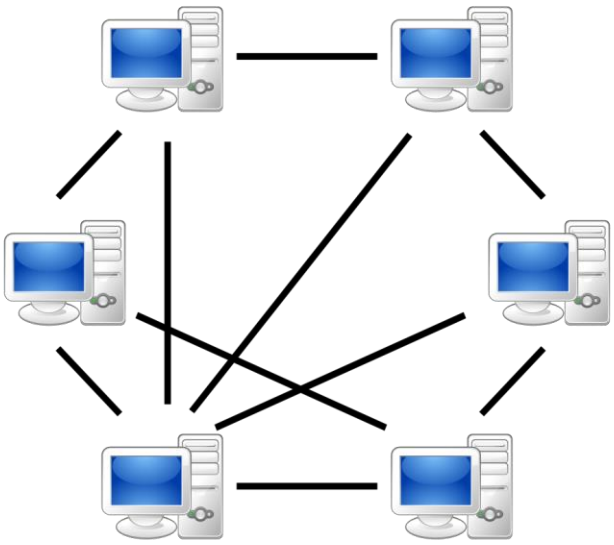
Projektna dokumentacija

Industrijski komunikacioni protokoli

Peer-to-peer (P2P) – Projekat 38



Radili:

Ivan Grubor PR 161/2018

Aleksa Šaršanski PR 156/2017

Sadržaj

Uvod

Ova dokumentacija ima svrhu da detaljno objasni način funkcionisanja aplikacije koja je rađena kao projekat u sklopu predmeta Industrijski Komunikacioni Protokoli. Kompletan projekat je urađen u programskom jeziku C, koristeći WinSock biblioteku.

Osnovno o Peer-to-Peer komunikaciji

Glavna namena ovog projekta je Peer-to-Peer (P2P) komunikacija, koja predstavlja distribuiranu aplikativnu arhitekturu, gde se podaci razmenjuju izmedju svih učesnika u mreži.
Svi učesnici su istovremeno i davaoci i primaoci resursa, što je kontrast tradicionalnom Client-Server modelu, gde su davaoc i primaoci resursa odvojeni.
Svi učesnici u mreži su po pravilu jednaki, ali ova aplikacija je primer centralizovane Peer-to-peer komunikacije, gde centralni server usmerava klijentske zahteve ka drugim klijentima.

Opis problema

Glavna beneficija korišćenja Peer-to-Peer mreže je brzo slanje podataka na velike distance direktno sa uređaja na uređaj. Ova vrsta komunikacije isključuje potrebu za centralnim serverom preko koga bi se vršila komunikacija. Ova aplikativna arhitektura sprečava pojavu da server biva usko grlo, smanjuje troškove za potrebe servera, i povećava skalabilnost celokupnog sistema, tj. otvara mogućnost za veoma lako dodavanje novih klijenata u sistem.

Ova aplikacija služi za deljenje fajlova izmedju više klijenata koji te fajlove žele da downloaduju kod sebe. Kao što je već napomenuto, za razliku od klasične decentralizovane Peer-to-Peer mreže, postoji centralni server koji služi da odgovara na klijentske zahteve, tako što, ukoliko se traženi fajl već nalazi kod nekog drugog klijenta, kao odgovor Server preusmerava klijenta da se „obrati“ tom klijentu i od njega preuzme deo fajla koji je tražen.
Svaki klijent koji učestvuje u ovoj komunikaciji je u obavezi da određene delove fajlova koje ima drži na raspolaganju u memoriji za druge klijente, u slučaju da prime zahtev.

Server inicijalno čuva sve fajlove kod sebe, a klijenti koji se konektuju na njega i imaju mogućnost da preuzmu njegov sadržaj. Server kod sebe čuva informacije o tome da li se neki deo traženog fajla nalazi u memoriji kod nekog drugog klijenta, sa željom da klijenta koji mu traži fajl preusmeri, kako bi mu se obratio (P2P komunikacija) i od njega preuzme taj deo fajla.

Zbog potrebe za visokom skalabilnošću, ova aplikacija podržava teoretski neograničen broj klijenata.

