraní (tj. definice funkcí ve vlastním jazyce daného *RPC* protokolu) vygenerují pro serverovou a klientskou část zdrojový kód, který při zavolání funkce v klientské části kódu zprostředkuje provedení odpovídající funkce na serveru a předání návratové hodnoty zpět klientovi. Výsledný kód typicky používá pro komunikaci mezi serverem a klientem protokol TCP, bylo by tedy nutné vygenerovaný program (nebo kód *RPC* frameworku) upravit, aby používal jiný transportní protokol. (viz kapitolu 2.4)

Kvůli usnadnění programování případných alternativních klientů této peer-to-peer sítě by sice bylo vhodné zvolit již existující *RPC* protokol a použít některou z jeho implementací, ale pro lepší flexibilitu jsme vytvořili jednoduchou vlastní implementaci *RPC* využívající řídicí zprávy pro komunikaci mezi klientem a serverem a podporující použití libovolného formátu uložení dat (viz kapitolu 2.6.1). Toto řešení je asynchronní – používá *callback funkce*: při přijetí odpovědi (návratové hodnoty) zavolá *RPC* mechanismus funkci zvolenou při zavolání/vytvoření *RPC* požadavku, podobně v případě chyby.

2.5.5 Autentikace

Pro správné směrování a rozdělení odpovědnosti za zdroje v *overlay network* a pro spolehlivou autorizaci je nutné, aby každý uzel měl jednoznačnou identitu, kterou nelze snadno zfalšovat. Každý uzel by měl být schopen se při ověřování identity prokázat něčím, co by neměl mít (nebo znát) žádný jiný uzel.

Běžně používané peer-to-peer sítě, např. Mainline DHT a Kad, umožňují pou-

žití libovolného identifikátoru uzlu (tj. jeho identity v síti) bez omezení tvaru ani významu, typicky je tento identifikátor generován náhodně při připojení do sítě [52].

2.5.5.1 Možné útoky

V případě umožnění volby libovolného identifikátoru je možné, aby případný útočník například vytvořil *zlý uzel*, který na dotaz na nějaký vybraný zdroj odpovídá „nenalezeno“ a nastavil identifikátor tohoto uzlu tak, aby byl odpovědný za tento zdroj. Tak by mohl útočník znepřístupnit/zcenzurovat jakýkoli zdroj v síti.

Podobně by mohl útočník vytvořit více zlých uzlů takových, aby byly všemi sousedy nějakého uzlu. Tím by vybraný uzel mohl odstřihnout od zbytku sítě.

Dále by mohl vytvořit mnoho uzlů s náhodnými, rovnoměrně rozmístěnými identifikátory, které by pouze nedestruktivně přeposílaly požadavky ostatním uzlům a požadavky si navíc lokálně ukládaly, čímž by útočník mohl monitorovat provoz v síti. Případně by kromě monitorování provozu mohl provádět i popsané tzv. *eclipse* útoky, čímž by izoloval vybrané zdroje, resp. uzly. Tyto útoky v praxi probíhají [62], proto je vhodné při návrhu sítě navrhnout také protiopatření.

2.5.5.2 Ochrana

Aby bylo možné zamezit těmto útokům, lze použít public-key kryptografii, kdy by každý uzel měl vlastní klíč (s veřejnou a soukromou částí). Identifikátor uzlu v síti můžeme odvodit (např. hashovací funkcí) od veřejné části klíče, díky tomu bude moci každý uzel prokázat svoji identitu vlastnictvím soukromé části klíče (na žádost může vlastník klíče podepsat náhodně vygenerovaný jednorázový řetězec – *cryptographic nonce*).

2.5.6 Autorizace

Autorizace zajišťuje povolení určitých akcí s objekty (např. soubory) pouze některým subjektům (uzlům). Většina typů objektů a akcí (např. „uzel“ – „zpracovat úlohu“) náleží do aplikační vrstvy, autorizace proto musí být částečně implementovaná až v aplikační vrstvě, protože je potřeba znát objekty, ke kterým se bude přidělovat přístup.

Některá oprávnění musí být kvůli zachování funkčnosti sítě implicitně povolena, např. oprávnění číst libovolný soubor je nutné pro replikaci (viz kapitolu 3.1.2). Autorizace lze v tomto případě řešit na vyšší vrstvě: soubory, které nemají být přístupné, lze zašifrovat.

Prvním velmi důležitým krokem autorizace je rozhodnout, jestli má být povoleno spojení s každým uzlem, se kterým se navazuje spojení. Toho lze použít k vytvoření uzavřené/bezpečné sítě, autorizace ji pak může chránit před nezvanými uzly.

2.5.7 Ochrana proti nezvaným uzlům

2.5.7.1 Public key infrastructure

Jediný spolehlivý způsob, jak zamezit nezvaným uzlům připojit se do sítě, je explicitně povolit přístup každému oprávněnému uzlu. To lze provést díky *pub-*

lic key kryptografii podepsáním veřejného klíče každého uzlu klíčem uznávaným všemi ostatními legitimními uzly – veřejná část klíče uzlu musí být podepsána klíčem, jehož soukromá část je známá pouze správci dané uzavřené peer-to-peer sítě a jehož veřejná část je pro účely ověření podpisů známá všem uzlům. Při každém navázání připojení k novému uzlu musí uzel ověřit podpis druhé strany a pokud je tento podpis neplatný, spojení přerušit.

K tomuto účelu lze použít například infrastrukturu/hierarchii certifikátů (*public key infrastructure*) podle standardu *X.509* [63], která je v současnosti používána k ověření identity především webových serverů a poštovních serverů při komunikaci protokolem *HTTPS*, resp. zabezpečenými verzemi protokolů *POP3*, *IMAP* a *SMTP*. Implementace je součástí knihovny OpenSSL [64] dostupné pod licencí nevyžadující zveřejnění zdrojového kódu aplikace.

Popsané řešení ochrany sítě je sice spolehlivé, ale náročné na zavedení: každému uzlu je potřeba vygenerovat podpis a předat certifikát. Proto je vhodné spíše do uzavřených sítí, potenciálně spravovaných centrálně, odmítajících jakékoli spojení s uzly, které nejsou explicitně povoleny, např. pro účel distribuce dat v rámci firmy. Společně se zavedením *public key infrastructure* by také bylo důležité zajistit *revokaci* (zneplatnění) certifikátů uzlů v případě úniku/napadení uzlu. Revokace je typicky řešena centralizovaně, pro potřeby peer-to-peer sítě by bylo nutné navrhnout rozdělení odpovědnosti za udržování záznamů o revokovaných certifikátech mezi uzly.

Implementačně může být snadnější počet nezvaných uzlů pouze omezit. Protože při velkoplošných útocích je obvykle spuštěno mnoho uzlů na každém útočníkově počítači, což při běžném provozu sítě není obvyklé, lze takové případy detekovat a snažit se množství nezvaných uzlů nějakým způsobem snížit.

2.5.7.2 Distribuovaná databáze uzlů

Detekce více uzlů na jedné síťové adrese by mohla být provedena např. takto: síť by poskytovala distribuovanou databázi připojených uzlů (zobrazení $IP \rightarrow M$, kde M je množina identifikátorů uzlů na dané adrese viděných např. za poslední 1 hodinu). V případě IPv6 není pro útočníka problém získat velké množství různých adres, proto by bylo nutné adresy sdružovat po blocích, (např. /64 blok chápat jako jednu IP adresu). Konkrétní velikost bloku však závisí na velikosti bloku, přiřazeného uživateli poskytovatelem připojení.

Při připojení k novému uzlu by byl odeslán identifikátor připojeného uzlu na uzel odpovědný za správu identifikátorů uzlů na dané adrese, např. uzel odpovědný za $hash(IP)$, kde IP je síťová adresa druhé strany. Tento uzel by měl za úkol spravovat seznam přijatých identifikátorů a poskytovat tento seznam, resp. počet prvků v seznamu. Aby se snížila pravděpodobnost, že tento uzel bude ovládaný útočníkem, bylo by vhodné tuto odpovědnost rozdělit mezi více uzlů a při kontrole provést více zápisů a dotazů na více zodpovědných uzlů.

Odpovědný uzel by pak odpověděl počtem uzlů na dané adrese. V případě, že by počet byl vyšší než nějaká daná mez, nově navázané spojení by bylo přerušeno, případně i včetně již existujících spojení s danou síťovou adresou. Kontrola počtu uzlů na jedné síťové adrese by měla být symetrická – obě strany by měly popsaným postupem ověřit adresu protistrany.

2.5.7.3 Úkolování

Další možností omezení počtu nezvaných uzlů by bylo zavedení povinných výpočtů [65]: každý uzel by byl povinný na vyžádání vyřešit nějaký výpočetně netriviální problém, jehož vyřešení může zadavatel snadno ověřit, např. pro daný výsledek hashovací funkce nalézt jeho vzor (doplnit určitý počet chybějících znaků na konec zadavatelem určeného řetězce).

Každý uzel by v nějakém časovém intervalu vždy vygeneroval a zadal každému ze svých sousedů (jiné) zadání, na jehož řešení by čekal. Pokud by nedorazilo včas, přerušil by spojení. Aby byla ochrana účinná, bylo by potřeba najít vhodnou složitost výpočtů, aby nebylo příliš plýtváno výkonem uzlů a zároveň nebyly výpočty příliš jednoduché i pro útočníka.

2.5.7.4 Ztížení generování identity

Podobně by bylo možné ztížit vytvoření nové identity omezením tvaru identifikátoru, před prvním připojením k síti by musel uzel jednorázově vygenerovat vhodný pár klíčů, např. takový, aby hash veřejného klíče (identifikátor) končil určitým počtem nul – inspirováno systémem potvrzování transakcí v síti Bitcoin. Opět by bylo možné nastavit složitost tohoto procesu, aby nebyla příliš vysoká i pro legitimní uživatele sítě.

2.5.7.5 Pasivní ochrana

Proti útočníkově snaze ovlivňovat směrování nebo výsledky dotazů na zdroje je možné se chránit zčásti i pasivně např. implementací tolerance chyb a případného opakování dotazu. Při obdržení odpovědi „zdroj nebyl nalezen“ je možné se dotázat znovu, tentokrát jiného uzlu, případně poslat více požadavků různými směry současně. Při malém počtu zlých uzlů lze takto jejich vliv na fungování sítě omezit.

2.5.7.6 Shrnutí

Kromě explicitního povolení každého legitimního uzlu úplně neochrání síť před zlými uzly žádná z popisovaných možností. Protože určování důvěryhodnosti uzlů (*trust management*) v peer-to-peer sítích je složitý problém nad rámec této práce, řešený více způsoby s různou úspěšností [65, kapitola 3.12], ponecháme jeho návrh a implementaci jako možné rozšíření.

2.5.8 Směrování a přeposílání zpráv v overlay network

Pro přenos zpráv v rámci sítě je nutné rozhodnout, jakým způsobem bude probíhat směrování a přeposílání zpráv.

Triviálním řešením by bylo připojit se k uzlu, kterému chceme zprávu odeslat, a data zprávy poslat přímo. Tím využijeme pouze směrování a přeposílání paketů protokolem IP (bez jakékoli vlastní nadstavby). Pokud by se jednalo o přenos většího množství dat (např. obsahu nějakého souboru), bylo by toto řešení vhodné, ale v případě jednorázového poslání zprávy je tento způsob neefektivní, protože navázání spojení a případná autentikace typicky trvá déle než přeposlání zprávy přes více uzlů po již navázané cestě. Navíc bychom pro přímé spojení

potřebovali znát transportní adresu (IP adresu a port) daného uzlu, dotaz na ni by ale musel proběhnout nepřímo – přes ostatní uzly.

Abychom se vyhnuli navazování nových spojení, lze využít již existujících – těch, které navázala a udržuje topologie sítě (*overlay network*). V tomto případě musíme zajistit směrování a přeposílání zpráv ručně. Topologie sítě zaručuje nalezení cesty k libovolnému uzlu, známe tedy, kterým směrem (tj. kterému sousednímu uzlu) máme zprávu poslat (v případě topologie Chord je to ten sousední uzel, jehož adresa leží první proti směru hodinových ručiček před cílovou adresou).

Přeposílání může probíhat dvěma způsoby: stavově nebo bezstavově.

Bezstavové přeposílání zpráv je podobné přeposílání paketů, které provádí na síťové vrstvě protokol IP – při přijetí zprávy určené jinému uzlu je zpráva přeposlána správným směrem a dále tímto uzlem není zpracována. Protože uzly na cestě neuchovávají žádné informace o přeposlané zprávě, nelze zajistit spolehlivé doručení zprávy – případnou spolehlivost typicky řeší odesílatel znovuposláním. V případě stavového přeposílání zpráv mezilehlý uzel naopak čeká na odpověď (nebo potvrzení přijetí), případně na chybu, kterou předá uzlu na cestě směrem k odesilateli.

Stavové přeposílání zpráv je použito v případě RPC, které také zajišťuje potřebnou spolehlivost, navíc podle důležitosti konkrétního typu RPC umožňuje různé úrovně spolehlivosti pro různé RPC funkce (důležité RPC funkce mohou v případě chyby např. poslat dotaz znovu, po méně optimální cestě). Kromě spolehlivosti může mít stavové přeposílání zpráv i tu výhodu, že odesílatel nemusí znát adresu (id) příjemce, např. po stažení souboru s identifikátorem `id` by bylo možné informovat uzel odpovědný za `id` zprávou „mám soubor s identifikátorem `id`“ adresovanou (neznámému) uzlu odpovědnému za `id` (viz kapitolu 3.1.2).

Pokud je zpráva požadavkem (příjemce na ni posílá odpověď), musí i odpověď nalézt cestu zpět. V případě stavového přeposílání je zpáteční cesta pevně daná (je stejná jako cesta požadavku, pouze odpověď je posílána opačným směrem), v případě bezstavového přeposílání se zpáteční cesta může lišit.

V programu jsou implementovány všechny tři varianty: pro přenos souborů se navazuje přímé spojení, bezstavové přeposílání je použito pro jednosměrné zprávy (`message_envelope`, např. registrace uzlu jako hostitele souboru na zodpovědném uzlu) a stavové přeposílání je použito pro RPC (např. na vyhledání následníka v topologii Chord).

2.5.9 Šifrování

Pro zamezení možnému odposlechu spojení mezi uzly je vhodné přenášená data šifrovat. Na rozdíl od *public key* kryptografie, použité pro ověření autenticity uzlu, je z důvodu efektivity pro tento účel vhodnější „symetrické“ šifrování, které pro zašifrování (před odesláním) i dešifrování dat (po přijetí) používá stejný klíč (*session key*). V praxi nejrozšířenější jsou algoritmy rodiny AES (*Advanced Encryption Standard*) [66].

TLS (*Transport Layer Security*) [67] je nejpoužívanější [68] protokol zajišťující zabezpečení přenosů na Internetu. Použití TLS řeší společně autentizaci, šifrování a kontrolu přenášených dat před poškozením nebo změnou třetí stranou, navíc umožňuje použití certifikátů *public key infrastructure*, čímž může vyřešit v uzavřených sítích i problém nezvaných uzlů.

