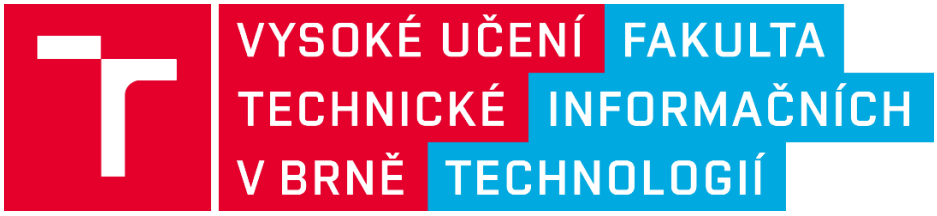
Vysoké učení technické v Brně
Fakulta informačních technologií



Přenos dat, počítačové sítě a protokoly

2018/2019

Hybridní chatovací P2P síť

Tomáš Aubrecht (xaubre02)

24.dubna 2019

Obsah

1 Úvod

Cílem tohoto projektu bylo naprogramovat hybridní chatovací peer-to-peer síť, která je tvořena z registračních uzlů a chatovacích peerů. Uzel i peer fungují jako nezávislí daemoni, kteří běží neustále a kteří po obdržení RPC pokynu vykonají nějakou akci. Z tohoto důvodu bylo potřeba dodat i separátní RPC program, díky kterému by bylo možné uzly i peery ovládat a následně projekt uniformě testovat.

Jazyk zvolený pro implementaci je Python a cílová platforma je referenční virtuální stroj pro tento předmět, na kterém běží operační systém Ubuntu.

Po naimplementování všech potřebných aplikací bylo potřeba provést validační a verifikační testování kompatibility s dalšími spolužáky a dosažené výsledky zdokumentovat.

2 Zadání

2.1 Komunikační protokol

Peeři i uzly používají jednoduchý komunikační protokol sestávající se z níže uvedených zpráv. Všechny zprávy mezi dvěma peery, dvěma uzly, či peerem a uzlem jsou přenášeny skrz UDP. Všechny zprávy mají JSON syntaxi, kde povinným atributem je *type*, který specifikuje typ zprávy. Před přenosem v UDP je obsah zprávy bencodován. Protože UDP negarantuje doručení, tak k potvrzení se používá zpráva *ACK*, která v sobě nese odkaz na jedinečný transakční identifikátor zprávy(*txid*), kterou potvrzuje. Pokud dojde při zpracování libovolné zprávy k chybě, odpovídá protistrana zprávou *ERROR*, která kromě identifikátoru transakce obsahuje i slovní popis problému, ke kterému došlo. Zprávy *HELLO*, *UPDATE* a *ERROR* není potřeba pomocí *ACK* potvrzovat. Na potvrzení *ACK* čekat maximálně 2 vteřiny, poté nejprve resetovat stav zpracování související s nepotvrzenou zprávou a posléze ohlásit chybu na stderr (která by ale neměla v ideálním případě vést k pádu programu, jen notifikovat uživatele o tom, co se děje). Obecně lze zprávy mimo očekávaný stav protokolu zahazovat. Za účelem rozlišení, ke které *GETLIST* zprávě *LIST* patří, může odesilatel navazující *LIST* zopakovat *TXID* z předchozí *GETLIST* zprávy od příjemce.

Protokol podporuje následující zprávy:

- HELLO := {"type":"hello", "txid":<ushort>, "username":"<string>", "ipv4":"<dotted_decimal_IP>", "port": <ushort>}
- GETLIST := {"type":"getlist", "txid":<ushort>}
- LIST := {"type":"list", "txid":<ushort>, "peers": {<PEER_RECORD*>}}
- PEER_RECORD := {"<ushort>":{"username":"<string>", "ipv4":"<dotted_decimal_IP>", "port": <ushort>}}
- MESSAGE := {"type":"message", "txid":<ushort>, "from":"<string>", "to":"<string>", "message":"<string>"}
- UPDATE := {"type":"update", "txid":<ushort>, "db": {<DB_RECORD*>}}
- DB_RECORD := {"<dotted_decimal_IP>,<ushort_port>":{<PEER_RECORD*>}}

- DISCONNECT := {"type":"disconnect", "txid":<ushort>}
- ACK := {"type":"ack", "txid":<ushort>}
- ERROR := {"type":"error", "txid":<ushort>, "verbose": "<string>"}

2.2 Bencode

Bencode je kódování používané pro sdílení souborů, pro ukládání a přenos volně strukturovaných dat. Bencoding se nejčastěji používá u torrent souborů. Bencoding je součástí samotné specifikace BitTorrent. Tyto soubory metadat jsou jednoduše kódované slovníky.

