Vysoké učení technické v Brně
Fakulta informačních technologií

Přenos dat, počítačové sítě a protokoly

Hybridní chatovací P2P síť

2019 Bc. Andrej Klocok, xkloco00

Obsah

1 Úvod

Cieľom projektu bolo vytvorenie hybridnej chatovej peer-to-peer siete. Táto sieť pozostáva z jednotlivých uzlov (nodes), na ktorých sú pripojený peerovia. Peerovia si môžu medzi sebou posielať správy. Uzly vytvárajú susedstvo s každým známym uzlom. Ďalej v rámci projektu bola vypracovaná aj rpc aplikácia - vzdialené volanie procedúr uzlu/peera, použitá na testovanie a overenie funkčnosti aplikácii. Komunikácia uzol-uzol, peer-peer, uzol-peer je popísaná protokolom, ktorý je súčasťou zadania.

V nasledujúcich kapitolách si priblížime jednotlivé pojmi ako P2P sieť, kapitola 2 a UDP, kapitola 3. Predstavíme protokol, pomocou ktorého peer a uzol komunikujú, kapitola 4. Implementácia je popísaná v kapitole 5, za ktorou nasleduje pasáž o testovaní, kapitola 6. Nakoniec zhrnieme výsledok v kapitole 7.

2 P2P

Peer to peer sieť sa od klasickej siete typu klient-server líši: architektúrou, adresovaním, smerovaním, decentralizovanosťou. Tieto siete sú známe hlavne kvôli zdieľaniu zdrojov medzi pripojenými peermi.

P2P siete môžeme rozdeliť na:

- pravé - odobratie ktoréhokoľvek uzlu nemá vplyv na stratu schopnosti poskytovať služby
- hybridné - pre svoju činnosť využívajú centrálny uzol pre poskytovania časti služieb

V tomto projekte implementujeme práve hybridnú P2P sieť, pričom za centrálny bod považujeme registračný uzol(node), ktorý obsahuje jednotlivé mapovanie peerov a uzlov. Môžeme si ho predstaviť ako "telefónny zoznam". Keď sa uzol odpojí, stratíme mapovanie na peerov, ktorý sú naň pripojení.

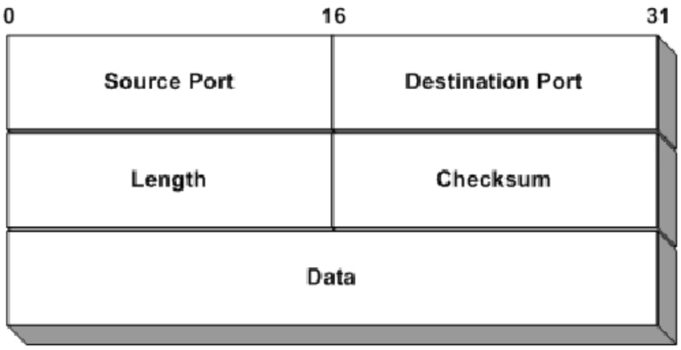
Registračné uzly sa snažia o vytvorenie mesh siete pomocou naviazania komunikácie danej protokolom.

Jednou z kľúčových vlastností siete je samo-organizovateľnosť. Jedná sa o decentralizovanú sieť, ktorú vytvárajú registračné uzly a sú zodpovedné za jej udržiavanie. Každý uzol si spravuje svoje zdroje - v našom prípade databázu pripojených peerov.

Jedným z najznámejších protokolov pre zdielanie obsahu je BitTorrent[1]. Jedná sa o protokol na zdieľanie súborov (P2P), ktorý sa používa na distribúciu dát a elektronických súborov cez internet.

3 UDP

UDP je protokol transportnej vrstvy navrhnutý v roku 1980 a formálne definovaný v RFC 768. Je orientovaný na posielanie správ (datagramov). Poskytuje kontrolu integrity (checksum) a payload. Negarantuje žiadne záruky protokolom vyšších vrstiev pre doručenie správ. Je bezstavový, neudržuje stav odoslanej správy, connection-less. Môže byť označený aj ako nespoľahlivý. Ak chceme zaručiť spoľahlivosť prenosu musí byť implementovaná priamo v aplikácii. UDP hlavičku môžeme vidieť na obrázku 1.