Aby nemohlo dojít ke zpětnému dešifrování minulé komunikace v případě úniku klíče (zajištění tzv. *perfect forward secrecy*), je nutné, aby *session* klíč byl pro každé spojení jiný. K vytvoření takového klíče (*sdíleného tajemství*) lze použít Diffie-Hellmanův algoritmus [69].

Dále je potřeba zamezit útokům typu *man-in-the-middle*, kdy komunikace probíhá přes útočníka, který se při navázání spojení vydává za druhou stranu, poté, co získá *session key* pro komunikaci s oběma stranami, přijatá data z jedné strany dešifruje jedním *session* klíčem, zašifruje druhým a pošle druhé straně, mezitím případně data pozmění a/nebo uloží. Pro ochranu je potřeba ověřit, že *session key*, kterým je komunikace šifrována, je na obou stranách stejný – ten, na kterém se komunikující strany dohodly, např. poslat druhé straně k ověření podpis *session* klíče (nebo parametrů algoritmu Diffie-Hellman použitých k jeho vygenerování) svým soukromým klíčem používaným pro autentikaci.

Stejně jako v případě *public key* kryptografie lze použít knihovnu OpenSSL, kde je implementován Diffie-Hellmanův algoritmus i autentikace druhé strany.

Přestože lze tímto způsobem zajistit ochranu před odposlechem a útoky zvenku, neřeší toto použití šifrování ochranu před zlými uzly v síti, ani nezajišťuje anonymitu komunikujících uzlů. Protože se na provozu sítě podílí všechny uzly – např. při vyhledávání uzlu odpovědného za daný zdroj je dotaz zpracován neznámými uzly na cestě – útočník s dostatečně mnoha uzly uvnitř sítě může stále sledovat, měnit nebo blokovat zprávy, které přes něj procházejí a zjistit, které uzly spolu komunikují (viz kapitolu 2.5.5.1).

Anonymitu v síti by bylo možné zajistit alespoň částečně – při přeposílání zprávy známému uzlu lze mezilehlým uzlům (případně i cílovému) skrýt identitu odesílatele. Bylo by možné implementovat *source routing* (odesílatel zná cestu k příjemci, celou ji uvede do odesílané zprávy; uzly na cestě pak pouze přeposílají zprávu na další uvedenou adresu) se zapouzdřením a zašifrováním každého kroku klíčem daného uzlu na cestě. Uzly na cestě by postupně rozbalovaly, dešifrovaly a přeposílaly přijaté *pakety*, aniž by znaly odesílatele, příjemce nebo obsah zprávy – pouze znají uzel, od kterého *paket* přijaly, a uzel, kterému ho mají přeposlat – podobně jako v síti Tor [40]. Aby bylo možné takový *paket* odeslat, odesílatel by musel znát veřejné klíče všech uzlů, přes které by chtěl zprávu poslat, při jejich vyhledávání a získávání by však zlé uzly mohly požadavky zachytit a ovlivnit (např. při vyhledávání uzlů ochotných přeposílat data podstrčit zlé uzly) a při samotném přeposílání z dostupných dat pak zjistit, které uzly spolu komunikují. Tor udržuje centrální databázi tzv. *relay* uzlů (přeposílajících *pakety*) – několik autoritativních serverů provozovaných správci sítě Tor sbírá informace o *relay* uzlech a zveřejňuje jejich seznam [70] – při sestavování cesty jsou stažené (a tím dostupné k výběru cesty) všechny veřejné klíče *relay* uzlů, nedotazuje se na ně jednotlivě. Pro účel čistě peer-to-peer sítě by bylo nutné podobnou databázi vytvářet a distribuovat decentralizovaně.

Kvůli netrivialitě návrhu anonymizace v peer-to-peer síti a bezpečnostním opatřením nad rámec této práce ponecháme problematiku anonymizace a šifrování pouze jako další možné rozšíření.

2.6 Prezentační vrstva (L6)

Prezentační vrstva přizpůsobuje data potřebě aplikace, případně může provádět jejich kompresi a dekompresi.

2.6.1 Serializace

Uzly v síti pracují (lokálně) se zprávami v nativním formátu – např. pořadí bajtů v čísle (*endianness*) je závislé na platformě uzlu – navíc mohou zprávy obsahovat netriviální datové typy, např. množiny nebo asociativní pole. Aby spolu mohly uzly komunikovat, je potřeba formát posílaných dat sjednotit včetně určení, jak mají být netriviální datové typy převedeny do/z podoby řetězce, který je možné poslat po síti.

Pro serializaci můžeme použít buď textový, nebo binární formát uložení dat. Textové formáty – např. XML (*Extensible Markup Language*) [71] nebo JSON (*JavaScript Object Notation*) [72] – mají výhodu čitelnosti, která se může hodit při ladění programu, binární formáty jsou vhodnější pro běžný provoz, protože mají menší režii. Pro možnost snadnějšího ladění i dosažení dobrého výkonu je vhodné umožnit přepínání mezi více formáty.

XML nabízí dobrou flexibilitu – možnost definovat vlastní značky (*elementy*) a strukturu celého XML dokumentu a formálně definovat použitý jazyk v tzv. *schématu* – ale kromě samotných dat ukládá i data, která nejsou nutná (např. zavření *elementu* obsahuje název zavíraného *elementu*). Protože XML dokumenty obsahují většinou tištitelné znaky a názvy elementů i atributů se často opakují, lze je dobře komprimovat. Vhodnější by bylo použít šetrnější binární zápis XML, např. *EXI* (*Efficient XML Interchange*) [73]. V případě použití XML pro serializaci zpráv by kvůli/díky předem známým typům zpráv byly informace o vnitřní struktuře zprávy nadbytečné – pro deserializaci stačí znát typ zprávy a obsah jednotlivých polí. V případě tzv. *schema-aware* kompresního algoritmu, využívajícího znalosti možné struktury komprimovaného dokumentu, je možné velikost XML zpráv ještě dále snížit.Pro implementaci je možné použít knihovnu *EXIP* [74], která podporuje *schema-aware* kompresi a nabízí rozhraní podobné *SAX* – při parsování o každé události (začátku/konci elementu, ...) informuje knihovna aplikaci zavoláním nastavené callback funkce.

Další možností by bylo použití úspornějšího formátu. V případě potřeby textového formátu je šetrnější například *JSON*, jinak bychom mohli použít nějaký binární (*TLV* (*type length value*)) formát. Podobně jako v případě XML, datový model formátu *JSON* lze také ukládat binárně ve formátu *CBOR* (*Concise Binary Object Representation*) [75]. Opět je možné použít existující implementaci, např. *libcbor* [76] (C API, podporuje přístup jako k asociativnímu poli) nebo *cbor-cpp* [77] (C++ API, pouze *SAX-like* interface).

Protože samotné zprávy je vhodné v aplikaci definovat jako třídy, je nutné implementovat jejich serializaci (případně převod do/z formátu XML/JSON za použití vhodné knihovny) v jazyce C++. Bylo by možné implementovat vlastní řešení, ale již existující knihovny mohou serializaci usnadnit, použít lze například *Protocol Buffers* [78] nebo *boost::serialization* [79]. Knihovna *Protocol Buffers* vyžaduje napsání speciální definice zpráv, kterou je nutné odděleně přeložit, čímž vznikne zdrojový kód tříd (C++), který je teprve možné použít v aplikaci. V pří-

padě *boost::serialization* se definuje obsah zpráv přímo v kódu – uvnitř třídy v C++ stačí přidat/implementovat serializační metodu a třídu zaregistrovat jako serializovatelnou. Pro implementaci (de)serializace byla z důvodu jednodušší integrace použita knihovna *boost::serialization*.

Knihovna *boost::serialization* zajišťuje typovou kontrolu a umožňuje implementaci vlastních (de)serializátorů, lze tedy pro přenos dat použít libovolný formát zápisu, navíc obsahuje implementaci (de)serializátorů do/z několika vlastních formátů, např. textový, XML, *non-portable native binary archive* a přenositelný binární formát (bez podpory ukládání floating-point čísel). Pro použití úplného přenositelného binárního formátu podporujícího všechny datové typy lze použít kód třetí strany: *EOS Portable Archive* [80]. Protože však floating-point čísla žádná zpráva neobsahuje, postačí vestavěné (de)serializátory.

Po převedení zprávy do podoby řetězce je dále potřebné jednotlivé serializované zprávy odeslat druhé straně skrz *byte stream* a na druhé straně je od sebe odlišit. Pokud před každou zprávou pošleme velikost posílané zprávy, může pak příjemce snadno poznat, kdy dorazila celá zpráva, a deserializovat ji.

2.6.2 Komprese

Kromě použití efektivního formátu uložení zpráv při serializaci je možné objem přenášených dat zmenšit jejich kompresí. Komprimovat lze buď celé spojení (serializované zprávy i přenášená data souborů), nebo samotné soubory či zprávy.

V případě efektivního uložení zpráv nemá komprese převažujících malých zpráv příliš vysokou účinnost, navíc by další krok při zpracování zprávy před odesláním přidával další zpoždění. Komprese celého spojení nebo autentikačních zpráv by navíc mohla (v případě šifrovaného přenosového kanálu) prozradit případnému útočníkovi, jak moc jsou data podobná dříve odeslaným datům, z čehož by mohl útočník odvodit, jaká data jsou přenášena – pokud dokáže útočník ovlivňovat druhou stranou odesílaná data a měřit velikost odesílaných zpráv, z kompresního poměru může zrekonstruovat řetězec obsažený v odesílaných zprávách (útok *CRIME* [81]). Kompresi zpráv ani celého proudu dat jsme proto neimplementovali.

V případě komprese obsahu souborů by případný únik kompresního poměru neměl mít pro útočníka žádný význam, nebo by k němu vůbec nemělo dojít – soubory v síti jsou přístupné všem uzlům (viz kapitolu 3.1.2) a daný soubor typicky stahuje každý uzel pouze jednou, nedochází tedy k opakovaným požadavkům potřebným pro provedení útoku *CRIME*.

Soubory je možné komprimovat buď při přenosu, nebo už při uložení. Protože některé soubory mohou být již komprimované, jejich dalšího zmenšení bychom nedosáhli a pouze bychom plýtvali výkonem uzlů. Z tohoto důvodu necháme kompresi až na aplikační vrstvě jako možné rozšíření (komprese při importu souboru do sítě), nebo na uživatelích (import již komprimovaného souboru).

2.7 Aplikační vrstva (L7)

Aplikační vrstva popisuje funkce a chování aplikace. Protože se nejedná o analýzu možných řešení, ale spíše o vlastní návrh aplikace využívající síť popsanou

v rámci této analýzy, popíšeme aplikaci a její funkce v samostatné kapitole (viz kap. 3).

2.8 Existující peer-to-peer sítě

Nejrozšířenější peer-to-peer sítě jsou většinou jednoúčelové, nejčastěji navržené pro ukládání a/nebo sdílení souborů, např. *BitTorrent* [36], *eDonkey* [37], *Gnutella* [38] a *IPFS* [39].

Dalším rozšířeným typem peer-to-peer sítí jsou sítě, které *pouze zprostředkovávající spojení mezi uzly*, příp. zajišťují jejich anonymitu, např. *Tor* [40] a *I2P* [41]. Tyto sítě lze sice využívat k provozování různých typů aplikací, neposkytují však funkce, které by vývoj a provoz těchto aplikací usnadnily (např. decentralizované datové struktury pro ukládání souborů nebo fronty úloh), ty je nutné implementovat samostatně.

Distribuované výpočty jsou typicky centralizované, např. *SETI@home* [42], *Rosetta@home* [43] a *Great Internet Mersenne Prime Search* [44]. Jediné použití čistě peer-to-peer sítě pro výpočty jsme nalezli v případě sítě *Ethereum* [45] kombinující kryptoměnu a distribuovaný virtuální stroj. Přestože tato síť provádí výpočty distribuovaně, není určena pro *výkonové výpočty*, ale mnohonásobným ověřením výpočtů na více uzlech má zajistit jejich spolehlivost.

2.8.1 Odlišnosti navrhované sítě

Námi navrhovaná síť by měla být použitelná současně více typy aplikací a měla by navíc umožňovat jejich snadný vývoj poskytováním podpůrných funkcí (uložení souborů, příp. další decentralizované datové struktury).

3. Aplikace

Díky infrastruktuře popsané v předchozí kapitole (síť a rozdělení odpovědnosti za provoz mezi jednotlivé připojené uzly) můžeme implementovat aplikaci poskytující funkcionalitu, kterou popíšeme v této kapitole. Výslednou síť zajišťující spojení mezi uzly a využitelnou pro různé druhy aplikací jsme pracovně nazvali *minet* (*multipurpose interconnection network*).

3.1 Implementované funkce

Pro praktické ověření funkčnosti navržené infrastruktury a možnost měření jejího chování jsme vytvořili implementaci klienta sítě *minet* – program *min*.

Klient sítě implementuje decentralizované ukládání souborů s automatickým vytvářením replik pro zajištění spolehlivosti a správu centralizované *work queue* pro distribuované zpracování úloh.

3.1.1 Ukládání souborů

Jednou z primárních funkcí této sítě je ukládání souborů. Klient sítě musí umožňovat vložení souboru do sítě a jeho následné získání ze sítě.

3.1.1.1 Umístění

Pro uložení každého souboru je nutné zvolit vhodné umístění – uzel, na kterém má být uložen a který ho má poskytovat.

Soubor je možné uložit přímo na odpovědném uzlu (určeném použitou topologií sítě, viz kapitolu 2.5.3), ale vhodnější je spíše jej umístit na uzlu, který o něj *má zájem*. Abychom toho dosáhli, musíme oddělit metadata od obsahu souborů – odpovědný uzel musí kvůli vyhledání vědět, kde je soubor uložen, samotná data však mohou být uložena na libovolném uzlu nebo na více uzlech. Pokud tedy uživatel prostřednictvím svého uzlu stáhne ze sítě nějaký soubor, může daný soubor poskytovat ostatním pro další stahování. Tím zároveň vylepší rozložení zátěže mezi uzly – čím žádanější soubor, tím více bude dostupných jeho kopií.

Protože při nahrávání souboru do sítě ještě nevíme, které uzly budou *mít o soubor zájem*, první kopii můžeme nahrát na libovolný uzel, např. na uzel, který je odpovědný za uložení metadat daného souboru. Automatické udržování uložených kopií souboru je dále popsáno v kapitole 3.1.2.

3.1.1.2 Adresování

Dále je nutné rozhodnout, jak přiřadit souborům identifikátory-adresy: buď podle názvu (*location-addressed storage*), nebo podle obsahu (*content-addressed storage*). Adresování podle názvu (identifikátorem souboru je hash jeho názvu) umožňuje vyhledávání souborů podle názvu, ale je nutné vyřešit případné duplicity (rozlišení stejně pojmenovaných souborů), např. přidat každému názvu souboru prefix – jméno autora, případně cestu (opět je nutné zajistit i jejich jednoznačnost).

Protože v síti mohou být připojeny *zlé* uzly, je vhodné mít možnost ověřit, že stažený soubor je nepoškozený a že je to ten, který byl požadován – což nelze zaručit, pokud identifikátor neodvodíme od obsahu daného souboru. Z tohoto důvodu je vhodnější adresování podle obsahu, kdy identifikátorem souboru je hash jeho obsahu.