Podporuje čtyři různé typy hodnot a to:

- **bajtové řetězce**: Bajtový řetězec (posloupnost bajtů) je kódována jako <délka>:<obsah>. Délka řetězce je kódována v desítkové soustavě. Například řetězec "projekt" by byl kódován jako *7:projekt*.
- **celá čísla**: Celé číslo je kódováno jako i<číslo v desítkové soustavě>e. Nuly na začátku čísla nejsou povoleny a záporná nula není též povolena. Číslo 42 by tedy bylo kódováno jako *i42e* a číslo 0 jako *i0e*.
- **seznamy**: Seznam hodnot je kódován jako l<obsah>e, kde obsah se skládá z kódovaných prvků seznamu v daném pořadí a tyto prvky jsou zřetězeny. Seznam obsahující řetězec "projekt" a číslo 42 bude kódován jako *l7:projekti42ee*.
- **slovníky** neboli **asociativní pole**: Slovník je kódován jako d<obsah>e. Každý prvky slovníku je zakódován jako klíč bezprostředně následovaný jeho hodnotou. Všechny klíče musí být bajtové řetězce a musí se zobrazovat v lexikografickém pořadí. Slovník, který spojuje hodnoty 42 a "projekt" s klíči "key1" a "key2" (jinými slovy {"key1": "projekt", "key2": 42}), bude zakódován následujícím způsobem: *d4:key17:projekt4:key2i42ee*.

Nejsou zde žádná omezení ohledně toho, jaké hodnoty mohou být uloženy v těchto seznamech a slovnících. Obvykle obsahují jiné seznamy a slovníky, a to umožňuje kódování libovolně složitých datových struktur [*2*].

2.3 Node

Registrační uzel si udržuje aktuální databázi peerů, které se k němu zaregistrovali. S příchodem každé HELLO zprávy aktualizuje údaje v této databázi pro daného peera. V případě přijetí HELLO zprávy s nulovými údaji (IP adresa, port) odebírá peera (resp. uživatele specifikovaného v parametru username) z databáze, stejně tak ve chvíli, kdy od peera neuslyší žádnou HELLO zprávu po dobu delší než 30 vteřin.

Registrační uzly si udržují přehled o peerech existujících v síti v rámci databáze. Na dotazy GETLIST jen od svých peerů odpovídá uzel zprávami LIST, ve kterých jsou všechna dostupná mapování uživatelských jmen peerů a jejich IP adresami a porty (tzn. všech peerů v síti). Interně je registrační uzel schopen v této databázi rozlišit mezi svými peery (které se k němu registrují) a cizími (které se registrují k jiným uzlům). Za účelem tohoto rozlišení má každý záznam v databázi kromě údajů o mapování peera i IP adresu uzlu, který tohoto peera registruje.

Registrační uzel je schopen vytvořit si sousedství s jiným uzlem a vyměnit si informace ve svých databázích peerů. V rámci synchronizace databází si uzly ad-hoc (tj. na základě změny mapování peera) či pravidelně (tj. nejpozději 4 vteřiny od poslední synchronizace) zasílají UPDATE zprávy. V UPDATE zprávě zasílá uzel stav své databáze peerů. Ze své podstaty UPDATE zpráva obsahuje dva typy záznamů: autoritativní (záznamy o těch peerech, kteří se k danému uzlu registrují) a neautoritativní (záznamy o peerech zaregistrovaných k jiným uzlům a které

jsou pouze zprostředkované). Při přijetí a zpracování UPDATE zprávy uzel aktualizuje ve své databázi jen autoritativní záznamy od sousedního uzlu. Z neautoritativních záznamů je tento registrační uzel schopen zjistit IP adresu dalšího uzlu a vytvořit si s ním v případě potřeby nové sousedství. Takto mezi registračními uzly vzniká full-mesh síť. V případě odpojení uzlu od sítě odesílá všem svým sousedům zprávu DISCONNECT, kterou dává pokyn k odstranění záznamů z databáze peerů, pro které je autoritativní. K odstranění selektivního záznamu v databázi dojde také v případě, že uzel neuslyší od autoritativního uzlu žádnou novou UPDATE zprávu po dobu delší než 12 vteřin.