Obr. 1: UDP hlavička

UDP je orientovaný na prenos, čo ho robí vhodným pre jednoduché dotaz-odpoveď protokoly ako DNS, NTP. Poskytuje datagramy vhodné pre modelovanie iných protokolov ako IP tuneling, RPC, NFS. Kedže je bezstavový, jeho využitie je vhodné v topológii s veľkym počtom klientov, napríklad streamovanie videa - IPTV, P2P [1].

[1] `https://www.bittorrent.com/`

4 Protokol

Registračné uzly a peerovia používajú jednoduchý protokol prenášaný cez UDP. Tento protokol má JSON[2] syntax. Každá správa má povinný atribút `type`, ktorý špecifikuje typ správy. Obsah správy je bencodovaný [5.2] pred odoslaním. Protokol podporuje nasledujúce správy:

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>'}

Keďže UDP negarantuje doručenie, použijeme správu typu `ACK`, ktorá v sebe nesie odkaz na jedinečný identifikátor správy, ktorý potvrdzuje.

Ak dôjde pri spracovaní správy k chybe, odpovedá protistrana správou `ERROR`. Táto správa obsahuje aj krátky popis chyby.

Správy `Hello`, `UPDATE`, `ERROR` nie je potrebné potvrdzovať správou `ACK`. Na `ACK` čakať maximálne 2s. V prípade timeoutu zresetovať stav spracovávania súvisiaci s nepotvrdenou správou a ohlásiť chybu na stderr, pričom by nemala viesť k ukončeniu programu. Správy mimo protokol zahadzovať.

Registračný uzol/peer vyvynú maximálne úsilie doručiť správu tým, že sa ju skúšajú poslať až 3x. V prípade, že sa nepodarí správu poslať vypíšu na stderr chybu.

5 Implementácia

V tejto kapitole vypichneme zaujímavejšie pasáže implementácie. Projekt bol implementovaný v jazyku `C++`. Boli implementované 3 CLI aplikácie: registračný uzol(Node), peer a rpc.

5.1 Json

V projekte využívame textový protokol s JSON syntaxou. V `C++` nemáme štandardnú knižnicu na parsovanie JSON. Po prezretí existujúcich riešení som zvolil implementáciu od pána Niels Lohmann [2], JSON for Modern C++. Túto implementáciu som si zvolil hlavne kvôli ich designovým cieľom a faktu, že táto knižnica nie je mŕtva. Je dostupná s `MIT` licenciou, ktorá je priložená spolu s implementáciou v adresáry `libs`.

5.2 Bencoder

Zo zadania musí byť každá správa pred odoslaním B-enkódovaná. Toto kódovanie využíva BitTorrent pre ukladanie a prenos voľne štruktúrovaných dát. V našom prípade sa jedná o JSON.

Bencode používa ASCII znaky.

[2] `https://www.json.org/`

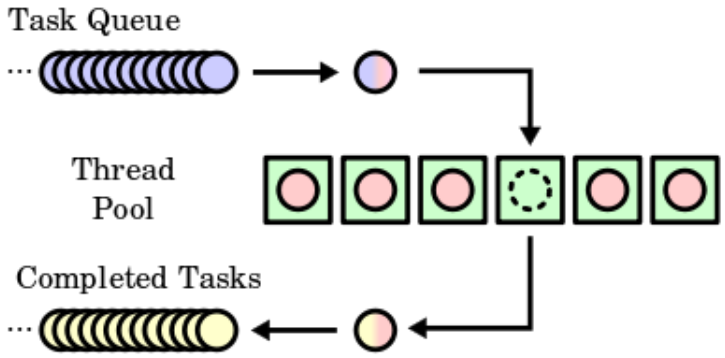
- Integer - zakódujeme ako i<integer so základom 10>e. Napríklad číslo 42 zakódujeme ako i42e, 0 ako i0e, a -42 ako i-42e.
- Byte string - zakódujeme ako <dĺžka>:<obsah>. Dĺžka je kódovaná so základom 10 ako integer, ale nepripúšťa sa záporné číslo (0 je povolená). Napríklad slovo "spam"zakódujeme ako 4:spam.
- List elementov - zakódujeme ako l<obsah>e. Obsah obsahuje benkodované elementy. Napríklad ak by list obsahoval elementy 42, spam tak po zakódovaní dostaneme l4:spami42ee.
- Slovník - zakódujeme ako d<obsah>e. Všetky kľúče musia byť bytové reťazce a musia sa vyskytovať v lexikografickom poradí. Napríklad slovník {"bar": "spam", "foo": 42} zakódujeme ako d3:bar4:spam3:fooi42ee.