Dostatečně dlouhý hash souboru s vysokou pravděpodobností jednoznačně určuje daný soubor. Hashování navíc poskytuje pasivní ochranu před zlými uzly a chybami přenosu – výpočtem hashe staženého souboru lze ověřit, že soubor nebyl při přenosu poškozen nebo nahrazen jiným souborem, např. některým *zlým* uzlem v síti.

3.1.1.3 Velikost adresového prostoru

Také je nutné zvolit vhodnou velikost prostoru identifikátorů souborů a uzlů. Pokud bychom zvolili příliš krátké adresy, byla by vyšší pravděpodobnost kolizí (dva soubory se stejnou adresou), naopak příliš dlouhé adresy by plýtvaly přenosovou kapacitou sítě. Velikost adresového prostoru souvisí s hashovací funkcí použitou pro přiřazení adresy souboru. Pokud použijeme některou z běžně používaných hashovacích funkci, umožníme vyhledávání souborů v síti podle známého hashe (např. nalezeného na webu).

Mezi nejrozšířenější hashovací funkce patří MD5 a SHA-1. [82]

Výstup funkce MD5 má délku 128 bitů, MD5 však neposkytuje dostatečnou bezpečnost – existují kolize, k danému souboru lze snadno vytvořit jiný soubor se stejným hashem [83].

Hashovací funkce SHA-1 má 160bitový výstup, což by měla být dostatečná délka pro použití jako identifikátoru: podle *narozeninového paradoxu* [84] by pro kolizi s pravděpodobností alespoň 10^{-9} bylo potřeba více než 10^{19} různých vstupů (souborů). Úmyslné kolize sice mohou být v budoucnosti potenciálním problémem [85], ale nalezení kolize je stále v praxi příliš nákladné.

Kromě použití SHA-1 jako *kontrolního součtu* k ověření, že nedošlo k poškození souboru při přenosu ([86]), je tato hashovací funkce použita také k vytvoření identifikátoru sdíleného obsahu (tzv. *info_hash*) `.torrent` souboru v síti BitTorrent [36] a interně v systému správy verzí Git [87] pro identifikaci objektů (např. *commit*).

Pro implementaci jsme použili hashovací funkci SHA-1, případná budoucí změna použité funkce (např. na SHA-256) je triviální.

Díky volbě *kryptograficky bezpečné* hashovací funkce, jejíž výstup „vypadá jako náhodná data“, je přiřazení identifikátorů souborům a uzlům rovnoměrné. Na hodnoty výstupu hashovací funkce (bez znalosti vstupu) totiž lze nahlížet jako na hodnoty z rovnoměrného rozdělení, protože každý výstup má stejnou pravděpodobnost. Každý uzel by tedy měl být odpovědný za přibližně stejný počet souborů.

3.1.2 Replikace

Aby nedocházelo ke ztrátě/nedostupnosti dat při odpojení uzlu od sítě, je nutné v síti vytvářet a udržovat více kopií (*replik*).

3.1.2.1 Vytváření a umístění replik

Stejně jako v případě jiných úkolů je potřeba zvolit uzel, který má být odpovědný za kontrolu a případné vytváření dalších replik. Protože uzel odpovědný za identifikátor určitého souboru uchovává metadata o tomto souboru – především seznam uzlů, které daný soubor poskytují – je vhodné odpovědnost za replikaci tohoto souboru přiřadit právě tomuto uzlu.

Pro repliky metadat (případně i obsahu) souborů je nutné zvolit vhodné uzly, na kterých mají být uložena. Abychom minimalizovali množství přenášených dat, je vhodné další repliky ukládat na takových uzlech, které převezmou odpovědnost za soubor, pokud selže odpovědný uzel. V případě topologie Chord, kde za každý soubor je odpovědný uzel následující za identifikátorem souboru, získá po selhání odpovědného uzlu odpovědnost jeho následník (tj. druhý následník identifikátoru daného souboru). Proto je výhodné repliky ukládat na následnících odpovědného uzlu.

Protože není možné spolehlivě předpovídat selhání uzlů, musí každý uzel periodicky procházet soubory, za které je odpovědný, a kontrolovat, jestli je dostupný dostatečný počet replik. V opačném případě musí uzel zajistit replikaci na svého následníka (své následníky): poslat jim metadata a požádat je o stažení souboru (z uzlů obsažených v metadatech). Protože množina souborů, za které je daný uzel odpovědný, a *náhradní* uzel (resp. více *náhradních* uzlů), který přebere odpovědnost za daný identifikátor v případě selhání odpovědného uzlu, jsou závislé na použité topologii, musí topologie poskytnout potřebná data, např. umožnit iteraci přes soubory, za které je lokální uzel odpovědný a iteraci přes *náhradníky* odpovědných uzlů.

3.1.2.2 Počet replik

Protože v síti mohou být uloženy různě důležité soubory (např. dostupnost veřejných klíčů uzlů a adresářových stromů může být důležitější než dostupnost *obyčejných* souborů), je vhodné různou důležitost souborů zohlednit při replikaci různým počtem vytvářených replik. Nastavitelnost počtu replik pro každý soubor navíc umožňuje použití sítě v různých prostředích přizpůsobením počtu replik spolehlivosti jednotlivých uzlů, např. u *uživatelských* a mobilních zařízení lze očekávat, že se budou častěji odpojovat než např. servery a síťová úložiště. Pokud se bude udržovat vyšší počet replik, sníží se pravděpodobnost, že všechny repliky zaniknou *najednou* – dříve, než dojde k vytvoření dalších kopií.

Protože počet replik souboru není součástí jeho obsahu a změna souboru (přidání požadovaného počtu replik) by způsobila změnu jeho identifikátoru, je nutné ukládat počet replik (a případné další informace o souboru) odděleně od obsahu souborů do samostatného úložiště.

3.1.3 Úložiště metadat

Pro uchovávání metadat souborů a pro lokální potřeby klientské aplikace (např. navázání spojení s posledně připojenými uzly po restartu aplikace) je důležité zajistit vhodné úložiště. Metadata o souborech a informace o ostatních uzlech je vhodné ukládat tak, aby bylo snadné v nich vyhledávat. Protože mezi entitami (např. „uzel“, „soubor“ a „IP adresa“) jsou přirozené vztahy (např. „uzel

poskytuje soubor“ a „uzel má IP adresu“), je pro uložení vhodné použít relační databázi.

Klasické systémy řízení báze dat (*RDBMS*) např. MySQL [89] nebo PostgreSQL [90] jsou zbytečně náročné – mají vysoké požadavky na hardware (úložiště a operační paměť) a navíc typicky vyžadují instalaci. Protože nepotřebujeme žádné jimi nabízené *pokročilé* funkce – např. více uživatelských účtů s různým oprávněním pro přístup k jednotlivým tabulkám ani efektivní paralelní přístup do databáze z více aplikací (se zamykáním záznamů/tabulek) – postačí nám jednodušší *embedded RDBMS*, který lze slinkovat přímo do aplikace, např. SQLite [91].

SQLite je open-source knihovna zveřejněná jako volné dílo (licence *public domain*), používaná je především jako databáze na webových serverech s malou návštěvností (např. v PHP je při překladu podpora SQLite defaultně zapnuta [93]) a mobilními aplikacemi pro ukládání dat, např. v iOS je SQLite použita jako úložiště SMS, poznámek a kalendáře. [92] Pro definici schématu a dotazování je použit jazyk SQL, SQLite provádí transakce splňující vlastnosti ACID.

Vhodné uložení metadat o souborech a uzlech umožňuje dobrou škálovatelnost sítě a efektivní sdílení souborů. Proto je vhodné, aby na odpovědném uzlu byl přístupný seznam všech uzlů, které soubor poskytují, pro aplikace používající síť však může být užitečné ukládat o souborech dodatečné informace. Kromě zohlednění důležitosti souboru uložením počtu jeho replik lze například ukládat u souboru i jeho původ – identifikátor uzlu, který soubor do sítě nahrál. Tento uzel by pak mohl mít k danému souboru zvláštní oprávnění, například by mohl požádat ostatní uzly o jeho smazání.

Lokální databázi lze použít také jako úložiště záznamů distribuované databáze, viz kap. 3.2.3.

3.1.4 Centralizovaná distribuce úloh

Peer-to-peer síť může výrazně usnadnit a zefektivnit distribuované zpracování úloh. Při zadávání úloh uzlům (typicky z jednoho počítače) je z principu nutné odeslat/poskytnout zadání mnoha uzlům, což může být *bottleneck*. Jako zadání úlohy budeme chápat určení algoritmu, který poskytne výsledek a vstupní data.

Jednotlivá zadání úlohy si často bývají podobná – některé soubory jsou společné pro více zadání. Program, který má provést zadaný výpočet a poskytnout výsledek, bude v mnoha zadáních identický a také jeho vstupní data mohou být společná, např. v případě renderování (raytracingu) 3D animace – scéna a objekty jsou stejné po mnoho snímků, mění se pouze pozice objektů v prostoru. Pokud by se data pro každé zadání lišila pouze částečně, mohla by být uložena v *Merkle tree* (viz kap. 3.2.2), díky čemuž by společné soubory nebylo nutné do sítě nahrávat vícekrát.

Problém distribuce úloh je obvykle řešen centrální frontou úloh čekajících na vyřešení (tzv. *task queue*). Jednotliví řešitelé z fronty odebírají zadání úloh a provádí je. Aby nedocházelo k vícenásobnému provedení nějaké úlohy, je *task queue* typicky spravována centralizovaně – na jednom počítači.

Přestože je rychlost přidělování úloh (odebírání z fronty) omezena výkonem a konektivitou jednoho počítače, jedná se o implementačně nejjednodušší řešení, které navíc umožňuje efektivnější přidělení úloh – pokud zadavatel zná vlastnosti

uzlů-řešitelů, může jim přiřadit úlohy např. podle jejich dostupných systémových prostředků a aktuálního zatížení – a díky přímému spojení zadavatele a řešitele minimalizuje případnou prodlevu před začátkem zpracování úlohy.

Task queue lze provozovat také decentralizovaně (viz kap. 3.2.5), ale implementace je náročná a nad rámec této práce, proto jsme implementovali pouze centralizovanou variantu *task queue*.

3.1.4.1 Zabezpečení

Pro zpracování úlohy musí řešitel spustit program daný zadavatelem, je tedy nutné, aby zadavateli důvěřoval, nebo alespoň zajistil, aby nebyl program škodlivý. Proto je vhodné, aby správce uzlu-řešitele explicitně povolil spouštění programů, kterým důvěřuje. Dále by bylo vhodné, aby byly programy prováděny s minimálním nutným oprávněním v uzavřeném prostředí (*sandbox*u), např. pouze s přístupem k výpočetní kapacitě (CPU, případně i GPU např. přes rozhraní OpenCL) a omezenou možností komunikace s okolím umožňující právě odeslání výsledku úkolu. V případě uzavřené sítě, kde zadavatel je typicky důvěryhodným správcem, tato opatření nejsou nutná.

3.2 Další možná rozšíření

V této části popíšeme návrh funkcí, jejichž implementace je nad rámec této práce. Tyto funkce ponecháme jako možná rozšíření vzniklé aplikace.

3.2.1 Dělení souborů

Dosud jsme předpokládali, že každý soubor bude v síti uložen vcelku. Aby se však zátěž lépe rozdělila mezi více uzlů (např. sdílením části stahovaného souboru před dokončením stahování celého souboru), můžeme jednotlivé soubory rozdělit na menší části (bloky), které by byly nahrány na více uzlů. Tak to probíhá např. v případě sítě BitTorrent [36].

Pokud bychom soubory dělili, nebyl by celý soubor snadno vyhledatelný podle hashe (protože hash celého souboru není jeho adresou, resp. adresou jednotlivých částí), ale bylo by nutné zajistit vyhledání souboru (podle hashe) jiným způsobem (viz kapitolu 3.2.3) a nalezení jeho jednotlivých částí, např. v *seznamu souborů*.

3.2.2 Adresářové stromy

Při sdílení dat často není praktické sdílet jednotlivé soubory, ale více souborů ve složce, případně celý adresářový strom. Toho lze uživatelsky dosáhnout uložením adresářového stromu do archivu, který pak lze sdílet jako jeden soubor, ale tento přístup není optimální, protože sdílená data lze uložit efektivněji. Pokud je nějaký soubor obsažen ve více adresářových stromech, nemusí být v síti uložené jeho zbytečné kopie, ani při nahrávání složky do sítě není nutné uploadovat ty soubory a složky, které jsou již v síti dostupné.

Pro implementaci efektivnějšího uložení adresářových stromů můžeme použít tzv. *Merkle tree* [88]. *Merkle tree* je strom, který v každém vrcholu obsahuje hash

a hash v každém vnitřním vrcholu je *kombinací* potomků (např. hash konkatenace hashů potomků). Pokud budeme předpokládat, že nedochází ke kolizím, každý vnitřní uzel včetně kořene jednoznačně určuje celý podstrom. Do listů *Merkle tree* můžeme uložit hashe souborů, vnitřními vrcholy můžeme reprezentovat složky, hash v kořeni pak bude jednoznačně určovat celý adresářový strom včetně jeho struktury. Kromě samotných hashů můžeme v *Merkle tree* ukládat i další informace, např. vrcholy mohou obsahovat názvy souborů nebo složek.

3.2.2.1 Merkle tree a dělení souborů

V případě dělení souborů před jejich ukládáním do sítě (viz kapitolu 3.2.1), musíme zajistit jejich následné sloučení. Seznam částí děleného souboru můžeme chápat jako soubory v jedné složce, které mají být po stažení opět spojeny.

Aby měl tento seznam jednu adresu v síti, lze ho reprezentovat pomocí (jednoúrovňového) *Merkle tree*. Tento seznam však bude mít jiný hash než výsledný stažený soubor po sloučení částí, proto je nutné důvěřovat autorovi seznamu, že po sloučení bude hash odpovídat požadovanému souboru. Návrh vyhledávání mj. seznamů souborů podle hashe výsledného souboru je popsán v kapitole 3.2.3.

3.2.2.2 Dělení Merkle tree

Samotný *Merkle tree* můžeme uložit buď vcelku – celý *Merkle tree* jako jeden soubor – nebo decentralizovaně – každý vnitřní vrchol (seznam jeho potomků) uložit do samostatného souboru.

Mekle tree uložený v jednom souboru umožňuje rychlé zobrazení obsahu celého adresářového stromu, ale v případě velkých adresářových stromů obsahujících mnoho položek by mohl být problém s jeho velikostí (např. pokud by *Merkle tree* obsahoval celý souborový systém – zálohu). Navíc v případě potřeby jeho aktualizace by musel být do sítě nahrán celý znovu i pokud by byl změněn pouze jeden soubor.

V případě děleného *Merkle tree* je možné úsporněji procházet velké adresářové stromy, případně do jednoho adresářového stromu vložit druhý (jako list nového *Merkle tree* vložit kořen-hash již existujícího), což umožňuje v případě zálohování vytvářet *snapshoty* nebo při vytvoření a zveřejnění nové verze adresářového stromu odkázat na předchozí verzi. Aktualizace děleného *Merkle tree* při změně souboru vyžaduje pouze aktualizaci jedné větve stromu (té, na které změněný soubor leží, od listu-souboru ke kořeni), čímž vznikne nový *Merkle tree*.