Ciljevi zadatka

Uspostavljanje servera, koji na osnovu informacija koje čuva izdaje komande klijentu kako bi mu na najlakši i najpovoljniji način isporučio celokupan traženi fajl.
Server određuje koji deo fajla određeni klijent treba da sačuva u unapred određenom memorijskom prostoru.
Klijenti treba da čuvaju fiksno 20% fajla kod sebe, tako da i pri isporuci server zna da treba ukupno da bude 5 delova fajla, i treba da se pobrine da svaki deo fajla koji se ne nalazi ni kod jednog klijenta treba biti isporučen.
Ova informacija se u sklopu odgovora šalje klijentu, koji je po protokolu dužan to da ispoštuje.

Odgovornosti servera:

- **Upravljanje listom konektovanih klijenata**
- **Evidentiranje fajlova koji se čuvaju u memoriji klijenata**
- **Preduzimanje adekvatnih akcija u slučaju diconnect-ovanja klijenata**
- **Odgovaranje na klijentski zahtev**
- **Preusmeravanje klijenata na Peer-to-Peer komunikaciju ukoliko se deo traženog fajla već nalazi kod nekog drugog klijenta.**
- **Konstantno osluškivanje novih pristiglih zahteva za konekciju klijenata**
- **Konstantno osluškivanje novih pristiglih zahteva za fajl od strane konektovanih klijenata**

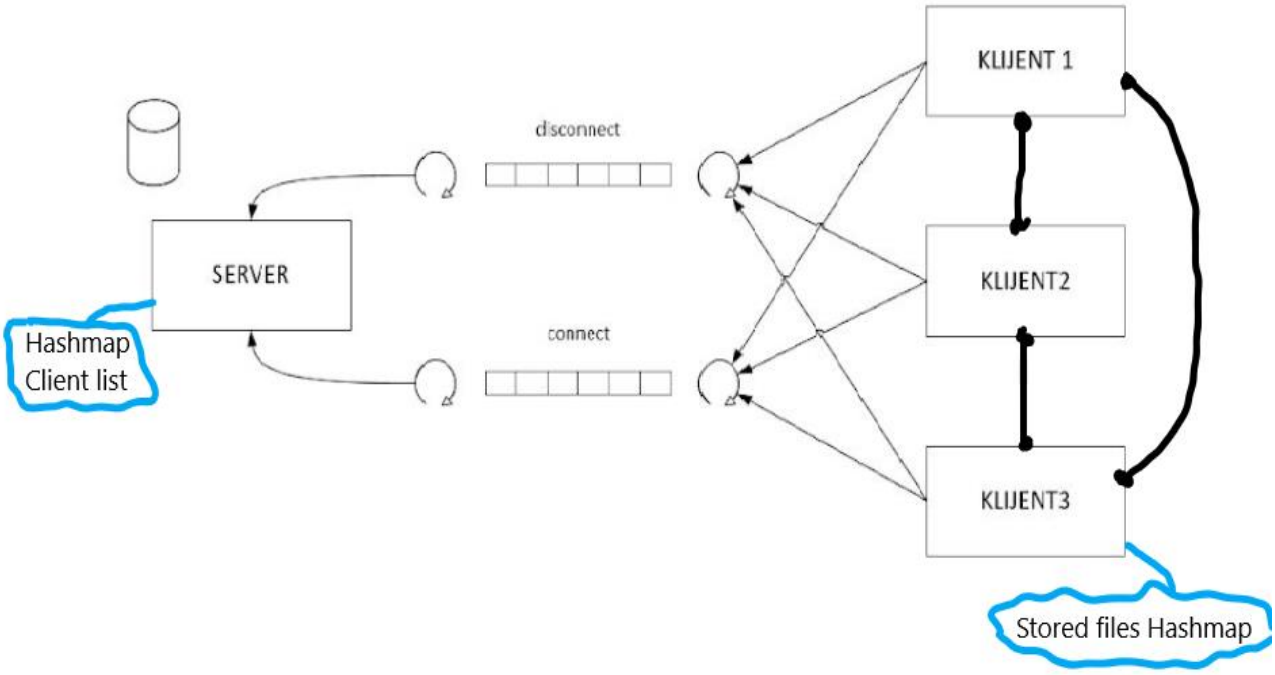
Odgovornosti klijenata:

- **Čuvanje dela fajla koji je server naznačio da treba da sačuva u memoriji**
- **Konstantno osluškivanje zahteva za Peer-to-Peer vezu i odgovor na zahtev za deo fajla**

Svaki od klijentskih zahteva je u obliku strukture **ClientRequest,** koja u zavisnosti od njenog atributa Mode može imati različita značenja. (0 - Traženje fajla, 1 – Connect, 2 - Disconnect).