Bencoding je jednoduchý pretože čísla sú zakódované ako text v desiatkovom zápise a nie je ovplyvnený endianitou, čo je dôležité pre multiplatformovú aplikáciu, ako je BitTorrent.[3]

Tento algoritmus vykonáva trieda `Bencoder` pomocou metód `encode(json)` a `decode(string)`.

5.3 ThreadPool

Jednotlivé správy sú posielané pomocou transportného protokolu UDP, nemáme vytvorené trvalé spojenie ako v prípade TCP. Keď bindeme soket na danom porte, očakávame správy od ostatných registračných uzlov/peerov. Týchto správ nám môže prísť viac v jeden okamih. Využijeme princíp takzvaného konkurentného servera, kedy budeme mať istý pool vláken, ktorým budeme predávať správy, ktoré nám prídu na soket. Teda budeme mať ThreadPool.



Obr. 2: ThreadPool

ThreadPool je návrhový vzor na dosiahnutie súbežnosti, nazývaný aj replikovaný pracovník. ThreadPool si udržiava viacero vláken, ktoré čakajú na úlohy. Úlohy sa vykonávajú súbežne. Uchovávaním skupiny vláken sa zvyšuje výkon a zabraňuje latencia pri výpočte, kôli častému vytváraniu a ničeniu vláken pre krátkodobé úlohy.

Plánovanie úloh je vykonávané pomocou synchronizovanej fronty úloh. Vlákna vyberajú z fronty čakajúce úlohy. Po ich spracovaní ich umiestnia do fronty dokončených úloh [4].

V rámci projektu bola použitá knižnica od pána Vitaliy Vitsentiy [5]: Modern and efficient C++ Thread Pool Library. Je dostupná s `Apache License 2.0` licenciou, ktorá je priložená spolu s implementáciou v adresáry `libs`.

Projekt využíva konfiguračný súbor `config`, umiestnený v zdrojovom adresáre. Tento súbor obsahuje počet vláken pre Threadpool registračného uzlu a peera. Obsahuje hodnoty <string><medzera><integer>. Jeho obsah vyzerá nasledovne:

ThreadPoolServer 20
ThreadPoolClient 4

Tabuľka 1: Konfiguračný súbor

5.4 Node

Registračný uzol je implementovaný pomocou triedy `Node`, pričom jeho databáza je reprezentovaná triedou `NodeStorage`. Pomocou metódy `registerRpcRequest(std::string keyName, rpcFunction func)` môžeme zaregistrovať novú funkciu, ktorá vykoná volanie rpc. Obdobne metódou `registerBaseRequest(std::string keyName, baseFunction func)` môžeme zaregistrovať nový typ správy protokolu spomínaného v kapitole 4.

Trieda `NodeServer` využíva threadPool 5.3 a je zodpovedná za prečítanie správy z bindnutého soketu, zostavenie štruktúry `NodeWork`. Následne uloží novú úlohu do fronty úloh.

Registračný uzol obsahuje mapovanie peerov na registračné uzly. Na uzol sa registrujú peerovia pomocou správy `HELLO`, ktorú by mal peer posielať každých 10 sekúnd. V prípade, že peer neodošle správu `HELLO` do 30 sekúnd, uzol peera odregistruje a zmaže si jeho záznam z databáze. Od neregistrovaných peerov uzol prijme len správu `HELLO`, ostatné ignoruje.

Databáza registrovaných peerov na konkrétny uzol je implementovaná pomocou slovníka `std::map<std::string, PeerRecord*>`, kde kľuč je meno peera a hodnota je štruktúra uchovávajúca ip adresu, port, timer.