3.2.2.3 Měnitelné adresářové stromy

Merkle trees jsou z principu neměnné, ale pro různé účely je někdy vhodné, aby autor mohl adresářový strom aktualizovat při zachování jeho *identity*. To by umožnilo např. snadné získání seznamu souborů aktuálně sdílených nějakým uzlem (jako *Merkle tree*) nebo vytváření aktualizovatelných adresářových stromů, např. pro implementaci *World Wide Web* nebo souborových systémů nad peer-to-peer sítí.

Aby nemusel být autor adresářového stromu stále připojen (pro poskytování seznamu sebou sdílených souborů), je možné uložit daný seznam jako soubor do sítě. Abychom předešli falšování takových seznamů souborů, lze podepsat daný

seznam soukromým klíčem autora. Místo podepisování celých seznamů stačí pouze zveřejnit (nahrát do sítě) adresářový strom (*Merkle tree*), podepsaný hash jeho kořene a svůj veřejný klíč. Každý uzel pak může spravovat svůj zveřejněný strom souborů a ostatní uzly mohou ověřit jeho autenticitu.

Aby byl *adresářový strom spravovaný uzlem* (resp. jeho aktuální verze) vyhledatelný podle nějakého pevného identifikátoru, měl by být dostupný při znalosti identifikátoru autora. Protože hash seznamu souborů (*Merkle tree*) neodpovídá identifikátoru autora, nelze ho vyhledat jako běžný soubor – před stažením samotného seznamu je nutné získat jeho adresu.

Nalezení adresy tohoto seznamu může být zajištěno např. vyhledáváním v distribuované databázi (viz kapitolu 3.2.3), při každé změně zveřejněného adresářového stromu musí autor aktualizovat informace v databázi, ta by měla poskytovat vždy tu nejnovější uloženou hodnotu a akceptovat pouze platné (podepsané) požadavky na aktualizaci. Uživatelé pak budou moci zkontrolovat, že obsah *Merkle tree* pochází od daného autora, zlé uzly pak mohou v nejhorším případě pouze poskytovat starší verzi nebo požadavky blokovat (viz kapitolu 2.5.5.1).

3.2.3 Distribuovaná databáze

Pro možnost vyhledávání souborů a adresářových stromů podle různých parametrů (autor, název, velikost, ...) a pro obecné využití aplikacemi nad peer-to-peer sítí je možné implementovat a použít distribuovanou databázi. Popíšeme, jak může taková databáze v peer-to-peer síti fungovat.

3.2.3.1 Databázový model

Provozovat relační databázi na mnoha (nedůvěryhodných) uzlech je netriviální problém, pro účely aplikace to však není nutné, postačí nám nejjednodušší možná forma databáze – *key-value store* – úložiště párů řetězců (*klíč*, *hodnota*). Každý pár je jedním záznamem v databázi.

Záznam v databázi reprezentuje relaci „má hodnotu“ – záznam (x, y) reprezentuje „x má hodnotu y“. Pro některé klíče je vhodné uchovávat pouze jednu hodnotu (kardinalita 1:1) – např. *adresářový strom spravovaný uzlem* může být v dané chvíli pouze jeden – v jiných případech je vhodné pro jeden klíč uchovávat více hodnot (kardinalita 1:N) – např. seznam souborů obsahujících v názvu slovo „prace“.

Abychom přiřadili každému záznamu odpovědný uzel, který má uchovávat a poskytovat hodnotu (resp. více hodnot) pro určitý klíč, můžeme spočítat hash daného klíče, čímž získáme identifikátor záznamu (číslo z prostoru identifikátorů uzlů a souborů), ke kterému pak najdeme odpovědný uzel podle pravidel použité topologie.

Podobně jako v případě souborů i zde je nutné záznamy replikovat, aby nedocházelo k jejich ztrátě nebo nedostupnosti při odpojení nějakého uzlu.

Pro zajištění replikace a vyhledávání musí mít libovolný uzel přístup ke čtení záznamů, zápis je však vhodné omezit. U některých záznamů (např. kořen *adresářového stromu spravovaného uzlem*) je důležité ověřit, že pochází od určitého uzlu, zvláště pokud je jeho klíč vázán na identifikátor autora. Implementace je jednoduchá – kromě hodnoty lze u daného klíče uložit i identifikátor autora záznamu a autorův podpis ukládané hodnoty. U takového záznamu pak musí být

před uložením na nějaký uzel vždy ověřena platnost podpisu. Ostatní záznamy nepotřebují ověření identity autora, aby však bylo možné snáze rozlišit a odebrat nebo ignorovat záznamy vytvořené *zlými* uzly, je vhodné vyžadovat podpis u všech záznamů.

3.2.3.2 Vlastnosti

U distribuované databáze provozované i na potenciálně nedůvěryhodných uzlech, nelze zaručit spolehlivost – může docházet ke ztrátě záznamů i k vytváření nových-nepravdivých – každému záznamu lze důvěřovat nejvýše tak, jako jeho autorovi. Databáze proto může sloužit pouze jako heuristika při vyhledávání obsahu.

Díky odvození identifikátoru záznamu hashováním by stejně jako v případě ukládání souborů mělo docházet k rovnoměrnému rozložení záznamů mezi uzly, některé klíče však mohou být více zastoupené v běžných datech – např. menších souborů je mnoho (různé textové dokumenty, zdrojové soubory, konfigurační soubory, ...) [94, Fig. 3 – Observed data] – což může vést k vyšší zátěži uzlů, které jsou odpovědné za metadata malých souborů. Proto je nutné zvolit vhodné klíče záznamů, aby nedocházelo k nadměrnému shlukování, např. zvýšit granularitu diskretizace numerických atributů (viz následující kapitolu).

3.2.3.3 Vyhledávání

Aby byla databáze prakticky použitelná, měla by kromě přímého přístupu k uloženým datům umožňovat také efektivní vyhledávání.

Vyhledávání podle celého klíče (názvu záznamu) je triviální – stačí spočítat hash klíče, nalézt uzel odpovědný za daný záznam (hash) a dotázat se ho na hodnotu (resp. hodnoty) patřící danému klíči. Přímočaré vyhledávání jiné než na rovnost klíče není možné provádět efektivně.

Vyhledávání podle numerických atributů v tradičních systémech řízení báze dat typicky používá vyhledávací stromy, ale jejich distribuované uložení na nedůvěryhodných uzlech je nevhodné. Vyhledávání podle numerických atributů lze však implementovat pomocí více záznamů v *key-value store*. Podobně i vyhledávání podle části klíče nebo hodnoty lze implementovat pomocí vyhledávání podle rovnosti.

Numerické atributy lze diskretizovat do intervalů, je však nutné zvolit jejich vhodnou velikost – příliš velké intervaly by způsobily vyšší zátěž na odpovědných uzlech (do intervalu by náleželo více souborů), malé intervaly by ztěžovaly vyhledávání – vyhledávající uzel by musel síti položit více dotazů.

Univerzálním způsobem diskretizace je zápis čísla v tzv. vědeckém formátu, např. `1.23e45` (tj. $1{,}23 \cdot 10^{45}$). Konkrétní velikosti intervalů (resp. počet platných číslic) je vhodné určit experimentálně podle zátěže odpovědných uzlů pro určitá data uložená v síti.

3.2.3.4 Indexování

Key-value store lze použít k indexování metadat souborů – pokud klíčem reprezentujeme určitou vlastnost souboru (např. v názvu je obsaženo slovo „prace“), jako hodnotu můžeme uložit identifikátor souboru, který danou vlastnost

splňuje. Při dotazu na daný klíč pak získá dotazující se uzel seznam souborů, které danou vlastnost splňují.

Aby bylo možné vyhledávat, musí již být záznamy uložené v databázi, jejich vytvoření tedy musí být zajištěno už při vložení souboru do sítě. Uzel by při nahrávání souboru mohl uložit do distribuované databáze např. název souboru, každé slovo obsažené v názvu, velikost, typ/koncovku, případně i indexovat obsah textového souboru (každé slovo z obsahu) nebo jiné údaje odvozené z obsahu souboru podle jeho typu: datový tok a délku v případě zvukových/video souborů, rozlišení v případě fotografií nebo videa, ...

Každý typ parametru musí mít přiřazen unikátní řetězec, aby klíče reprezentující různé vlastnosti nekolidovaly. Identifikátor nahraného souboru by pak mohl být uložen jako hodnota u těchto klíčů:

- `file:name:Bakalarska prace.pdf`,
- `file:extension:pdf`,
- `file:in_name:bakalarska`,
- `file:in_name:prace`,
- `file:size_estimate:1.42e6` a
- `file:size:1421864`.

3.2.3.5 Vyhledávání podle více parametrů

V navrhované distribuované databázi lze vyhledávat soubory splňující i více parametrů současně. Při dotazu na daný klíč získáme od odpovědného uzlu množinu odpovídajících hodnot, proto může vyhledávající uzel provést více dotazů. Množinu odpovídající všem dotazům lze získat jako průnik množin hodnot z jednotlivých odpovědí.

Tento postup poskytuje prostor pro optimalizace, například vhodným pořadím dobře položených dotazů lze snížit množství dat přenášených po síti, především lze zmenšit velikost odpovědí. Pokud se bude vyhledávající uzel dotazovat iterativně (ne paralelně) od nejméně pravděpodobných klíčů po pravděpodobnější (od malých množin po velké) a v dotazu uvede množinu *zajímavých výsledků* (hodnot, které vyhledávajícího zajímají – současná množina potenciálních výsledků), dotazovaný uzel může vrátit jen takovou podmnožinu hodnot z dotazu, která odpovídá dotazu.

3.2.4 Soubor pro uzel

Pro některé aplikace používající síť může být užitečné umožnit posílání zpráv i uzlům, které nejsou právě k dispozici. K tomu slouží koncept *souboru pro uzel*.

Pokud se odesílajícímu uzlu nepodaří poslat zprávu adresátovi přímo, může ji uložit jako soubor do sítě a do distribuované databáze uložit výsledný identifikátor např. pod klíčem `message:for:`<*id adresáta*>, který adresát při připojení do sítě zkontroluje a doručené zprávy přijme. Před uložením zprávy jako soubor je vhodné ji podepsat, aby nemohlo dojít k podvržení odesílatele, případně i zašifrovat, aby k ní měl přístup pouze adresát. Uzel, pro který je soubor určen, by měl mít oprávnění soubor vymazat.

Triviálním využitím *souborů pro uzel* může být komunikace (např. posílání textových zpráv) mezi uživateli sítě (správci jednotlivých uzlů). *Soubor pro uzel*

lze také použít při distribuci úloh – pokud se zadavatel úlohy odpojí, řešitelé mohou stále posílat výsledky (jako soubory pro zadavatele) a po opětovném připojení zadavatele může zadavatel výsledky zpracovat.

3.2.5 Decentralizovaná distribuce úloh

Pokud není možné nebo vhodné spravovat *task queue* centrálně (např. aby se zadavatel mohl odpojit od sítě a řešitelé stále měli dostupnou frontu úloh k řešení), je nutné předat odpovědnost za udržování *task queue* uzlům v síti. Protože se jedná o stále se měnící data, která mají být dostupná pod nějakým pevným identifikátorem, je vhodné použít distribuovanou databázi.

Task queue lze implementovat v decentralizované databázi jako seznam hodnot s klíčem např. `tasks_by:`<*id zadavatele*>. Hodnota každého záznamu pod tímto klíčem je identifikátorem adresářového stromu (*Merkle tree*) se zadáním úlohy.

Abychom zajistili atomicitu (přidělení každé úlohy právě jednomu uzlu), musíme (kromě uložení páru (*klíč*, *hodnota*) a získání hodnot pro daný klíč) implementovat ještě další operaci distribuované databáze – *dequeue* – získání hodnoty z fronty a její smazání. Také je vhodné vyřešit navrácení zadání úlohy do fronty v případě, že se řešitel odpojí od sítě ještě před jejím vyřešením.

Po vyřešení úlohy musí být každý řešitel schopen odeslat řešení zadavateli – pokud není zadavatel dostupný, lze řešení úlohy uložit do sítě jako *soubor pro uzel* adresovaný zadavateli.

3.2.6 Údržba sítě

Pro možnost úklidu nepotřebných dat v síti je vhodné umožnit ruční mazání souborů. Právo smazat soubor by měl mít jeho autor, nebo příjemce v případě *souboru pro uzel*, např. po zpracování řešení úlohy by měl zadavatel již nepotřebná data řešení smazat. Aby byla síť schopna dlouhodobého provozu bez správy uživateli-správci uzlů, můžeme implementovat následující automatické úklidové mechanismy.

Síť by neměla sloužit jako archivní úložiště souborů, proto je vhodné provádět automatické mazání souborů, o které *nikdo nemá zájem*. Například lze ukládat počty stažení souborů do lokální nebo distribuované databáze, pak v případě potřeby volného místa pro další soubory (nebo automaticky po určité době) mazat nejméně stahované soubory nebo ty, které již dlouho nebyly staženy.

Aby nedošlo k náhlému znepřístupnění nějakého souboru, je vhodnější (namísto smazání souboru z více uzlů současně) postupně snižovat požadovaný počet replik podle stáří souboru. Tím přestane docházet k automatickému obnovování počtu replik souboru na plný autorem určený počet uzlů mechanismem replikace, nové repliky však budou vznikat, pokud bude o soubor zájem, tj. pokud nějaký uzel soubor stáhne na příkaz uživatele. Po zániku poslední repliky je úsporné vymazat i metadata daného souboru z odpovědného uzlu.

Také záznamy v distribuované databázi je vhodné čistit. Každý uzel může podobně jako v případě souborů sledovat přístupy k záznamům, za které je odpovědný, a záznamy, ke kterým dlouho nikdo nepřistoupil, mazat.

3.2.7 Další možné aplikace

Kromě ukládání a sdílení souborů a distribuce úloh je možné vyvinout další aplikace, které by využívaly funkce sítě.

Díky *fault-tolerant* úložišti souborů a adresářových stromů s dobrou škálovatelností je snadné síť použít jako *content delivery network*. Každý uzel funguje jako cache souborů, které může poskytovat např. přes rozhraní HTTP – pokud by uzel neměl požadovaný soubor, stáhl by ho od ostatních uzlů a lokálně uložil. Toho lze ve spojení s ukládáním adresářových stromů triviálně využit k implementaci distribuovaného statického *World Wide Web*u.

Díky distribuované databázi lze implementovat i jednoduché dynamické webové aplikace. Přímočará implementace možnosti psát a zobrazovat komentáře k jednotlivým webovým stránkám, článkům nebo jiným souborům může komentáře ukládat jako hodnoty např. pod klíčem `comment:`<*identifikátor souboru*>. Pro složitější dynamické webové aplikace, např. diskusní fóra lze implementovat vztahy jednotlivých objektů (příspěvků) v distribuovaném *key-value store*, např. identifikátory souborů s textem reakce na určitý příspěvek (také uložený jako soubor) lze uložit jako hodnoty klíče `reactions:`<*id souboru příspěvku*>.