Dizajn projekta

Funkcionalnost ove aplikacije se bazira na dva tipa komponenti, server i klijenti.
U radu aplikacije, pokreće se samo jedna instanca Servera, i teoretski neograničen broj klijentskih aplikacija.



Komunikacija izmedju prikazanih komponenti realizovana je korišćenjem TCP protokola.
Serverska aplikacija se nalazi na fiksnom portu kojeg klijenti znaju, i na njega se konektuju.
Kompletna komunikacija se dešava na jednom računaru, dakle localhost IP adresa 127.0.0.1.

Server u svom fajl sistemu čuva fajlove. Pored njih, on poseduje još dve strukture podataka (listu trenutno konektovanih klijenata, i hashmapu koja sadrži informacije o tome koji deo fajla se nalazi kod kog klijenta).

Klijent čuva delove fajlova koje mu je server odredio da treba da čuva. Ovi delovi fajlova se čuvaju u Hashmapi.

Svaka od ovih struktura podataka je obezbedjena Kritičnom sekcijom kako ne bi došlo do neželjenih grešaka prilikom paralelnog rada.
(*Svaka od ovih komponenti će biti dodatno objašnjena u sekciji* ***Strukture podataka.***)

Strukture podataka

Serverska Hashmapa

Implementacija HashMape kojom Server rukuje, kao i definicije funkcija za upravljanje njome su definisane u okviru Static Library projekta **HashMap_StaticLib** *(HashTable.h & HashTable.cpp)*

Hashmapa je realizovana kao niz kojem se pristupa preko pozicije. Predstavlja niz pokazivača na strukturu **ClientFile**, koji u sebi ima pokazivač na sledeći element i na taj način čine listu.

```
typedef struct ClientFile
{
    char fileName[FILE_NAME_SIZE]; // Naziv fajla
    int clientPort;  // Port klijenta koji čuva ovaj deo fajla
    int filePart;  // Deo fajla (1 - 5)
    struct ClientFile* next;
} ClientFile;
```

Budući da ime fajla jedinstveno identifikuje svaki fajl, korišteno je kao identifikator fajlova. Hash funkcija koja je korištena za ovu Hash mapu pretvara naziv fajla u broj, koji se koristi kao indeks za direktan pristup traženom elementu.

Pri pokretanju programa, HashTable je prazna i sva polja su inicijalizovana na NULL. Svaki put kad se zatraži fajl, dodaje se u HashMapu, tj. uvezuje se u listu pod odredjenim indeksom novi objekat ClientFile koji ima u sebi informacije da će ubuduće taj klijent čuvati sledeći po redu deo fajla.

U slučaju da klijent prekine konekciju sa Serverom, nije poželjno da se drugi klijenti upućuju da izvrše P2P konekciju na njega. Iz tog razloga je neophodno ukloniti iz HashMape ClientFile koji ukazuje da on čuva delove fajlova.

Client List

Client List je struktura podataka realizovana kao jednostruko spregnuta lista sa pokazivačem na sledeći element. Njena definicija, kao i operacije sa njom se čuvaju u Statičkoj biblioteci **ClientList_StaticLib** *(ClientList.h & ClientList.cpp).*

```
typedef struct Klijent
{
    SOCKET socket;  //Accepted socket klijenta..
    int port;
    struct Klijent* next;
} Klijent;
```

Za svakog konektovanog klijenta se čuva informacija o **Socketu**, kao i **Port** na kome drugi klijenti mogu da mu se obrate preko Peer-to-Peer zahteva.