2.4 Peer

Po spuštění se peer připojí k právě jednomu registračnímu uzlu pomocí zprávy HELLO. Ve zprávě HELLO posílá svoje uživatelské jméno (které ostatní používají při zasílání chatových zpráv), IP adresu a port, na kterém peer naslouchá k příjmu chatových zpráv od ostatních. Následně peer zprávu HELLO se stejnými parametry zasílá registračnímu uzlu každých 10 sekund pro udržení spojení. Při ukončení aplikace peer odesílá HELLO zprávu s nulovou IP adresou a nulovým číslem portu, čímž uzlu indikuje, že se odregistrovává ze sítě.

Když chce peer odeslat chatovou zprávu jinému peerovi, tak požádá svůj registrační uzel o aktuální údaje jiného peera pomocí zprávy GETLIST. Odpovědí na tuto zprávu je zpráva LIST, která obsahuje mapování mezi uživatelskými jmény a IP adresami a porty jak peerů zaregistrovaných k tomuto uzlu, tak i peerů zaregistrovaných k jiným uzlům, se kterými je peerův registrační uzel propojen. Pokud odesílající peer neobdrží ve zprávě LIST údaje o příjemci (jeho IP adresa a port), pak není schopen chatovou zprávu odeslat a odmítne takový pokyn. Pro přenos chatu je vytvořeno ad-hoc spojení mezi peery (od odesílajícího na IP a port příjemce), kde se chatová data přenáší ve zprávě MESSAGE. Chatovací zprávy se vypisují v rámci činnosti na stdout, diagnostické informace pak na stderr.

2.5 RPC

Způsob implementace RPC aplikace byl ponechán zcela na řešiteli, kde všechny implementované programy mohou vytvářet dočasné soubory za účelem předávání informací RPC aplikaci. Nicméně všechny dočasné soubory musí být po ukončení programu mazaný tak, aby v souborovém systému nezůstával nepořádek. RPC příkazy by měly být atomické (tzn. dělají jednu věc, typicky odesílají jednu zprávu) a jsou zde pro účely opravování, ale i skriptování chování aplikace. Nicméně implementované aplikace by měly fungovat sami bez sebe i bez jakékoli intervence ze strany RPC [*1*].

3 Implementace

3.1 Komunikační protokol

Pro realizaci komunikace byly naimplementovány dvě třídy: *Message*, která zajišťuje kódování a dekódování zpráv zasílaných mezi uzly a peery, a *Protocol*, která zajišťuje kódování a dekódování RPC příkazů mezi RPC aplikací a uzly a peery. Obě tyto třídy pracují na stejném

principu využívajícím JSON syntaxe a bencoding, kde se liší pouze v seznamu podporovaných příkazů/zpráv a jejich povinných parametrů. Při inicializaci přijímají jako vstupní parametr slovník specifikující daný příkaz/zprávu nebo jejich zakódovanou podobu, kterou nejprve dekódují. Tento vstup je následně zkontrolován, zdali odpovídá komunikačnímu protokolu. Po inicializaci je potřeba zkontrolovat, zdali je daný příkaz či zpráva validní.

Pro kódování a dekódování obsahu zpráv a příkazů zde byla implementována třída *Bencodec* zajišťující tento proces dle specifikace popsané v kapitole 2.2.

3.2 UnitRPC

Pro komunikaci mezi RPC aplikací a jednotlivými uzly či peery se používají unixové sockety. Pro tento účel zde bylo naimplementováno rozhraní *UnitRPC*. Toto rozhraní poskytuje metody, pomocí kterých si každý uzel a peer si při inicializaci vytvoří unikátní jméno socketu, které má následující podobu: "*xaubre02_pds18_<jednotka>_<ID>*", kde jednotka je *node* nebo *peer*, které označuje, zdali se jedná o uzel či peera, a *ID* je identifikátor, který si uživatel zvolí při spuštění uzlu či peera. Adresář, kde se socket vytvoří, je */tmp/*. Pokud takový socket již existuje, uzel či node skončí s chybou informující uživatele, že si má zvolit jiný identifikátor. V opačném případě si daná aplikace vytvoří nový socket, na kterém čeká na příchozí RPC příkazy. Uzel a peer si po ukončení automaticky smažou jimi vytvořený socket pro RPC komunikaci.