Distribuovaná databáze dále umožňuje použití sítě jako distribuovaného vyhledávače. Kromě souborů ručně nahrávaných do sítě mohou uzly procházet a indexovat *World Wide Web*, jednotlivé stránky indexovat a ukládat jako soubory do sítě společně s jejich původem (URL) v metadatech. Při indexování je možné využít výpočetní výkon sítě např. pro rozpoznávání objektů na stažených obrázcích pro následné použití sítě pro vyhledávání obrázků podle takto získaných klíčových slov.

Síť také lze využít i k přímé komunikaci mezi uživateli. *Soubor pro uzel* umožňuje posílání souborů nebo textových zpráv – distribuovaná obdoba e-mailu. Také můžeme použít peer-to-peer síť pouze pro zprostředkování spojení mezi uživateli pro komunikaci v reálném čase (chat, audio/video hovory), díky šifrování přenosů a možnosti ověření identity protistrany by navíc poskytovala určitou úroveň bezpečnosti.

Dále síť může sloužit k obecnému *tunelování* spojení přes peer-to-peer síť např. pro obejití cenzury nebo kvůli *hole punching*u pro zpřístupnění služby na počítači bez veřejné adresy. Síť by mohla také zajišťovat distribuci *streamů* – uzel může duplikovat přijatý proud dat a přeposílat ho více směry např. pro distribuci multimediálních přenosů (rádio/TV).

Díky ukládání souborů lze síť použít i k zálohování souborů a celých adresářových stromů bez zbytečného ukládání duplicit, s automatickým obnovováním replik a případným indexováním ukládaných dat.

4. Vyhodnocení

4.1 Měření výkonu

4.1.1 Metodika měření

Klientský program navržené peer-to-peer sítě *minet*, program *min* ukládá záznamy o své aktivitě do logu. Z rozdílu časových značek zaznamenaných událostí odpovídajících začátku a konci měřené operace (např. začátek a konec stahování určitého souboru) byla vypočítána délka operace.

Pro možnost porovnání s již existujícím řešením byly některé operace prováděny více způsoby.

Měření probíhala v 1 Gbit/s síti LAN v laboratoři Rotunda na Malé Straně na počítačích `u-pl[1-25].ms.mff.cuni.cz` s operačním systému *Gentoo/Linux* (*Gentoo Base System release 2.2*) pro architekturu x86_64.

Při potřebě měření chování sítě při vyšším počtu uzlů bylo spuštěno více instancí uzlu na každém počítači. Jejich spouštění bylo zajištěno vlastním skriptem *multiply.sh*, pro spuštění tohoto skriptu na více počítačích byl použit program *mussh*. Pro zpracování logu programu *min* a výpočet času uplynulého mezi začátkem a koncem určité operace byl vytvořen skript *time.sh*.

4.1.2 Vyhledání uzlu

4.1.2.1 Motivace

Oproti centralizovanému řešení, kde klient zná server, ze kterého chce např. stahovat soubor (např. klient-server protokolem HTTP), má peer-to-peer síť prodlevu – čas do nalezení uzlu, odpovědného za požadovaný zdroj. Toto měření zjišťuje výkon sítě jako velikost této prodlevy.

Prodleva by měla být závislá na velikosti sítě, proto má smysl provést více měření s různým počtem uzlů.

4.1.2.2 Popis měření

Při měření byly nejdříve spuštěny všechny uzly – 1/5/10/20 instancí programu min (větev `delayed_fix_fingers`[1]) na každém z 25 počítačů. Dále byl spuštěn *měřicí* uzel, který se každou $1\,s$ dotázal sítě na uzel odpovědný za náhodný identifikátor a zaznamenal čas dotazu a čas přijetí odpovědi (min, větev `auto_request_responsible_node`) – log tohoto uzlu byl použit při výpočtu doby vyhledávání daného uzlu. Kromě těchto uzlů byl pro automatické sestavení topologie použit jeden vyhrazený uzel jako *bootstrap*, po připojení všech uzlů do sítě byl vypnut – měl totiž navázané spojení se všemi ostatními uzly, čímž by mohl výsledky měření ovlivnit – zkracoval by cesty mezi uzly.

Po spuštění všech uzlů bylo na všech uzlech současně (v časovém rozmezí cca $1\,s$) aktivováno navazování *finger* spojení (tj. spojení pro efektivnější směrování požadavků a vyhledávání uzlů). Od tohoto momentu probíhalo měření ($t = 0\,s$).

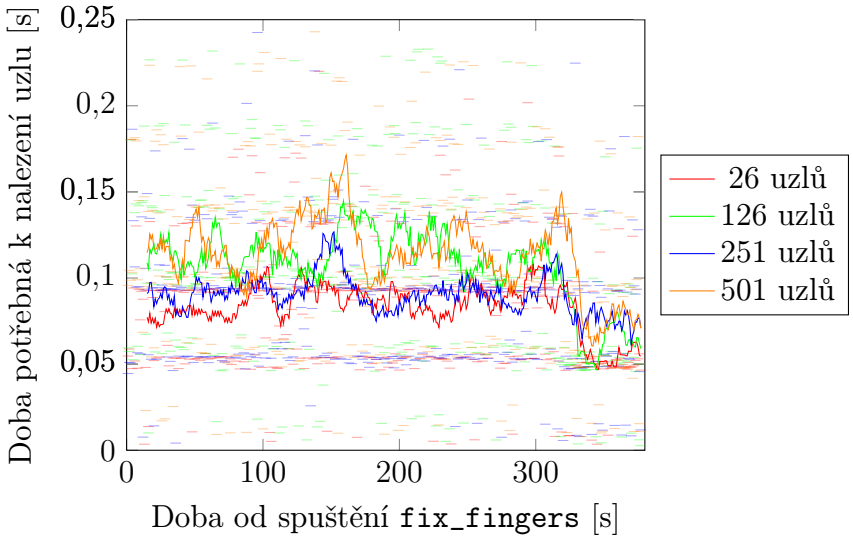
[1] kvůli nestabilitě sítě (přerušování navázaných spojení) při použití transportního protokolu UDT pro spojení mezi uzly byl místo UDT použit protokol TCP

Síť by měla dosáhnout maximálního výkonu za cca 320 s od spuštění (je potřeba nalézt následníky 160 identifikátorů, interval `fix_fingers` je 2 s).

4.1.2.3 Význam měření

Tento postup sice není reálný (současné spuštění všech uzlů typicky nenastává v běžných peer-to-peer sítích), ale poskytuje horní a spodní hranici prodlevy (ihned po připojení do sítě, resp. po navázání všech *fingers*) předpokládané v reálném provozu. Měření také ukazuje význam *finger* spojení na efektivitu/rychlost směrování požadavků v síti a vliv velikosti sítě na dobu nalezení uzlu.

4.1.2.4 Výsledky měření



Obrázek 4.1: Graf – doba potřebná k nalezení uzlu v závislosti na čase uplynulém od začátku navazování *finger* spojení pro různé velikosti sítě.

Světlými barvami jsou zakresleny naměřené hodnoty – prodlevy od poslání požadavku na odpovědný uzel do přijetí odpovědi o jeho nalezení – a sytými čarami je zakreslen klouzavý průměr z posledních 15 hodnot.

4.1.2.5 Interpretace

Rozdělení velikosti prodlevy do nalezení odpovědného uzlu je nerovnoměrné – v grafu jsou zřetelné shluky okolo hodnot 0,05 s, 0,1 s, 0,14 s a 0,18 s. Tyto shluky jsou způsobeny přeposíláním požadavků – při každém přeposlání se zvýší prodleva o cca 43 ms.

Kdykoli je požadavek na vyhledání uzlu přeposlán jinému uzlu, je požadavek na straně odesílatele serializován, vložen do odesílacího bufferu, odeslán operačním systémem do sítě (IP) a směrován jedním nebo více směrovači na cestě (příp. je zpracován lokálně, pokud je požadavek posílán mezi uzly na stejném počítači), data požadavku jsou pak přijata a uložena operačním systémem příjemce a předána aplikaci, která provede deserializaci a zpracování požadavku. Samotný přenos paketů mezi počítači je rychlý – cca 0,2 ms (průměrná doba odpovědi

na *ICMP echo request* posílaném mezi dvěma počítači v laboratoři měřená programem *ping*), prodleva je tedy způsobena transportní nebo vyšší vrstvou. Na aplikační vrstvě by mohla prodleva vzniknout, pokud by při přijetí požadavku aplikace např. zpracovávala předchozí požadavek. V tomto případě by však měl být větší rozptyl délek prodlev, než pozorujeme v naměřených datech. Další možný zdroj prodlevy na aplikační vrstvě by se mohl vyskytnout při odesílání a přijímání dat, protože však aplikace zprávy odesílá ihned po jejich serializaci a o přijatých datech je ihned informována operačním systémem (volání `select`/`epoll`), měla by být takto vzniklá prodleva zanedbatelná ve srovnání s prodlevou způsobenou přenosem dat po síti. Dále by prodleva mohla být způsobena samotnou serializací a deserializací zprávy, ale tato doba by také měla být zanedbatelná. Problém je pravděpodobně způsoben na transportní vrstvě zdržením serializovaných požadavků ještě před odesláním (Nagelův algoritmus a odložené potvrzování přijatých paketů). Další hledání příčiny této prodlevy a optimalizaci implementace ponecháme jako možné rozšíření.

Krátkodobé zvýšení prodlevy („špičky“ viditelné především na klouzavém průměru) jsou pravděpodobně způsobeny zatížením počítačů, na kterých běžely programy uzlů, nebo vytížením sítě v laboratoři jinými uživateli, případně také náhodným výběrem takových identifikátorů (ke kterým mají být nalezeny odpovědné uzly), že odpovědné uzly jsou topologicky vzdálené od měřícího uzlu (tj. s vyšším počtem přeposlání požadavku v peer-to-peer síti).

Měření ukázalo, že průměrná prodleva nalezení uzlu na *nestabilizované* síti se pohybuje mezi 0,08 s a 0,12 s (podle velikosti sítě). Přibližně po 320 s od navázání spojení do peer-to-peer sítě skutečně dochází ke snížení průměrné prodlevy díky posílání požadavků po efektivnějších cestách k odpovědným uzlům – po navázání všech *fingers* spojení je vyhledávání uzlů v síti rychlejší o 22 % až 46 %.

4.1.3 Přenos jednoho souboru

4.1.3.1 Motivace

Po navázání přímého spojení s uzlem je možné s daným uzlem komunikovat. Cílem tohoto měření je zjistit propustnost tohoto spojení.

Vysoká propustnost je důležitá pro přenos velkých souborů, efektivní využití přenosového kanálu je kritické především pro file-sharing sítě.

4.1.3.2 Popis měření

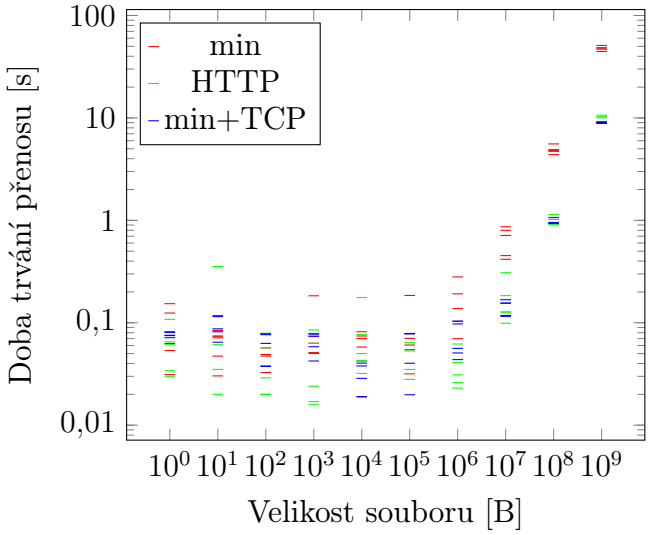
Byla měřena doba přenosu souboru mezi dvěma uzly, instance příjemce a odesílatele byly spuštěny na různých počítačích. K měření byly použity soubory s pseudonáhodným obsahem (generované z /dev/urandom) o velikostech od 1 B do 1 GB (po desetinásobcích: 1 B, 10 B, 100 B, 1 kB, ...), pro každou velikost a typ přenosu bylo provedeno 5 měření.

Měřené způsoby přenosu byly následující:

- min – doba od požádání uzlu, který poskytuje soubor, o daný soubor do přijetí a uložení celého souboru (pro spojení mezi uzly byl použit transportní protokol UDT),
- HTTP – doba běhu klienta stahujícího soubor – obsahuje poslání požadavku, přijetí a uložení odpovědi. Jako HTTP server byl použit program

lighttpd 1.4.35, jako klient *GNU Wget 1.17.1*, doba běhu programu *wget* byla měřena nástrojem *time*,
- min+TCP – viz min, měřeno s použitím transportního protokolu TCP.

4.1.3.3 Výsledky měření



Obrázek 4.2: Graf – doba trvání přenosu souboru v závislosti na jeho velikosti.

V grafu jsou zakresleny doby přenosu souborů různých velikostí třemi způsoby.

4.1.3.4 Interpretace

Pro soubory menší než $1\,MB$ převažuje *šum* – doba navázání spojení (pouze v případě HTTP) a vytvoření, odeslání a přijetí požadavku (tj. režie).

S velikostí rostoucí nad $1\,MB$ je režie zanedbatelná, celková doba stahování roste lineárně s velikostí – poměr velikosti a času odpovídá cca $20\,MB/s$ v případě výsledků programu min, nebo cca $100\,MB/s$ (HTTP). Vyšší rychlost HTTP je způsobena transportním protokolem – HTTP používá protokol TCP, který je vhodnější do LAN, protokol UDT je optimalizovaný pro WAN. Měření proto bylo zopakováno s upraveným programem min používajícím také transportní protokol TCP, jehož výkon je srovnatelný s HTTP.

5. Uživatelská dokumentace

5.1 Popis programu

Program *min* je klient peer-to-peer sítě *minet* umožňující decentralizované uložení souborů a distribuované zpracování úloh.

5.2 Systémové požadavky

Operační systém:

- Windows XP nebo novější, nebo
- Linux 2.6, architektura i386, x86_64 nebo armv6, nebo
- Mac OS X 10.6 nebo novější

Úložiště:

- 50 MB (pro aplikaci a konfigurační soubor),
- další místo pro uložení souborů (velikost závisí na konkrétním použití sítě)

RAM: 256 MB

Síťové připojení: připojení k síti s veřejnou IPv4 adresou (Internet nebo lokální síť). Pokud je adresa sdílená více počítači (NAPT), je nutné nastavit *port forwarding*.

5.3 Popis instalace

Program se distribuuje jako jeden spustitelný soubor obsahující všechny závislosti potřebné pro běh na daném operačním systému.

Instalace programu probíhá zkopírováním souboru do libovolného adresáře.

Na operačních systémech Linux a Mac OS X může být nutné umožnit spouštění programu např. příkazem „`chmod +x min`“.

5.4 Konfigurace

Nastavení programu lze měnit v souboru *config.ini* v pracovním adresáři programu.