Klijentska Hashmapa

Svrha Klijentske Hashmape je da čuva delove fajlova u memoriji kako bi, u slučaju Peer-to-Peer zahteva Klijent isporučio drugom klijentu traženi deo. Ova hashmapa se popunjava na zahtev servisa, koji određuje koji deo je potrebno da klijent sačuva.

Implementacija HashMape kojom Klijent rukuje, kao i definicije funkcija za upravljanje njome su definisane u okviru Static Library projekta **ClientOperations_StaticLib** *(ClientFIles.h & ClientFIles.cpp).*

Radi po identičnom principu kao i Hashmapa na Serverskoj strani, istom Hash funkcijom mapira ime fajla koji se čuva u broj, i na taj način se pristupa pozicijama u nizu. U osnovi je niz pokazivača na strukturu **FileKeep.**

```
#define FILE_NAME_SIZE 24
#define FILE_PART_CONTENT 536

typedef struct FileKeep
{
    char fileName[FILE_NAME_SIZE];
    char filePartContent[FILE_PART_CONTENT];
} FileKeep;
```

Razlika u odnosu na Serversku Hashmapu je ta da su elementi niza pokazivači na element, a ne jednostruko spregnuta lista.
Struktura **FileKeep** čuva ime fajla, i deo fajla koji klijent treba da sačuva. Ovo nam omogućava lak pristup delu fajla koji je tražen po indeksu, kojeg dobijemo Hash algoritmom nad imenom fajla.

Tok programa i osnovna logika

I Klijent i Server su programi koji se izvršavaju koristeći više niti.

Serverka glavna nit pokreće nit koja služi da osluškuje pristigle klijentske zahteve, i prihvata ih. Svaki put kad se prihvati nova konekcija, otvara se nova nit koja služi da osluškuje pristigle klijentske zahteve i da odgovara na njih.

Klijentska glavna nit pri pokretanju najpre traži unos Porta, a zatim se pokreće posebna nit koja služi za osluškivanje Peer-to-Peer zahteva od strane drugih klijenata, tj. kreira se Socket na unetom Portu. Na glavnoj klijentskoj niti se dešava glavna logika Klijenta (zatraživanje fajlova).

Statičke biblioteke (Static library)

U okviru ovog projekta, kreirane su 4 statičke biblioteke kako bi se odvojili delovi koda koji služe za manipulaciju strukturama podataka i TCP konekcijom na klijentima i na serveru.

- **ClientList_StaticLib** – Biblioteka za manipulaciju klijentskom listom.
- **HashMap_StaticLib** – Biblioteka za manipulaciju serverskom Hashmapom
- **TCP_Methods_StaticLibrary** – Biblioteka za TCP operacije za klijente i server
- **ClientOperations_StaticLib** – Biblioteka za manipulaciju sačuvanih fajlova kod Klijenta

ClientList Static Library

Statička biblioteka koju koristi Server kako bi vodio evidenciju o klijentima koji su na njega povezani, i čuvao informacije o njima.
U sebi sadrži definiciju strukture **Klijent** *(Strana 6, slika 2),* i operacije za rukovanje njome.

Metode:

void Init_criticalSectionClientList(); - Inicijalizacija kriticne sekcije

void init_client_list(); - Inicijalizacija liste klijenata, postavljanje prvog elementa na NULL

int DodavanjeKlijenataUListu(SOCKET socket, int port); – Kreira novu instancu strukture Klijent, i dodaje je u listu.
Povratna vrednost: **IZMENITI DA NE VRACA PORT**

Klijent* GetClientByPort(int clientPort); – Pronalazi klijenta u listi klijenata na osnovu prosleđenog Porta i vraća pokazivač ka njemu.

void RemoveClientFromList(int clientPort); – Uklanjanje klijenta iz liste po Portu.

int GetLastClientPort(); - Vraća Port poslednjeg klijenta u listi. Ukoliko ne postoji, vraća 0

void EmptyClientList(); - Brisanje svih klijenata iz liste počev od zadnjeg. Poziva se kada želimo da ugasimo server, a nismo disconnectovali sve klijente.