Spuštění RPC aplikace je následující:

```
./pds18-rpc.py [-h] --id <ID> --command <příkaz> (--node | --peer)
               [--reg-ipv4 <IPv4>] [--reg-port <port>]
               [--from <username1>] [--to <username2>]
               [--message <zpráva>],
```

kde

`--id <ID>` specifikující jedinečný identifikátor instance,
`--command <příkaz>` specifikující RPC příkaz a
`--node` nebo `--peer` specifikující cíl příkazu
jsou argumenty povinné a
`-h, --help` zobrazující nápovědu,
`--reg-ipv4 <IPv4>` specifikující IPv4 adresu registračního uzlu,
`--reg-port <port>` specifikující port registračního uzlu,
`--from <username1>` specifikující odesílatele,
`--to <username2>` specifikující adresáta a
`--message <message>` specifikující danou zprávu pro uživatele
jsou argumenty volitelné.

RPC aplikace implementuje rozhraní *UnitRPC*. Svou činnost zahajuje kontrolou uživatelem zadaných argumentů, zdali specifikoval všechny potřebné argumenty pro daný příkaz. Pokud ne, informuje o tom uživatele, zobrazí mu veškeré podporované příkazy a jejich formát a skončí s chybou. V opačném případě se pokusí připojit na požadovaný cílový socket, jehož jméno bylo vytvořeno ze vstupních argumentů stejným způsobem, jako je uvedeno výše. Pokud se nepodaří připojit na daný socket, opět o tom informuje uživatele a skončí s chybou. V opačném případě zakóduje příkaz stejným způsobem jako zprávy předávané mezi uzly a

peery, tedy pomocí bencoding, a tento příkaz odešle a skončí, přičemž již nečeká na odpověď a ani nic dále nevypisuje. Veškeré výpisy probíhají na straně uzlů a peerů. Pokud uzel příjme zprávu náležící peerovi, tak ji ignoruje, obdobně peer ignoruje zprávy adresované uzlu.

Seznam podporovaných příkazů pro peera:

- `--peer --command message --from <username1> --to <username2> --message <obsah>`, který se pokusí odeslat chat zprávu
- `--peer --command reconnect --reg-ipv4 <IP> --reg-port <port>`, který se odpojí od současného registračního uzlu (nulové HELLO) a připojí se k uzlu specifikovaném v argumentech
- `--peer --command getlist`, který vynutí aktualizaci seznamu v síti známých peerů, tj. odešle zprávu GETLIST a nechá si ji potvrdit
- `--peer --command peers`, který zobrazí aktuální seznam peerů v síti, tj. peer si s node vymění zprávy GETLIST a LIST, přičemž obsah zprávy LIST vypíše

Seznam podporovaných příkazů pro uzel:

- `--node --command connect --reg-ipv4 <IP> --reg-port <port>`, který se pokusí navázat sousedství s novým registračním uzlem
- `--node --command disconnect`, který zruší sousedství se všemi uzly a odpojí uzel od sítě
- `--node --command neighbors`, který zobrazí seznam aktuálních sousedů registračního uzlu
- `--node --command database`, který zobrazí aktuální databázi peerů a jejich mapování
- `--node --command sync`, který vynutí synchronizaci DB s uzly, se kterými uzel aktuálně sousedí

3.3 UnitUDP

Pro zajištění komunikace pomocí UDP transportního protokolu zde byla implementována abstraktní třída *UnitUDP*. Tato třída implementuje rozhraní *UnitRPC* pro podporu RPC komunikace. Samotná třída poskytuje metody pro: inicializaci komunikace, získání unikátního transakčního identifikátoru(*txid*), potvrzování zpráv, odesílání zpráv typu *ERROR*, příjem *RPC* příkazů(běžící v samostatném vlákně), tisk informací a zpracování signálů. Při zachycení signálu *SIGINT* se nastaví příznak běhu aplikace v nekonečných smyčkách na hodnotu *False*, zastaví se veškeré časovače a uvolní se naalokované zdroje. *UnitUDP* dále obsahuje dvě abstraktní metody, které musí být implementovány třidami dědících tuto abstraktní třídu. Tyto metody představují zahájení činnosti dané aplikace a zpracování přijatých RPC příkazů. Další funkcionalita, kterou tato třída poskytuje, je vytváření časovačů a jejich ukládání do slovníku. Tyto časovače fungují tak, že po vypršení stanoveného časového limitu se vytvoří vlákno, které provede funkci, která byla předána časovači při jeho vytvoření. Časovače se využívají například při čekání na potvrzení zprávy.