5.4.1 Formát

Formát konfiguračního souboru je textový, prázdné řádky a řádky uvozené znakem „`;`“ jsou ignorovány. Každý řádek obsahuje buď

- uvození sekce: „`[`*název sekce*`]`“, nebo
- přiřazení hodnoty danému parametru v aktuální sekci: „*klíč*`=`*hodnota*“.

Nastavit lze následující parametry v těchto sekcích:

- `[db]` – nastavení databáze

  - `file` – cesta k souboru databáze (pro ukládání metadat souborů a informací o uzlech), implicitně „db.db“
- `[overlay]` – nastavení peer-to-peer sítě
  - `local_id` – identifikátor lokálního uzlu, unikátní v síti, implicitně „MISSING_OVERLAY_ID!!“
- `[storage]` – nastavení úložiště souborů
  - `path` – cesta k adresáři, který má být použit jako úložiště souborů, implicitně pracovní adresář
- `[server]` – nastavení komunikace s ostatními uzly
  - `public_addr` – veřejná adresa počítače, na které je dostupný pro ostatní uzly, implicitně „127.0.0.1“
  - `port` – číslo portu, na kterém má program čekat na připojení ostatních uzlů, implicitně „9000“
    Pokud nemá počítač vlastní IP adresu dostupnou z ostatních uzlů (např. pokud je počítač připojen přes směrovač provádějící překlad adres), je nutné na směrovači povolit funkci *port forwarding* podle návodu od výrobce. Číslo portu nastavené v konfiguračním souboru programu se musí shodovat s číslem portu viditelným *zvenku*.
- `[cli]` – nastavení rozhraní pro uživatelské ovládání programu uzlu
  - `addr` – adresa síťového rozhraní, na kterém má být ovládání přístupné, implicitně „127.0.0.1“, tj. dostupné pouze lokálně
  - `port` – číslo portu, na kterém má být ovládání přístupné, implicitně „1111“
- `[json_rpc]` – nastavení rozhraní pro ovládání programu uzlu jinými programy (např. pro přidání úkolu do fronty)
  - `addr` – adresa síťového rozhraní, na kterém má být ovládání přístupné, implicitně „127.0.0.1“, tj. dostupné pouze lokálně
  - `port` – číslo portu, na kterém má být ovládání přístupné, implicitně „2222“
- `[worker]` – nastavení uzlu pro zpracování úloh
  - `workgroup_manager` – adresa uzlu spravujícího frontu úloh, z níž má lokální uzel odebírat zadání úloh, a jemuž má odesílat výsledky
- `[bootstrap_nodes]` – seznam uzlů, se kterými má být navázáno spojení po spuštění programu, ve tvaru „*hostname nebo IP adresa=seznam čísel portů oddělených čárkou*“

5.4.2 Příklad konfiguračního souboru

```
[overlay]
local_id=a81062d09e5c4cc24d9dd92b9c96e8565fd5434b
[server]
public_addr=195.113.21.131
port=12001
[cli]
; povolí ovládání uzlu odkudkoli (!)
addr=0.0.0.0
port=14001
```

```
[json_rpc]
port=16001
[bootstrap_nodes]
; pokusí se o navázání spojení s třemi uzly - na 2 různých počítačích
u-pl1.ms.mff.cuni.cz=9000
195.113.21.155=9000,9001
```

5.5 Použití programu

5.5.1 Spuštění

Program se spouští z příkazové řádky, bez parametrů. Pro spuštění programu v adresáři, kde je program nainstalován, provedeme např. příkazy

„`cd cesta/k/programu`“

„`./min`“ (Linux a Mac OS X) nebo „`min.exe`“ (Windows)

Po spuštění program vypisuje na výstup protokol událostí (např. informace o začátku a konci stahování souboru), detailnější log včetně informací sloužících k ladění se ukládá do pracovního adresáře.

5.5.2 Ovládání klientského programu

Po spuštění je možné program uzlu ovládat pomocí dvou rozhraní: uživatelského a aplikačního. Obě rozhraní podporují stejné příkazy, jejich syntaxe se však liší.

5.5.2.1 Ovládací rozhraní pro uživatele

Program lze ovládat přes příkazovou řádku, k té je možné se připojit protokolem TCP na portu nastaveném v konfiguračním souboru v sekci `[cli]` např. nástrojem *netcat*: „`nc localhost 1111`“.

Po připojení program nabízí interpreter příkazů, které lze zadávat ve formátu „*název příkazu [parametry...]*“ a potvrdit odřádkováním. Seznam příkazů lze získat příkazem „`help`“, stručná nápověda k vybranému příkazu je dostupná příkazem „`help` *název příkazu*“.

5.5.2.2 Ovládací rozhraní pro ostatní aplikace

Program uzlu je možné ovládat také z jiných programů příkazy ve formátu JSON (nyní je implementována pouze minimální podmnožina protokolu JSON-RPC 2.0 nutná pro zpracování validních požadavků). Program poslouchá na TCP portu nastaveném v konfiguračním souboru v sekci `[json_rpc]`.

Po připojení je možné program ovládat příkazy ve tvaru „`{"method": "`*název příkazu*`", "params":` *parametry*`}`“. *Parametry* mohou být buď poziční (zapsané jako JSON array) nebo pojmenované (jako JSON object).

5.5.3 Ukládání souborů

5.5.3.1 Vložení souboru do sítě

Pomocí příkazové řádky je možné do sítě vložit nový soubor příkazem „`put` *cesta k souboru [počet replik]*“ (přes uživatelské rozhraní) nebo „`{"method": "put", "params": ["`*cesta k souboru*`"[, "`*počet replik*`"]]}`“ (přes aplikační rozhraní).

Cesta k souboru určuje soubor uložený na počítači, na kterém běží ovládaný uzel. Cesta může být buď relativní (od pracovního adresáře ovládaného uzlu), nebo absolutní. Při použití uživatelského rozhraní nesmí parametr obsahovat mezery, při použití aplikačního rozhraní na systému Windows je nutné v cestě zdvojit oddělovače, např. „`C:\\Documents and Settings\\Administrator\\My Documents\\test.txt`“.

Počet replik určuje počet kopií daného souboru, které má síť udržovat, tento parametr je nepovinný (implicitně je roven 1).

Po zavolání tohoto příkazu je pro daný soubor vypočítán hash (identifikátor v síti), soubor je zkopírován do úložiště ovládaného uzlu a metadata souboru jsou poslána uzlu odpovědnému za daný identifikátor, aby soubor mohl být nalezen ostatními uzly a aby docházelo k případné replikaci souboru na více uzlů. Poté je vypsán identifikátor uloženého souboru, pomocí kterého lze soubor stáhnout na jiném uzlu.

5.5.3.2 Získání souboru ze sítě

Pro stažení souboru s daným identifikátorem slouží příkaz

„`get` *identifikátor souboru*“, resp.

„`{"method": "get", "params": ["`*identifikátor souboru*`"]}`“.

Tím je stahování daného souboru zařazeno do fronty souborů ke stažení. Po dokončení stahování souboru se do protokolu událostí zapíše řádek

„`file downloaded: "`*cesta ke staženému souboru*`"`“.

Tím je soubor dostupný v úložišti ovládaného uzlu v daném umístění.

5.5.4 Zpracování úloh

Klient *min* umožňuje distribuované zpracování úloh s centrální frontou. Každý uzel může být správcem fronty úloh a/nebo zpracovávat úlohy z fronty na vybraném uzlu.

Úloha je trojice $(program, data, argument)$.

Program je identifikátor spustitelného souboru, *data* je identifikátor souboru se vstupem pro *program*, *argument* je parametr pro *program* – poslaný programu jako prefix vstupu. (Soubory *program* i *data* mohou být společné pro více zadání úloh.)

5.5.4.1 Zadání úkolu

Úkol je možné zadat příkazem `add_task`.

Přes uživatelské rozhraní (jako jeden řádek, odřádkováno pro přehlednost):
„`add_task {"program":"`*id programu*`","argument":"`*parametr*`",`
`"data":"`*id dat pro program*`"}`“

nebo přes aplikační rozhraní:
„`{"method": "add_task",`
`"params": { "program": "`*id programu*`",`
`"argument": "`*parametr*`",`
`"data": "`*id dat pro program*`" }`
`}`“.

Na příkaz přijde odpověď obsahující přiřazený identifikátor úlohy:
„`{"result":{"task_id":"`*přiřazené id úlohy*`",`
`"status":"added to task queue"}}`“.

Po vložení zadání je úloha dostupná ve frontě ovládaného uzlu. Uzly, které mají nastaveny identifikátor ovládaného uzlu jako `workgroup_manager` se dotazují daného uzlu na zadání, jeden z nich vloženou úlohu zpracuje (tj. spustí program s daným vstupem, jeho výstup uloží do sítě jako soubor) a zpět pošle výsledek – trojici (*id zadání úlohy*, *id výstupního souboru*, *návratová hodnota programu*). Výsledek přijatý uzlem *správcem fronty* je uložen do hotových úloh, které je možné z ovládaného uzlu získat.

5.5.4.2 Získání výsledků

Výsledky již zpracovaných úloh lze z uzlu vyzvednout příkazem „`get_results`“, resp. „`{"method": "get_results"}`“ v případě aplikačního rozhraní. Výstupem je seznam řešení úloh jako JSON array:
„`[{"task_id":"`*přiřazené id úlohy*`","output_file":"`*id výstupu*`",`
`"exit_code":"`*návratová hodnota*`"}]`“.

Tímto příkazem dojde po vypsání seznamu výsledků k jejich smazání z ovládaného uzlu.

6. Programátorská dokumentace

6.1 Úvod

Peer-to-peer síť *minet* slouží k decentralizovanému uložení souborů a distribuci úloh. Síť je založena na topologii Chord a používá distribuovanou hashovací tabulku s replikací záznamů pro zajištění spolehlivosti.

Program *min* je klient sítě *minet* – umožňuje přístup k uloženým souborům, nahrávání nových souborů, zadávání a řešení úloh a zároveň zajišťuje chod sítě tím, že se podílí na jejím provozu.

6.2 Struktura programu

O zajištění chodu sítě se starají jednotlivé komponenty:

- **Server** – čeká na příchozí spojení, navazuje odchozí spojení a zajišťuje spolehlivý kanál (*byte stream*) pro přenos dat mezi uzly, (síťová a transportní vrstva RM ISO/OSI)
- **Connection manager** – zajišťuje navazování a udržování spojení s ostatními uzly, (relační vrstva)
- **Overlay** – určuje identitu uzlu, udržuje topologii sítě (*overlay network*) a stará se o abstraktní adresování a směrování podle identity uzlů, (relační vrstva, viz kap. 2.5.3)
- **DHT** – umožňuje ukládání a stahování souborů ze/do sítě včetně automatického nalezení uzlu, ze kterého, resp. na který má být soubor přenesen, (aplikační vrstva, viz kap. 3.1.1)
- **Storage** – skladuje lokálně uložené soubory, umožňuje jejich čtení a přidávání nových souborů, (aplikační vrstva)
- **Replicator** – kontroluje dostupnost souborů, případně vytváří kopie souborů na více uzlech, (aplikační vrstva, viz kap. 3.1.2)
- **Task Queue** – udržuje frontu zadání úloh, které čekají na zpracování a seznam již vyřešených úloh, (aplikační vrstva, viz kap. 3.1.4)
- **Worker** – zpracovává úlohy z fronty zadaných úloh, (aplikační vrstva, viz kap. 3.1.4)
- **CLI**, **JSON-RPC server** – umožňují ovládání klientského programu uživatelem nebo programem.

Jednotlivé komponenty odpovídají třídám, které implementují popsanou funkcionalitu.

Při spuštění programu nechá funkce `main` vytvořit instance potřebných komponent a instanci *čekací smyčky* `boost::asio::io_service`, ve které by měl program strávit zbytek času svého běhu. Jednotlivé komponenty se při vytvoření naváží na `io_service` a posílají do ní události, např. událost připojení nového uzlu od komponenty `server`. Čekací smyčka postupně spouští zpracování těchto událostí.

Zpracování jedné události může do `io_service` přidat nebo odebrat jednu nebo více dalších událostí. Například po odeslání požadavku na jiný uzel se nastaví časovač čekání na odpověď (naplánuje se selhání – timeout), naopak po přijetí

odpovědi na tento požadavek se timeout zruší – naplánovaná událost se odebere z čekací smyčky.

Jednotlivé komponenty na sobě mohou být závislé, např. třída `dht` potřebuje instanci třídy `overlay`, která potřebuje instanci třídy `connection_manager`. O poskytování vytvořených instancí komponent/tříd a případnou automatické inicializaci komponent, na kterých je požadovaná komponenta závislá, se stará třída `di` (*dependency injection*). Třída `di` je vlastníkem instancí již vytvořených komponent. Při žádosti o ještě nevytvořenou komponentu požádá třídu `factory` o její vytvoření, při němž jsou případné závislosti vytvářené třídy získány opět z `di`.

Pro jednoduché použití je třída `di` singleton se statickými metodami, stačí tedy kdekoli v programu požádat `di` o potřebnou komponentu a získanou instanci přímo používat. Například `di::get_dht().put("a.txt", 1)` uloží soubor `a.txt` do sítě, přičemž automaticky zajistí mj. připojení do sítě a inicializaci lokálního úložiště.

Program je vícevláknový, událostmi řízený, s asynchronním zpracováním událostí v hlavním vlákně, ostatní vlákna jsou vytvořena a používána pro každý spuštěný server – kvůli nezávislému čekání na příchozí spojení.

6.3 Popis důležitých částí programu

6.3.1 Connection manager

Hlavní částí programu je třída `connection_manager`, která zajišťuje navazování nových spojení, zpracování přijatých zpráv a předání zprávy jiné části programu. Při vytvoření třídy `connection_manager` je vytvořen server, jehož typ je určený parametrem šablony. Server je obalený třídou `server_service`, která zajišťuje thread-safe běh programu. Při události od serveru (přijetí dat nebo při novém připojení) vloží `server_service` zpracování dané události do fronty čekací smyčky, čímž zajistí její zpracování v hlavním vlákně.

Aplikace předpokládá spojovaný, spolehlivý přenos dat (*byte stream*). Implementovány jsou servery pro transportní protokoly TCP (`tcp_server`) a UDT (`udt_server`), zajišťující spolehlivost nad protokolem UDP. Implementace dalších protokolů (např. μTP) se provede odvozením nového serveru od abstraktní třídy `abstract_server` – definicí callbacků v konstruktoru (pro předání událostí ze serveru aplikaci) a implementací potřebných metod (volaných aplikací, pro předání příkazů z aplikace serveru).

6.3.2 Zprávy

Zprávy jsou potomci třídy `message`, lze je serializovat a deserializovat. Formát serializace je určený knihovnou *boost::serialization*, mezi formáty serializace zpráv lze přepínat změnou definice typů `serialization_oarchive_t` a `serialization_iarchive_t` v souboru `serialization.h`.

Protože přenos zpráv probíhá v *byte stream*u, je nutné posílat délku serializované zprávy samostatně. O to se starají třídy `message_writer` a `message_reader` uvnitř třídy `connection_manager`. Typ zápisu délky zprávy je možné přepínat mezi „čitelným“ (base-10) a „úsporným“ (base-255) změnou definice typu

`uint_serialization_traits` v souboru `serialization.h`.