Uzly vytvárajú susedstvo s inými uzlami, pričom si vymieňajú databázu svojich peerov. V rámci synchronizácie si uzly vymieňajú správu `UPDATE` každé 4s. Uzol používa vlákno vykonávajúce funkciu `nodeUpdate()`, v ktorej každú sekundu inkrementuje čítač v mape susedov pre každého známeho suseda. V prípade, že timer dosiahne 12 sekúnd vymaže záznam príslušného uzlu z mapy. Raz za 4 sekundy uzol pošle všetkým susedom, na základe záznamu v mape správu `UPDATE`.

V tejto správe zasiela uzol stav svojej databáze peerov. V `UPDATE` správe sa vyskytujú 2 typy záznamov:

- autoritatívne - záznamy o peeroch pripojených k danému uzlu
- neautoritatívne - záznamy o peeroch pripojených k iným uzlom a sú len sprostredkované.

Pri prijatí update správy uzol aktualizuje vo svojej databázy autoritatívne záznamy od susedného uzlu. Z neautorizovaných záznamov zistí IP adresu ďalšieho uzlu a vytvorí s ním nové susedstvo. Takto vzniká topológia full mesh, zobrazená na obrázku 3.

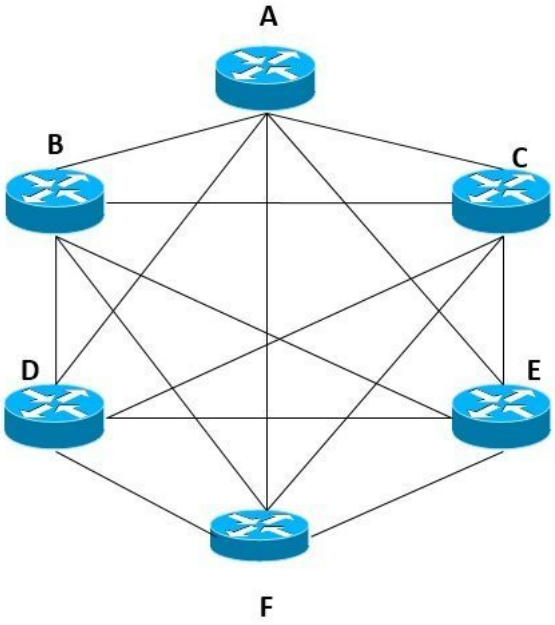
Databáza susedov je implementovaná pomocou pomocou slovníka `std::map<std::pair<std::string, unsigned int>, NodeRecord*> neighbors;`, kde kľúčom je dvojica IP adresa a číslo portu a hodnotou je ukazateľ na štruktúru `NodeRecord`. Táto štruktúra obsahuje timer, vektor záznamov `PeerRecord` a smerovacie informácie.

V prípade odpojenia uzlu zasiela všetkým susedom správu `DISCONNECT` na odstránenie autoritatívnych záznamov z databáze peerov. K odstránení selektívneho záznamu dôjde až pri dovŕšení spomínaného timeout 12s. Uzol reaguje na signál `SIGINT`, kedy odošle správu `DISCONNECT` a nechá si ju potvrdiť od okolitých susedov a až potom sa ukončí.

Funkcionalita démona registračného uzlu bude dostupná v rámci spustiteľného súboru pds18-node, ktorý pri spustení používa následujúce argumenty:

`./pds18-node --id <identifikátor> --reg-ipv4 <IP> --reg-port <port>`

- `--id` je unikátny identifikátor inštancie uzlu pre prípady, kedy je potreba rozlíšiť medzi nimi v rámci jedného hosťa (operačného systému), na ktorom beží
- `--reg-ipv4` a `--reg-port` je IP adresa a port registračného uzlu, na ktorom prijíma registrácie peerov a synchronizácie databáze od susedných uzlov;



Obr. 3: Full-mesh sieť.

5.5 Peer

Peer je implementovaný pomocou triedy `Peer`. Peer využíva lokálnu databázu implementovanú v triede `PeerStorage`.

Trieda `PeerServer` využíva ThreadPool 5.3, je zodpovedná za prečítanie správy z bindnutého soketu, zostavenie štruktúry `PeerWork`. Následne uloží novú úlohu do fronty úloh.