3.4 Uzel

Uzel dědí výše zmíněnou abstraktní třídu *UnitUDP*. Tato třída zajišťuje vytvoření samostatně běžícího vlákna, ve kterém uzel čeká na příchod příkazů od RPC aplikace. Po přijetí příkazu jej dekóduje. Pokud se jedná o příkaz určený pro peera, tak jej ignoruje a pokud se nepodaří příkaz dekódovat vůbec, tak o tom informuje zprávou na standartní chybový výstup a čeká na další příkaz. V opačném případě zahájí provedení daného příkazu.

- **database** – Vypíše seznam všech známých peerů, jejich adresy a adresy jejich registračních uzlů.
- **connect** – Pokud je zablokováno přijímání zpráv *UPDATE*, tak jej odblokuje a odešle zprávu *UPDATE* s aktuálním stavem databáze zadanému uzlu.
- **disconnect** – Přestane přijímat zprávy *UPDATE* a odešle zprávu *DISCONNECT* všem svým sousedům a čeká na jejich potvrzení. Poté odstraní ze své databáze všechny neautoritativní záznamy peerů. To provede i v případě, kdy nebyly potvrzeny všechny zprávy. Uzel je nyní pro ostatní uzly nedostupný a to až do doby, než obdrží příkaz *connect*, kdy opět začne přijímat zprávy *UPDATE*.
- **neighbors** - Vypíše seznam všech známých uzlů.
- **sync** - Odešle zprávu *UPDATE* s aktuálním stavem databáze všem svým sousedům.

Souběžně s čekáním na RPC příkazy uzel po své inicializaci a kontrole vstupních argumentů zahajuje svou činnost vstupem do nekonečné smyčky, která je přerušena až s příchodem signálu *SIGINT*, který je zachycen a zpracován pomocí metody definované v abstraktní třídě *UnitUDP*. V této smyčce přijímá na svém UDP socketu určeném pro registraci veškeré příchozí zprávy. Tyto zprávy nejprve dekóduje a v případě chyby v průběhu dekódování odešle na adresu, ze které chybná zpráva přišla, zprávu *ERROR* informující odesílatele o chybě a čeká na příchod další zprávy. Pokud je zpráva úspěšně dekódována, zpracuje ji na základě jejího typu. Zpracování jednotlivých zpráv je následující:

- **ACK** - Zastaví časovač pro dané potvrzení. Pokud potvrzení nedorazí a časovač vyprší, aplikace vypíše chybu na stderr a resetuje svůj stav.
- **UPDATE** – Pokud uzel přijímá zprávy *UPDATE*(není ve stavu po příkazu disconnect), tak ji zpracuje. Nejprve si z adresy, ze které zpráva došla, vytvoří záznam o uzlu. Pokud je tento uzel již uložen v databázi, tak si pouze restartuje časovač jeho validity. Pokud není uložen, tak si jej uloží a spustí časovač validity uzlu, kdy po neobdržení další *UPDATE* zprávy od tohoto uzlu do 12 sekund dojde k jeho odstranění z databáze a s ním i všech peerů, kteří jsou k němu registrovaní. Dále pomocí časovače zahájí automatické zasílání *UPDATE* zpráv každé 4 sekundy tomuto uzlu. Dalším krokem je synchronizace vlastní databáze s databází z této zprávy, kde si aktualizuje pouze autoritativní záznamy peerů. To jsou takové záznamy, kde je adresa registračního uzlu peeru shodná s adresou, ze které zpráva přišla. V případě neautoritativních záznamů se dle potřeby naváže spojení s uzly, ke kterým jsou peeři z neautoritativních záznamů registrováni.
- **HELLO** – Vytvoří si z dané zprávy záznam o peeru a zkontroluje si, zdali ho má již uloženého v databázi. Pokud ano, tak si pouze restartuje časovač validity daného peera. Pokud ne, tak si ho uloží, informuje ostatní uzly a spustí časovač validity, kdy po

neobdržení další *HELLO* zprávy od tohoto peera do 30 sekund dojde k jeho odstranění z databáze a informování ostatních uzlů.