Aby se předešlo zbytečnému větvení, zpracování zpráv probíhá zavoláním virtuální metody `get_handled` na zprávě, „každý typ zprávy ví, jak má být zpracován“. V parametrech této metody je předán kontext: buď reference na třídu reprezentující připojený uzel, který zprávu odeslal, nebo ID odesílajícího uzlu, pokud byla zpráva přeposílána přes jiné uzly.

6.3.3 RPC (v rámci sítě)

RPC zajišťuje odeslání požadavku a omezení doby čekání na odpověď. O výsledku volání informuje volajícího pomocí callbacků. Pokud přijde včas odpověď, je volajícímu předána prostřednictvím callbacku s parametrem-výsledkem, jinak je zavolán druhý callback, který volajícího informuje o důvodu selhání.

O vytvoření a odeslání požadavku lze požádat `rpc_manager` metodou `call<`*typ RPC požadavku*`>(ok_callback, error_callback, parametry...)`, kde

- *typ RPC požadavku* je potomek třídy `abstract_rpc`,
- `ok_callback` je funkce, která má být provedena při úspěšném přijetí odpovědi a
- `error_callback` je funkce, která má být zavolána v případě chyby.

RPC je použito při dotazování se na informace o uzlu, informace o souboru a při stabilizaci *overlay network*.

6.3.4 Overlay

Aby byla síť decentralizovaná, je nutné rozdělit odpovědnost za úlohy potřebné pro její provoz mezi připojené uzly. O to se stará komponenta `overlay`, která určuje, který uzel má být se kterým spojený (určuje *topologii sítě*), jak má probíhat stabilizace při připojování a odpojování uzlů, jak má probíhat směrování zpráv a definuje který uzel má odpovídat na které požadavky.

Síť používá topologii *Chord*. Každý uzel má v rámci sítě unikátní adresu (*ID* uzlu – m-bitové číslo); adresy jsou uspořádány na kružnici (po `0xFF..FF` následuje `0x00..00`), každý uzel je spojen se svým následníkem.

Každý uzel je zodpovědný za soubory v intervalu (*ID předchůdce*, *vlastní ID*>. Protože je použit spojovaný přenos (původní Chord počítá s přenosem zpráv v samostatných paketech), bylo nutné návrh topologie upravit. Provedené změny, jejich motivace a důsledky jsou popsány ve vygenerované dokumentaci na stránce: „Chord overlay network“.

6.3.5 DHT

Rozdělení odpovědnosti za libovolné *ID* v *overlay network* umožňuje jednoduchou implementaci distribuované hashovací tabulky (*DHT*). Ta umožňuje ukládat textové řetězce (např. soubory) do sítě tak, že je lze opět získat podle jejich identifikátoru, získaného při uložení. Rozhraní *DHT* poskytuje následující funkce:

- `put(string)` → `ID`,
  resp. `put(`*cesta k souboru*`, `*počet replik*`)` → `file` a

- `get(ID)` → `string`,
  resp. `get(ID,` *download finished callback* typu `void(file),` *error callback* typu `void(dht_error))` → `void`.

ID souboru je generováno jako hash obsahu souboru vypočítaný algoritmem `SHA-1`, proto *ID* (s dostatečně vysokou pravděpodobností) jednoznačně určuje konkrétní soubor a soubory jsou mezi uzly rozděleny rovnoměrně.

Typ `file` reprezentuje lokálně uložený soubor s přiřazeným identifikátorem.

O ukládání souborů na disku uzlu se stará třída `folder_storage`, o ukládání metadat o souborech (např. počet replik a seznam známých uzlů, které obsahují daný soubor) třída `db`.

Při vkládání každého souboru do sítě je nutné zvolit počet kopií, které má síť udržovat (viz kap. 3.1.2.2).

6.3.6 Replikace

Aby bylo ukládání souborů v síti spolehlivé, tj. nedošlo ke ztrátě souborů při odpojení uzlu, je nutné ukládat soubory na více místech. To zajišťuje abstraktní třída `replicator`. V případě topologie *Chord* se vyhledává odpovědný uzel za *ID* jako první uzel následující za požadovaným *ID*, proto je vhodné, aby se kopie (*repliky*) ukládaly na následnících odpovědného uzlu. To provádí třída `chord_replicator`.

Kontrolu, zda je dostupný potřebný počet replik, provádí uzel, který je za daný soubor odpovědný. Pokud je v síti replik daného souboru málo, požádá odpovědný uzel své následníky, aby soubor stáhli a poskytovali ho.

6.3.7 Přenosy souborů

Třída `transfer_manager` umožňuje současné stahování a uploadování více souborů (do určitého počtu) a spravuje frontu přenosů čekajících na spuštění. Pro spuštění přenosu stačí předat instanci třídy `upload` nebo `download` třídě `transfer_manager` v metodě `enqueue`. Po dokončení přenosu zavolá instance třídy `upload`/`download` callback poskytnutý při spuštění přenosu.

Pokud uzel chce stáhnout nějaký soubor, nalezne odpovědný uzel a získá metadata o souboru (především seznam uzlů, které poskytují daný soubor) a spojí se s uzlem, od kterého bude soubor stahovat. Vybranému uzlu pošle požadavek na soubor (zprávu `file_request_message`).

Požadavek o soubor je na druhém uzlu zpracován třídou `file_server`, která vytvoří nový `upload` a předá jej instanci třídy `transfer_manager`. Ten požádá server, aby byl callbackem informován, kdykoli se uvolní odesílací buffer spojení s přijímajícím uzlem. Kdykoli je serverem upozorněn, že může odesílat, pošle blok dat ve zprávě `file_part_response_message`, dokud neodešle celý soubor.

Při úspěšném stažení souboru se uzel zaregistruje u odpovědného uzlu do seznamu uzlů, které tento soubor poskytují (tj. pošle danému uzlu zprávu typu `register_replica_message`), čímž umožní ostatním uzlům, aby od něj daný soubor stahovali.

6.3.8 CLI, JSON-RPC server

Třídy `cli` a `json_rpc_server` umožňují ovládat program uzlu. Při inicializaci vytvoří server a čekají na příchozí spojení. Při připojení a přijetí příkazu předají příkaz třídě `interpreter`, která příkaz provede a vrátí odpověď, kterou `cli/json_rpc_server` pošle volajícímu. Podporované příkazy a jejich syntaxe jsou popsány v uživatelské dokumentaci (viz kap. 5.5.2).

6.3.9 Distribuce úloh

Třída `task_queue` udržuje frontu zadaných úloh a řešení těch, které již byly provedeny. Třída `worker` odebírá úlohy z fronty (typicky na jiném uzlu), provádí je a odesílá výsledky zpět do `task_queue`. Třída `task` reprezentuje zadání programu – obsahuje identifikátor programu a identifikátor souboru se vstupními daty.

Zpracování úlohy probíhá následovně:

1. zadavatel úlohy vloží zadání úlohy do `task_queue` na uzlu *správce fronty* např. přes rozhraní JSON-RPC,
2. `worker` (na uzlu, který má zpracovávat úlohy) vyhledá *správce fronty* (tj. uzel s identifikátorem zadaným při vytvoření instance třídy `worker`), požádá ho o zadání úlohy v rámci sítě prostřednictvím RPC volání `task_rpc` (pošle zprávu `rpc_task_request_message`),
3. *správce fronty* odpoví zprávou `rpc_task_response_message` obsahující zadání úkolu,
4. `worker` stáhne soubory obsažené v zadání (tj. program a data), úlohu provede a výsledek (výstup programu) uloží do souboru, který zpřístupní v síti; poté odešle správci fronty řešení zpracované úlohy – `task_result` obsahující identifikátor souboru s výstupem programu,
5. zadavatel vyzvedne řešení z `task_queue` např. přes rozhraní JSON-RPC, případně stáhne výstup a dále ho zpracuje (nebo zadá další úlohu, která s ním bude dále pracovat).

6.3.10 Více informací

Detailní popis dalších částí programu a popis ostatních tříd je obsažen ve vygenerované dokumentaci.

6.4 Překlad programu

Pro usnadnění instalace programu je použito statické linkování – všechny použité knihovny včetně standardních knihoven C++ jsou slinkovány s programem – výsledkem překladu je jeden spustitelný soubor s minimem závislostí. Postup překladu a sestavení programu, seznam jeho závislostí a popis jejich překladu je popsán ve vygenerované dokumentaci na stránce „COMPILE“.

7. Závěr

7.1 Výsledky této práce

V rámci této práce jsme navrhli peer-to-peer síť umožňující ukládání souborů a distribuované zpracování úloh.

Pro ověření návrhu vznikla implementace základní funkcionality navržené sítě, kterou je dále možné rozšiřovat o další funkce.

Výsledky testování a měření výkonu ukázaly, že je aplikace schopna po spuštění na mnoha uzlech vytvořit strukturovanou síť umožňující efektivní vyhledávání zdrojů a že rychlost přenosu dat mezi uzly v síti je srovnatelná s běžně používanými programy.

7.2 Možná vylepšení a další použití sítě

Program je možné dále rozšířit o dosud neimplementované části návrhu: hole punching, ověření identity uzlů a šifrování přenosů, adresářové stromy a distribuovanou databázi.

Také by bylo možné zkrátit dobu přenosu řídicích zpráv navázáním samostatného spojení pro přenos dat (odděleného od přenosu pro řídicí zprávy) a zvýšit propustnost přenosů souborů stahováním jednoho souboru současně z více uzlů.

Dále je možné ještě snížit nároky implementace: topologie může automaticky přerušovat spojení, která *nejsou potřeba* (tj. nejsou *fingers* a už dlouho nic nepřenášela).

Na aplikační vrstvě by také bylo vhodné více respektovat odlišnost prostředků uzlů: např. umožnit uzlu odmítnout uložení souboru při replikaci z důvodu nedostatku místa a vytvořit repliku na jiném uzlu.

Kromě vylepšování již implementované funkcionality je možné klientský program využít jako základ pro další aplikace, např. pro implementaci *content delivery network*, decentralizovaného *World Wide Web*u s dynamickými webovými aplikacemi, distribuovaný vyhledávač, instant messenger nebo VoIP komunikátor.

Reference

[1] Srisuresh, P. and M. Holdrege. *IP Network Address Translator (NAT) Terminology and Considerations*. RFC 2663, DOI 10.17487/RFC2663, August 1999, <http://www.rfc-editor.org/info/rfc2663>.

[2] L. D'Acunto, J.A. Pouwelse, and H.J. Sips. *A Measurement of NAT & Firewall Characteristics in Peer to Peer Systems*. In Proceedings of the ASCI Conference, Zeewolde, the Netherlands, June, 2009.

[3] Delft University of Technology. *Tribler*. <https://www.tribler.org>.

[4] Rosenberg, J., Mahy, R., Matthews, P., and D. Wing. *Session Traversal Utilities for NAT (STUN)*. RFC 5389, DOI 10.17487/RFC5389, October 2008, <http://www.rfc-editor.org/info/rfc5389>.

[5] Mahy, R., Matthews, P., and J. Rosenberg. *Traversal Using Relays around NAT (TURN): Relay Extensions to Session Traversal Utilities for NAT (STUN)*. RFC 5766, DOI 10.17487/RFC5766, April 2010, <http://www.rfc-editor.org/info/rfc5766>.

[6] Rosenberg, J. *Interactive Connectivity Establishment (ICE): A Protocol for Network Address Translator (NAT) Traversal for Offer/Answer Protocols*. RFC 5245, DOI 10.17487/RFC5245, April 2010, <http://www.rfc-editor.org/info/rfc5245>.

[7] Teluu Inc. *PJNATH – Open Source ICE, STUN, and TURN Library*. <http://www.pjsip.org/pjnath/docs/html/index.htm>.

[8] Dafydd Harries, Rémi Denis-Courmont, Kai Vehmanen, Youness Alaoui. *libnice – The GLib ICE implementation*. <http://nice.freedesktop.org/wiki/>.

[9] Google Inc. *libjingle*. <https://developers.google.com/talk/libjingle/developer_guide>

[10] Wikipedia contributors. *Google Talk*. Wikipedia, The Free Encyclopedia, c2016, Datum poslední revize 30. 4. 2016, 19:58 UTC, [citováno 30. 4. 2016]. <https://en.wikipedia.org/w/index.php?title=Google_Talk&oldid=717970424>.

[11] Valve Corporation. *Steam*. <http://store.steampowered.com/>.

[12] Google Inc. *libjingle source tree*. <https://chromium.googlesource.com/external/webrtc/+/master/webrtc/libjingle/>

[13] Holmberg, C., Hakansson, S., and G. Eriksson. *Web Real-Time Communication Use Cases and Requirements*. RFC 7478, DOI 10.17487/RFC7478, March 2015, <http://www.rfc-editor.org/info/rfc7478>.

[14] The Chromium Authors. *Chromium*. <https://www.chromium.org/Home>.

[15] Pidgin developers. *Pidgin, the universal chat client*. <https://pidgin.im>.

[16] Empathy developers. *Empathy*. <https://wiki.gnome.org/Apps/Empathy>.

[17] Network Architectures and Services, Technische Universität München. *NAT-Analyzer*. <http://nattest.net.in.tum.de/results.php>., <http://nattest.net.in.tum.de/ajax.php?id=3>. (3. 5. 2016)

[18] UPnP Standards. *Internet Gateway Device (IGD) Standardized Device Control Protocol V 1.0*, November 2001, <http://www.upnp.org/standardizeddcps/igd.asp>.

[19] Wing, D., Ed., Cheshire, S., Boucadair, M., Penno, R., and P. Selkirk *Port Control Protocol (PCP)*. RFC 6887, DOI 10.17487/RFC6887, April 2013, <http://www.rfc-editor.org/info/rfc6887>.

[20] Eric He, Jason Leigh, Oliver Yu and Thomas A. DeFanti. *Reliable Blast UDP: Predictable High Performance Bulk Data Transfer*. Proceedings of IEEE Cluster Computing, Chicago, Illinois, September, 2002. <https://www.evl.uic.edu/cavern/RBUDP/Reliable%20Blast%20UDP.html>.

[21] The Trustees of Indiana University. *Tsunami UDP Protocol*. <http://tsunami-udp.sourceforge.net>

[22] Eckart, Ben, Xubin He, and Qishi Wu. *Performance adaptive UDP for high-speed bulk data transfer over dedicated links*. In: Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on. IEEE, 2008. p. 1-10.

[23] Tuexen, M. and R. Stewart. *UDP Encapsulation of Stream Control Transmission Protocol (SCTP) Packets for End-Host to End-Host Communication*. RFC 6951, DOI 10.17487/RFC6951, May 2013, <http://www.rfc-editor.org/info/rfc6951>.

[24] Arvid Norberg. *uTorrent Transport Protocol*. <http://www.bittorrent.org/beps/bep_0029.html>.

[25] Yunhong Gu. *UDT: UDP-based Data Transfer Protocol*. Internet Draft, April 2010, <https://tools.ietf.org/html/draft-gg-udt-03>.