Pripojenie peera na registračný uzol je definované zasielaním správy `Hello`. Táto funkcionalita je implementovaná vláknom, ktoré vykonáva funkciu `peerCommunicator()`. Raz za 10 sekúnd pošle správu `Hello`.

Pre odosielanie správy musí peer získať aktuálne mapovanie, databázu uzla, na ktorý sa registroval. Peer si uloží správu do fronty `std::deque<json> messages`. Je potrebné zaslať správu `GETLIST`. Registračný uzol odošle správu `ACK`, ktorou signalizuje, že správu `GETLIST` prijal a následne odošle správu `LIST`, ktorá obsahuje aktuálne mapovanie, na čo peer odošle správu `ACK`.

Po prijatí správy `LIST` peer zistí, či sa nachádza adresát správy v databázy. Ak nie, ohlási na stderr chybu. Ak hej, správu odošle priamo adresátovi (peerovy) a počká na potvrdzujúcu správu `ACK`. V prípade, že peerovi príde správa, vypíše jej obsah na stdout.

Peer reaguje na signál `SIGINT`, kedy odošle správu `HELLO` s IP adresou "0.0.0.0", portom 0, čo znamená odhlásenie peera a ukončí sa.

Funkcionalita peer démona je dostupná v rámci spustiteľného súboru pds18-peer, ktorý pri spustení používa následujúce argumenty:

`./pds18-peer --id <identifikátor> --username <user> --chat-ipv4 <IP> --chat-port <port> --reg-ipv4 <IP> --reg-port <port>`

- `--id` je unikátny identifikátor inštancie peera pre prípady, kedy je potrebné rozlíšiť medzi nimi v rámci jedného hosťa (operačného systému), na ktorom beží
- `--username` je unikátne užívateľské meno identifikujúce peera v rámci chatu;
- `--chat-ipv4` a `--chat-port` je IP adresa a port, na ktorom peer naslúcha a prijíma správy od ostatných peerov/uzlov;

- `--reg-ipv4` a `--reg-port` je IP adresa a port registračného uzlu, na ktorom peer bude:
  - pravidelne zasielať `HELLO` správy;
  - odosielať dotazy `GETLIST` k zisteniu aktuálneho mapovania.

5.6 RPC

Aplikácia RPC slúži na ovládanie registračného uzla alebo peera. Uzol aj peer si vytvoria dočasný súbor (named pipe) s identickým názvom ako hodnota parametra `id`. Uzol aj peer majú implementované vlákno, ktoré čaká na zapísanie do tohto súbora a následne túto správu prečítajú. Ak odpovedá RPC protokolu tak ju spracujú.

Správy RPC sú podobné protokolu, teda jedná sa o JSON syntax, ktorá kopíruje argumenty programu.

Očakáva sa následujúca spustenia tohoto programu:
`./pds18-rpc --id <identifikátor> <-peer-node"> --command <príkaz> --<parameter1> <hodnota_parametra1> ...`

- –`id` obsahuje identifikátor inštancie peera alebo uzlu, ku ktorému sa má RPC príkaz zaslať
- `--peer`/`--node` určuje inštanciu peera/registračného uzlu
- `--command` a zoznam parametrov určujúci príkaz a parametre vzťahujúce sa k danému RPC volaniu

RPC podporuje nasledujúce príkazy:

- –peer –command message –from <username1> –to <username2> –message <správa>, ktorý sa pokúsi odoslať chat správu
  peer vypíše na stdout nasledujúci text, napríklad:
  `New message from: "xloginYZ"`
  "`Obsah správy`"
  `\n`
- –peer –command getlist, ktorý vynúti aktualizáciu zoznamu v sieti známych peerov, tj. odošle správu GETLIST a nechá si ju potvrdiť
- –peer –command peers, ktorý zobrazí aktuálny zoznam peerov v sieti, tj. peer si s uzlom vymení správy GETLIST a LIST, pričom obsah správy LIST vypíše v zozname <user>,<ipv4>:<port>, napríklad:
  `PEERS`
  `xkloco00,192.168.1.105:8081`
  `xlogin00,192.168.1.105:8089`
  `xloginYZ,192.168.1.106:8080`
  `\n`
- –peer –command reconnect –reg-ipv4 <IP> –reg-port <port>, ktorý sa odpojí od súčasného registračného uzlu (nulové HELLO) a pripojí sa k uzlu špecifikovanému v parametroch
- –node –command connect –reg-ipv4 <IP> –reg-port <port>, ktorý sa pokúsi naviazať susedstvo s novým registračným uzlom
- –node –command disconnect, ktorý zruší susedstvo s všetkými uzlami (stiahne z ich DB svoje autoritatívne záznamy) a odpojí node od siete
- –node –command sync, ktorá vynúti synchronizáciu DB s uzlami, s ktorými uzol aktuálne susedí