- **GETLIST** – Nejprve potvrdí tuto zprávu, odešle zprávu *LIST* s aktuálním seznamem peerů a následně čeká na potvrzení této zprávy. Pokud potvrzení nedojde, resetuje svůj stav a čeká na další zprávy.
- **DISCONNECT** – Odebere daný uzel ze své databáze známých uzlů, čímž dojde zároveň i k odstranění všech peerů z databáze, které jsou k tomuto uzlu registrovány, a zastavení zasílání *UPDATE* zpráv tomuto uzlu.
- **ERROR** - Vypíše obsah na standartní chybový výstup.

Před ukončením aplikace odešle všem svým sousedům zprávu *DISCONNECT*, ale již nečeká na jejich potvrzení.

Funkcionalita daemona registračního uzlu je dostupná v rámci spustitelného souboru *pds18-node.py*, který při spuštění používá následující argumenty:

```
./pds18-node.py [-h] --id <node ID> --reg-ipv4 <IPv4>
                --reg-port <port>,
```

kde

`-h, --help` zobrazující nápovědu

je volitelný argument a

`--id <node ID>` specifikující unikátní identifikátor instance registračního uzlu,

`--reg-ipv4 <IPv4>` specifikující IPv4 adresu tohoto registračního uzlu a

`--reg-port <port>` specifikující port tohoto registračního uzlu

jsou argumenty povinné.

3.5 Peer

Peer taktéž dědí abstraktní třídu *UnitUDP*. Obsluha příkazů je podobná jako v případě uzlu. Jejich zpracování je následující:

- **getlist** - Realizováno pomocí operace *getlist()*, kde peer odešle svému registračnímu uzlu zprávu *GETLIST* a spustí si časovač pro přijetí potvrzení této zprávy, po obdržení potvrzení čeká maximálně 3 vteřiny na zprávu *LIST*, kde pokud zpráva nedorazí, resetuje svůj stav a čeká na další příkaz.
- **message** - Provede operaci *getlist()*, zkontroluje, zdali se dozvěděl o cílovém uživateli a pokud ano, odešle mu danou zprávu a spustí si časovač čekající na potvrzení přijetí zprávy. V opačném případě informuje na standartní chybový výstup o nenalezení daného uživatele v síti. Peer také provede kontrolu shodu jmen odesílatele specifikovaného v příkazu a uživatelského jména specifikovaného při spuštění peera. Pokud se neshodují, vypíše chybu.
- **peers** - Provede operaci *getlist()* a získaný seznam vytiskne.
- **reconnect** - Zastaví časovač odesílající *HELLO* zprávy, odešle svému registračnímu uzlu zprávu *HELLO* s nulovou IP adresou a portem indikující odpojení peera, změní IP adresu a port svého registračního uzlu na adresu specifikovanou v příkazu, odstraní si lokální databázi uživatelů a opět spustí časovač odesílající *HELLO* zprávy, nyní však na nový uzel.

Souběžně s čekáním na RPC příkazy peer po své inicializaci a kontrole vstupních argumentů zahajuje svou činnost připojením se ke specifikovanému registračnímu uzlu a vytvořením si časovače, který pravidelně odesílá zprávu *HELLO* na daný uzel každých 10 sekund. Následně vstupuje do nekonečné smyčky, kde postupuje stejným způsobem jako uzel. Zpracování jednotlivých zpráv je následující:

- **ACK** - Zastaví časovač pro dané potvrzení. Pokud potvrzení nedorazí a časovač vyprší, aplikace vypíše chybu na stderr a resetuje svůj stav.
- **MESSAGE** nebo **ERROR** - Vypíše jejich obsah na standartní či chybový výstup a zprávu *MESSAGE* potvrdí
- **LIST** - Uloží si její obsah do lokální databáze peerů a tuto zprávu potvrdí.

Před ukončením aplikace odešle svému registračnímu uzlu zprávu *HELLO* s nulovou IP adresou a portem indikující odpojení peera.

Funkcionalita peer daemona je dostupná v rámci spustitelného souboru *pds18-peer.py*, který při spuštění používá následující argumenty:

`./pds18-peer.py [-h] --id <peer ID> --username <user> --chat-ipv4 <IP> --chat-port <port> --reg-ipv4 <IP> --reg-port <port>`,

kde

`-h, --help` zobrazující nápovědu

je volitelný argument a

`--id <node ID>` specifikující unikátní identifikátor instance peera,

`--username <user>` specifikující unikátní uživatelské jméno peera v rámci chatu,

`--chat-ipv4 <IP>` specifikující IPv4 adresu, na které peer naslouchá a přijímá zprávy,

`--chat-port <port>` specifikující port, na kterém peer naslouchá a přijímá zprávy,

`--reg-ipv4 <IP>` specifikující IPv4 adresu registračního uzlu a

`--reg-port <port>` specifikující port registračního uzlu

jsou argumenty povinné.