HashMap Static Library

Statička biblioteka koju koristi Server za čuvanje informacija o delovima fajlova koji su sačuvani kod klijenata, i za manipulaciju njome.
U sebi sadrži definiciju strukture **ClientFile** *(Strana 6. Slika 1).*

Metode:

unsigned int hash(char* fileName); - Algoritam Hashiranja imena fajla I dobijanja broj koji se koristi za indeksiranje Hashmape.

void init_criticalsectionHashTable(); - Inicijalizacija kriticne sekcije

void init_hash_table(); - Poziva se na početku programa, I inicijalizuje HashMapu, postavlja sve vrednosti u njoj na *NULL*;

int AddFileToHashTable(char *filename, int clientId); - Ažuriranje HashMape tj. Dodavanje nove informacije o cuvanju dela fajla na određenom klijentu

void ReadClientFiles(char *fileName, int *ports) –

Parametri : 1.) Ime fajla 2.) Niz portova preko reference

Funkcija koja cita sadržaj Hashmape, i učitava portove klijenata u parametar ports. Čitanje se izvršava prolaskom kroz listu u HashMapi i upisom portova iz elemenata liste redom, ukoliko postoje.

void NullateClientFileHashTable(int port) – Funkcija koja se poziva u slučaju kad se klijent otkači sa mreže. Radi se brisanje podataka o klijentu sa prosleđenim portom iz HashMape.

bool ClientHasFile(char* fileName, int port) - Na osnovu imena fajla vrši se provera u HashTabeli da li klijent sa prosleđenim portom već ima downloadovan taj fajl.

void ClearHashTable(); - Služi da izbriše podatke iz HashTabele, redom jedan po jedan ClientFile za svaki fajl koji je kod nekog klijenta bio učitan.

bool RemoveFromEnd(); - Briše (dealocira) memoriju za ClientFile sa kraja strukture.

TCP Methods Static Library

Statička biblioteka za upravljanje komunikacijom izmedju komponenti koristeći TCP protokol. Definiše strukture korištene u komunikaciji: **ClientRequest** i **ClientResponse.**

```
#define FILE_NAME_SIZE 24
#define FILE_PART_SIZE 512

typedef struct ClientRequest
{
    char fileName[FILE_NAME_SIZE];
    int Mode; // Moze biti 0,1,2  ||  0 - Regularno slanje |  1 - Dodavanje |  2 - Exit |
    int port; //Klijent salje svoj Port..
} ClientRequest;

typedef struct FileResponse
{
    char fileName[FILE_NAME_SIZE];
    int keep; //Oznacava deo fajla koji treba sacuvati.
    int ports[5]; // Portovi
    char fileParts[5][FILE_PART_SIZE]; //Delovi fajlova
} FileResponse;
```

ClientRequest je struktura koja služi da bi se formirao zahtev klijenta. Polje fileName predstavlja naziv traženog fajla. Port predstavlja Port na kojem je kreiran Socket za Peer-to-Peer komunikaciju na tom klijentu. Mode predstavlja indikator koja akcija je tražena od Servera. (0 – Traženje fajla, 1 – Connect to Server, 2 – Disconnect from Server)

Metode:

bool connectClientToServer(SOCKET* socket, char* Port) – Koristi se da bi klijent poslao zahtev za konekciju serveru. Ukoliko dođe do greške, return value je false, u suprotnom true.

bool disconnectClientFromServer(ClientRequest* ioc, SOCKET* socket, int port);
- Koristi se da bi klijent poslao zahtev Serveru za disconnect. Kao parametre prima pokazivač na strukturu koju šalje serveru, socket preko kog šalje, i svoj Peer-to-Peer Port. Return value je true ukoliko je slanje uspešno, u ukoliko je došlo do greške je false.

void sendAvailableFilesListToClient(SOCKET* socket, char* availableFiles) – Server poziva ovu metodu kada se klijent konektuje. Tada mu se šalje lista falova koje može da downloaduje.

void recieveAllStoredFiles(SOCKET* socket) – Poziva se nakon konekcije klijenta na server, i služi da preuzme listu dostupnih fajlova od Servera koje klijent može da downloaduje.

void sendClientFileRequestToServer(SOCKET *socket, ClientRequest* ioc) – Poziva se od strane klijenta, i služi sa slanje zahteva Serveru za isporučivanje tj. Slanja **ioc** preko socketa.