- –node –command database, ktorý zobrazí aktuálnu databázu peerov a ich mapovania. Napríklad:
```
------------<Database>------------
xkloco00,192.168.1.105:8081
xlogin00,192.168.1.105:8089
----------<Other nodes>-----------
Node:192.168.1.105:8043
Node:192.168.1.106:8042
\t xloginYZ, 192.168.1.106:8080
---------------------------------
\n
```
- –node –command neighbors, ktorý zobrazí zoznam aktuálnych susedov registračného uzlu. Napríklad:
```
Node neighbors:
IP: 192.168.1.105, port:8043
IP: 192.168.1.106, port:8042
\n
```

6 Testovanie

Projekt bol vyvíjaný na linuxe pod Ubuntu 18.04. Projekt bol testovaný pomocou siete uzlov vlastnej implementácie, na verejných adresách fakultného servera merlin[3] a nakoniec aj v referenčnom virtuálom stroji so systémom Ubuntu 18.04.

6.1 Kompatabilita

Kompatabilita bola otestovaná na fakultnom stroji merlin, pretože poskytuje verejnú IP adresu a je cieľom testovania ďalšich kolegov.

Testovanie prebiehalo najskôr s implementáciou pána Jána Hammera (`xhamme00`) v jazyku Python3. V rámci testovania som odhalil chyby protokola, ktorý bol ešte v rannej verzii a počas semestra sa zmenil. Jednalo sa napríklad o odhlasovanie peera, ale aj niektoré chybové stavy ako nesprávny forma správy `UPDATE`. Na druhej strane, pán Hammer tiež odhalil chybové stavy a indexy v `UPDATE` správe.

Node bol pustený na servere merlin dlhší čas a počas jeho behu sa vytvorila väčšia sieť. Zaujímavým zistením počas testovania bol fakt, že väčšina kolegov písala projekt v jazyku Python3. Kompatabilita bola manuálne otestovaná aj s nasledujúcimi implementáciami pričom komunikácia prebehla bez problémov:

- Roman Dobiáš, xdobia11, python3
- Adrian Kiraly, xkiral01, python3
- Lucia Pelanová, xpelan04, python3
- Tomas Blazek, xblaze31, python3

[3] `https://merlin.fit.vutbr.cz/`

7 Záver

Cieľom projektu bola implementácia hybridnej chatovacej P2P siete s daným protokolom nad transportným protokolom UDP. Správa v tomto protokole má JSON syntax, pričom je benkodovaná pri odoslaní a na druhej strane zas dekódovaná. Boli implementované 3 aplikácie - registračný uzol, chatovací peer, rpc aplikácia na testovanie peera/uzlu. V rámci uzlu a peera sme využili návrhový vzor ThreadPool, pre spracovanie požiadavky prečítanej zo soketu. Registračný uzol aj peer boli otestované na stroji merlin s ostatnými kolegami. U ostatných implementácii dominoval jazyk Python3.

Literatúra

[1] Wikipedia. User Datagram Protocol — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=User%20Datagram%20Protocol&oldid=893767225, 2019. [Online; accessed 27-April-2019].

[2] Niels Lohmann. Json for modern c++. https://github.com/nlohmann/json, 2013-2019.

[3] Wikipedia. Bencode — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Bencode&oldid=867361299, 2019. [Online; accessed 27-April-2019].

[4] Wikipedia. Thread pool — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Thread%20pool&oldid=887954926, 2019. [Online; accessed 27-April-2019].

[5] Vitaliy Vitsentiy. Modern and efficient c++ thread pool library. https://github.com/vit-vit/CTPL, 2014.