[26] Shalunov, S., Hazel, G., Iyengar, J., and M. Kuehlewind. *Low Extra Delay Background Transport (LEDBAT)*. RFC 6817, DOI 10.17487/RFC6817, December 2012, <http://www.rfc-editor.org/info/rfc6817>.

[27] BitTorrent, Inc. *libutp - The uTorrent Transport Protocol library*. <https://github.com/bittorrent/libutp>.

[28] Ernesto Van der Sar. *uTorrent Keeps BitTorrent Lead, BitComet Fades Away*. Torrentfreak.com, 16. 9. 2011, <https://torrentfreak.com/utorrent-keeps-bittorrent-lead-bitcomet-fades-away-110916/>.

[29] National Center for Data Mining, University of Illinois at Chicago. *UDP based Data Transfer Protocol, Breaking the Data Transfer Bottleneck (poster)*. <http://udt.sourceforge.net/doc/udt-sc08-poster.pdf>.

[30] VMware, Inc. *vCloud Connector for vCloud Air help*. <https://pubs.vmware.com/vca/topic/com.vmware.vcc.install.doc/GUID-440EDE10-8177-42C7-87F6-6A09542426A8.html>

[31] Yunhong Gu. *UDT: Breaking the Data Transfer Bottleneck (sourceforge.net*

project). <http://udt.sourceforge.net>.

[32] Yue, Zhaojuan, Yongmao Ren, and Jun Li. *Performance evaluation of UDP-based high-speed transport protocols*. Software Engineering and Service Science (ICSESS), 2011 IEEE 2nd International Conference on. IEEE, 2011. p. 69-73.

[33] Zhu, Ce; et al., eds. *Streaming Media Architectures: Techniques and Applications: Recent Advances*. IGI Global, 2010. strany 264 – 266. ISBN 9781616928339.

[34] Wikipedia contributors, Napster [online], Wikipedia, The Free Encyclopedia, c2016, Datum poslední revize 8. 4. 2016, 18:35 UTC, [citováno 19. 4. 2016] <https://en.wikipedia.org/w/index.php?title=Napster&oldid=714272582>.

[35] Wikipedia contributors, Direct Connect (protocol) [online], Wikipedia, The Free Encyclopedia, c2016, Datum poslední revize 8. 1. 2016, 05:31 UTC, [citováno 19. 4. 2016] <https://en.wikipedia.org/w/index.php?title=Direct_Connect_(protocol)&oldid=698780984>.

[36] Bram Cohen. *The BitTorrent Protocol Specification*. <http://bittorrent.org/beps/bep_0003.html>.

[37] Wikipedia contributors, *EDonkey network* [online], Wikipedia, The Free Encyclopedia, c2016, Datum poslední revize 10. 4. 2016, 06:45 UTC, [citováno 19. 4. 2016] <https://en.wikipedia.org/w/index.php?title=EDonkey_network&oldid=714513786>.

[38] Wikipedia contributors, *Gnutella* [online], Wikipedia, The Free Encyclopedia, c2016, Datum poslední revize 6. 5. 2016, 00:05 UTC, [citováno 17. 7. 2016] <https://en.wikipedia.org/w/index.php?title=Gnutella&oldid=718847132>.

[39] Protocol Labs, Inc. *IPFS is the Distributed Web*. <https://ipfs.io>.

[40] Roger Dingledine, Nick Mathewson. *Tor Protocol Specification*. Torproject.org, [citováno 11. 4. 2016] <https://gitweb.torproject.org/torspec.git/plain/tor-spec.txt>.

[41] The Invisible Internet Project. *I2P Anonymous Network*. <https://geti2p.net/en/>.

[42] University of California. *SETI@home*. <http://setiathome.ssl.berkeley.edu>.

[43] University of Washington. *Rosetta@home*. <http://boinc.bakerlab.org>.

[44] Mersenne Research, Inc. *Great Internet Mersenne Prime Search – GIMPS*. <http://www.mersenne.org>.

[45] Ethereum Foundation. *Ethereum – Blockchain App Platform*. <https://www.ethereum.org>.

[46] Ratnasamy; et al. *A Scalable Content-Addressable Network*. ACM SIGCOMM, 2001.

[47] A. Rowstron and P. Druschel. *Pastry: Scalable, decentralized object lo-*

cation and routing for large-scale peer-to-peer systems. IFIP/ACM International Conference on Distributed Systems Platforms (Middleware), 2001: 329–350.

[48] Ion Stoica, Robert Morris, David Liben-Nowell, David Karger, M. Frans Kaashoek, Frank Dabek, Hari Balakrishnan. *Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications*. IEEE Transactions on Networking 11, 2003.

[49] Petar Maymounkov and David Mazières. *Kademlia: A peer-to-peer information system based on the XOR metric*. In Proceedings of the 1st International Workshop on Peer-to-Peer Systems (IPTPS '02), 2002: 53-65.

[50] Wikipedia contributors, *Kad network*. Wikipedia, The Free Encyclopedia, c2016, Datum poslední revize 11. 10. 2015, 04:38 UTC, [citováno 21. 4. 2016] <https://en.wikipedia.org/w/index.php?title=Kad_network&oldid=685151271>.

[51] eMule developers. *eMule-Project.net – Official eMule Homepage. Downloads, Help, Docu, News...*. <http://www.emule-project.net>.

[52] Andrew Loewenstern, Arvid Norberg. *DHT Protocol*. <http://bittorrent.org/beps/bep_0005.html>.

[53] Bandara, H. M. N. D., Jayasumana, A. P. *Collaborative Applications over Peer-to-Peer Systems – Challenges and Solutions*. Peer-to-Peer Networking and Applications. DOI: 10.1007/s12083-012-0157-3 (2012).

[54] Frans Kaashoek, Frank Dabek, Joshua Cates. *Chord lookup service and the cooperative file system (CFS)*. <https://github.com/sit/dht>.

[55] Dave Winer. *XML-RPC Specification*. <http://xmlrpc.scripting.com/spec.html>.

[56] JSON-RPC Working Group. *JSON-RPC 2.0 Specification*. <http://www.jsonrpc.org/specification>.

[57] Thurlow, R. *RPC: Remote Procedure Call Protocol Specification Version 2*, RFC 5531, DOI 10.17487/RFC5531, May 2009, <http://www.rfc-editor.org/info/rfc5531>.

[58] Nakamoto, Satoshi. *Bitcoin: A Peer-to-Peer Electronic Cash System*. 24 May 2009.

[59] M. Gudgin, M. Hadley, N. Mendelsohn, J. Moreau, H. Frystyk Nielsen, A. Karmarkar, Y. Lafon. *SOAP Version 1.2 Part 1: Messaging Framework (Second Edition)*. W3C Recommendation, 27 April 2007. <http://www.w3.org/TR/2007/REC-soap12-part1-20070427/>.

[60] Eisler, M., Ed. *XDR: External Data Representation Standard*, STD 67, RFC 4506, DOI 10.17487/RFC4506, May 2006, <http://www.rfc-editor.org/info/rfc4506>.

[61] Haynes, T., Ed., and D. Noveck, Ed. *Network File System (NFS) Version 4 Protocol*, RFC 7530, DOI 10.17487/RFC7530, March 2015, <http://www.rfc-editor.org/info/rfc7530>.

[62] Wang, Liang; Kangasharju, Jussi. *Real-world sybil attacks in BitTorrent*

Mainline DHT. In: Global Communications Conference (GLOBECOM), 2012 IEEE. IEEE, 2012. p. 826-832.

[63] Cooper, D., Santesson, S., Farrell, S., Boeyen, S., Housley, R., and W. Polk. *Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile*, RFC 5280, DOI 10.17487/RFC5280, May 2008, <http://www.rfc-editor.org/info/rfc5280>.

[64] The OpenSSL Project. *OpenSSL – Cryptography and SSL/TLS Toolkit*. <https://www.openssl.org>.

[65] Urdaneta, Guido; Pierre, Guillaume; Steen, Maarten Van. *A survey of DHT security techniques*. ACM Computing Surveys (CSUR), 2011, 43.2: 8.

[66] United States National Institute of Standards and Technology (NIST). *Announcing the ADVANCED ENCRYPTION STANDARD (AES)*. Federal Information Processing Standards Publication 197. November 26, 2001.

[67] Dierks, T. and E. Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.2*. RFC 5246, DOI 10.17487/RFC5246, August 2008, <http://www.rfc-editor.org/info/rfc5246>.

[68] GlobalSign. *What Is SSL?*. Globalsign.com [citováno 9. 4. 2016], <https://www.globalsign.com/en/ssl-information-center/what-is-ssl/>.

[69] Rescorla, E. *Diffie-Hellman Key Agreement Method*, RFC 2631, DOI 10.17487/RFC2631, June 1999, <http://www.rfc-editor.org/info/rfc2631>.

[70] *Tor directory protocol, version 3*. Torproject.org, [citováno 12. 4. 2016] <https://gitweb.torproject.org/torspec.git/plain/dir-spec.txt>.

[71] Tim Bray, François Yergeau, Eve Maler, Jean Paoli, Michael Sperberg-McQueen. *Extensible Markup Language (XML) 1.0 (Fifth Edition)*. W3C Recommendation, 26 November 2008. <http://www.w3.org/TR/2008/REC-xml-20081126/>.

[72] Bray, T., Ed. *The JavaScript Object Notation (JSON) Data Interchange Format*. RFC 7159, DOI 10.17487/RFC7159, March 2014, <http://www.rfc-editor.org/info/rfc7159>.

[73] Rumen Kyusakov and John Schneider and Takuki Kamiya and Daniel Peintner. *Efficient XML Interchange (EXI) Format 1.0 (Second Edition)*. W3C Recommendation, 11 February 2014. <http://www.w3.org/TR/2014/REC-exi-20140211/>.

[74] Embedded Internet Systems Laboratory. *Embeddable EXI Processor in C*. <http://exip.sourceforge.net>.

[75] Bormann, C. and P. Hoffman. *Concise Binary Object Representation (CBOR)* RFC 7049, DOI 10.17487/RFC7049, October 2013, <http://www.rfc-editor.org/info/rfc7049>.

[76] Pavel Kalvoda. *libcbor*. <https://github.com/PJK/libcbor>.

[77] Wolong Naphaso. *cbor-cpp*. <https://github.com/naphaso/cbor-cpp>.

[78] Google Inc. *Protocol Buffers*. <https://developers.google.com/

protocol-buffers/>.

[79] Boost developers. *Serialization*. <http://www.boost.org/doc/libs/release/libs/serialization/>.

[80] pfligersdorffer. *EOS Portable Archive*. <https://epa.codeplex.com>.

[81] Rizzo, Juliano; Duong, Thai: The CRIME attack. Ekoparty, 2012. Dostupné online: <https://docs.google.com/presentation/d/11eBmGiHbYcHR9gL5nDyZChu_-lCa2GizeuOfaLU2HOU/edit>

[82] Wikipedia contributors. *Cryptographic hash function*. Wikipedia, The Free Encyclopedia, c2016, Datum poslední revize 25. 5. 2016, 23:39 UTC, [citováno 26. 5. 2016]. <https://en.wikipedia.org/w/index.php?title=Cryptographic_hash_function&oldid=722095683>.

[83] Nathaniel McHugh. *Create your own MD5 collisions*. <http://natmchugh.blogspot.com/2015/02/create-your-own-md5-collisions.html>.

[84] Přispěvatelé Wikipedie. *Narozeninový problém*. Wikipedie: Otevřená encyklopedie, c2015, Datum poslední revize 18. 12. 2015, 14:39 UTC, [citováno 26. 05. 2016]. <https://cs.wikipedia.org/w/index.php?title=Narozeninov%C3%BD_probl%C3%A9m&oldid=13149395>.

[85] Marc Stevens and Pierre Karpman and Thomas Peyrin. *Freestart collision for full SHA-1*. Cryptology ePrint Archive, Report 2015/967, 2015.

[86] Debian Project. *Checksum files for CD images of Debian 8.4.0*. <http://cdimage.debian.org/debian-cd/8.4.0/amd64/iso-cd/SHA1SUMS>.

[87] Git developers. *Git*. <https://git-scm.com>.

[88] Wikipedia contributors. *Merkle tree*. Wikipedia, The Free Encyclopedia, c2016, Datum poslední revize 4. 5. 2016, 06:24 UTC, [citováno 5. 5. 2016]. <https://en.wikipedia.org/w/index.php?title=Merkle_tree&oldid=718556612>.

[89] Oracle Corporation. *MySQL – The world's most popular open source database*. <https://www.mysql.com>.

[90] The PostgreSQL Global Development Group. *PostgreSQL: The world's most advanced open source database*. <http://www.postgresql.org>.

[91] SQLite developers. *SQLite Home Page*. <http://www.sqlite.org>.

[92] SQLite developers. *Most Widely Deployed and Used Database Engine*. <https://www.sqlite.org/mostdeployed.html>.

[93] The PHP Group. *PHP: Installation - Manual, SQLite3 extension*. <http://php.net/manual/en/sqlite3.installation.php>.

[94] Toshiko Matsumoto, Takashi Onoyama, and Norihisa Komoda. *File Size Distribution Model in Enterprise File Server toward Efficient Operational Management*. Proceedings of the World Congress on Engineering and Computer Science 2012 Vol II WCECS 2012, October 24-26, 2012, San Francisco, USA.

Přílohy

Obsah přiloženého CD

- `bin` – Distribuce přeloženého programu *min* pro různé operační systémy
  - Windows
  - Linux (x86 a ARM)
  - Mac OS X
- `src` – Zdrojové kódy programů
  - `lib`
    - `boost` – Zdrojové kódy balíku knihoven *Boost* používaného programem *min*, nutné přeložit, pokud distribuce použitého operačního systému tyto knihovny neposkytuje
  - `min2_build` – Sada skriptů/dávkových souborů pro překlad a sestavení programu *min* a potřebných knihoven, obsahuje jejich zdrojové soubory
    - `json_spirit` – Knihovna sloužící pro serializaci a deserializaci objektů z/do formátu JSON
    - `min2` – Program *min*
    - `openssl` – Knihovna implementující kryptografické funkce
    - `sqlitecpp` – Knihovna implementující relační databázi
    - `tracking_ptr` – Knihovna informující držitele chytrých ukazatelů o zániku objektu, na který ukazují
    - `udt` – Knihovna použitá pro síťovou komunikaci
  - `utils` – Skripty použité při testování sítě, měření jejího chování a zpracování výsledků měření
    - `min_node_generator` – Program vytvářející konfigurační soubory a pracovní adresáře pro program *min*, určen pro vytvoření více uzlů na jednom počítači
    - `multiTCP` – Program umožňující navázat mnoho TCP spojení a poslat libovolná data na všechna současně
    - `min2_log2time` – Program na zpracování logu programu *min* a výpočet časů strávených určitými operacemi
    - `mussh` – Program umožňující provedení daného příkazu na více počítačích
  - `measurements` – Hrubé výsledky měření a skripty pro jejich zpracování
- `doc`
  - `prace.pdf` – Text této práce
  - `html` – Vygenerovaná dokumentace programu *min*
    - `index.html` – Úvodní stránka
    - `md_COMPILE.html` – Popis překladu programu