4 Testování

Testování bylo provedeno ve dvou fázích, kde v první fázi, která probíhala na referenčním virtuálním stroji, se testovala vlastní implementace a komunikace mezi jednotlivými aplikacemi. Po odladění chyb se přešlo do druhé fáze, která obnášela testování funkcionality a kompatibility vůči jiným implementacím, které vytvořili moji kolegové v tomto předmětu.

Mezi prvními byl **David Bednařík(*xbedna62*)** a **Stanislav Bartoš(*xbarto87*)**, kteří implementovali své aplikace také v jazyce Python. Postupně se realizoval jeden scénář, ve kterém se otestoval komunikační protokol a kompatibilita samotná. Tento scénář byl testován dvakrát, kde jsme při prvním testování vypisovali veškerou komunikaci na standartní výstup za účelem její kontroly a případně rychlejší detekce chyb a nesrovnalostí. Tato fáze testování probíhala na školním serveru Merlin.

Při tomto scénáři se postupovalo následovně:

1. každý spustil dvě instance uzlů a dvě instance peerů
2. oba peeři se připojili k jednomu z uzlů (ne mezi kolegy)
3. připojení jednoho uzlu k druhému
4. u obou uzlů se zkontroloval obsah jejich databází peerů a uzlů
5. peeři si poslali zprávu
6. jeden z peeru se přepojil k druhému uzlu
7. u obou uzlů se zkontroloval obsah jejich databází peerů (autoritativní záznamy)
8. peeři si poslali zprávu
9. propojení uzlů s kolegy (kompatibilita)
10. kontrola databází u jednotlivých uzlů (full-mesh síť)
11. odeslání zprávy peerovi na jiné implementaci (kompatibilita)
12. přepojení peera na uzel jiné implementace (kompatibilita)
13. odeslání zprávy z tohoto peera
14. odpojení jednoho z peerů
15. kontrola databází u jednotlivých uzlů
16. snaha odeslat zprávu na odpojeného peera (nelze)
17. odpojení jednoho z uzlů
18. přepojení všech peerů připojených k tomuto uzlu a znovu připojení tohoto uzlu do sítě
19. kontrola databází u jednotlivých uzlů
20. násilné ukončení jednoho z uzlů (timeout)
21. násilné ukončení jednoho z peerů (timeout)
22. připojení nové instance peera k uzlu připojeného do sítě
23. výpis známých uživatelů na tomto peeru (peers)
24. spuštění nových instancí uzlů a peerů a vytvoření druhé sítě
25. propojení těch dvou sítí

Posloupnost jednotlivých příkazů RPC aplikace je dostupná v souboru *readme*.

Následně se zasáhlo do zdrojového kódu aplikací, kde se záměrně zavedli chyby, a sledovala se reakce ostatních implementací. Mezi tyto chyby patří špatné kódování zpráv, špatný formát zpráv, chybné údaje ve zprávách, chybějící odpověď na zprávu *GETLIST* a nepotvrzování zpráv. Každý měl jiný přístup ke zpracování daných chyb, ale žádná chyba nezpůsobila pád aplikace či jiné omezení její činnosti.

5 Závěr

Podařilo se splnit požadavky zadání a implementovat tak funkční aplikace pro vytvoření hybridní peer-to-peer sítě. Veškeré testování proběhlo úspěšně a nebyly nalezeny žádné odchylky od specifikovaného komunikačního protokolu. Aplikace je kompatibilní i s jinými implementacemi vytvořenými kolegy z mého ročníku a je odolná vůči odchylkám od komunikačního protokolu.

Reference

[1] wis.fit.vutbr.cz: *Hromadný projekt - Hybridní chatovací P2P síť.* [Online; navštíveno 1.04.2019]. URL https://wis.fit.vutbr.cz/FIT/st/course-sl.php.cs?id=685650&item=72486&cpa=1

[2] Wikipedia.org: *Bencode*. [Online; navštíveno 1.04.2019]. URL https://en.wikipedia.org/wiki/Bencode