int RecieveClientRequest(SOCKET *socket, ClientRequest* req) – Služi za primanje klijentskog zahteva. Kao parametre prima socket na kom primamo poruku, i pokazivač na ClientRequest strukturu u koju store-ujemo pristiglu poruku. Kao povratnu vrednost vraća broj primlj. bajtova.

void SendServerResponseToClient(SOCKET *socket, char* response) – Služi da bi poslao odgovor klijentu na zahtev. Prima dva parametra, prvi je Socket preko kog se odvija komunikacija, a drugi je pokazivač na niz karaktera gde se šalje struktura **FileResponse**.

int RecieveResponseFromServer(SOCKET *socket, FileResponse* response) – Služi za primanje Serverskog odgovora na Klijentski zahtev. Kao parametre prima Socket preko koga se vrši komunikacija, kao i pokazivač na strukturu gde se čuva odgovor. Kao povratnu vrednost šalje broj poslatih bajtova.

Peer-to-peer operacije

bool AskClientForFilePart(SOCKET* socket, char* fileName); - Funkcija koja služi za obraćanje Klijentu od strane drugog klijenta kako bi poslao deo fajla. Ova operacija je Peer-to-peer operacija. Kao parametre prima socket preko kog šalje zahtev, i ime fajla koji traži. Kao return value vraća true ukoliko je uspešno poslao P2P zahtev, u suprotnom vraća false.

bool RecieveFilePartFromClient(SOCKET* socket, char* fullFileContent); - Prihvatanje Peer-to-Peer odgovora, tj. dela fajla. Parametri su Socket preko kog se odvija komunikacija, i buffer u koji se upisuju jedan po jedan deo fajla. Vraća true ukoliko je uspešno, u suprotnom vraća false.

char* RecieveClientP2PRequest(SOCKET* socket) – Služi da Klijent prihvati P2P zahtev od drugog klijenta. Klijent prima sadržaj fajla, i vraća kao povratnu vrednost funkcije.

void AnswerP2P_Request(char* fileContent, SOCKET* socket); - Koristi se da pošalje fileContent *(parametar 1)* preko mreže, koristeći socket *(parametar 2).*

Pojašnjenje dodatnih funkcija na Client.cpp i Server.cpp

Metode objašnjene u ovom segmentu se izvršavaju u okviru Client.cpp i Server.cpp, i ne rukuju strukturama podataka ni TCP konekcijom.

Server.cpp Metode

bool ReadFilePart(char* fileName, int part, char* fileInput); - Citanje dela fajla u strukturu koja se šalje preko mreže.

int ReadFullFile(char* fileName, char* fullFileBuffer) – Učitavanje sadržaja fajla u alocirani buffer.

Thread Funkcije na Server.cpp

DWORD WINAPI AcceptClients(LPVOID lpParam); - Nit koja služi sa osluškivanje pristiglih zahteva za konekciju od strane klijenata. U slučaju zahteva za konekciju, iz ove niti se otvara nova nit za svaki zahtev, **ClientThread** funkcija. Nit AcceptClients je takođe zadužena da gasi klijentske niti u slučaju da se dati Klijent disconnectuje, i zatvara taj Handle. Kao parametre prima strukturu Parameters.

DWORD WINAPI ClientThread(LPVOID lpParam); - Nit kojoj je svrha opsluživanje klijenata i odgovaranje na klijentske zahteve. U ovoj niti je implementirana glavna logika Servera. Kao parametar prima strukturu Parameters. Svaki pristigli klijentski zahtev ***(ClientRequest)*** se ovde analizira, i generiše odgovarajući Serverski odgovor ***(FileResponse)*** na osnovu njega.

Thread Funkcije na Client.cpp

DWORD WINAPI ClientThread(LPVOID lpParam); - Nit koja se pokreće na Client.cpp, i služi za osluškivanje Peer-to-Peer zahteva za konekciju, i u slučaju da neki drugi klijent zatraži deo fajla, sadržaj dela fajla se učitava iz Klijentske HashMape i isporučuje zahtevaocu.

Stress Testovi

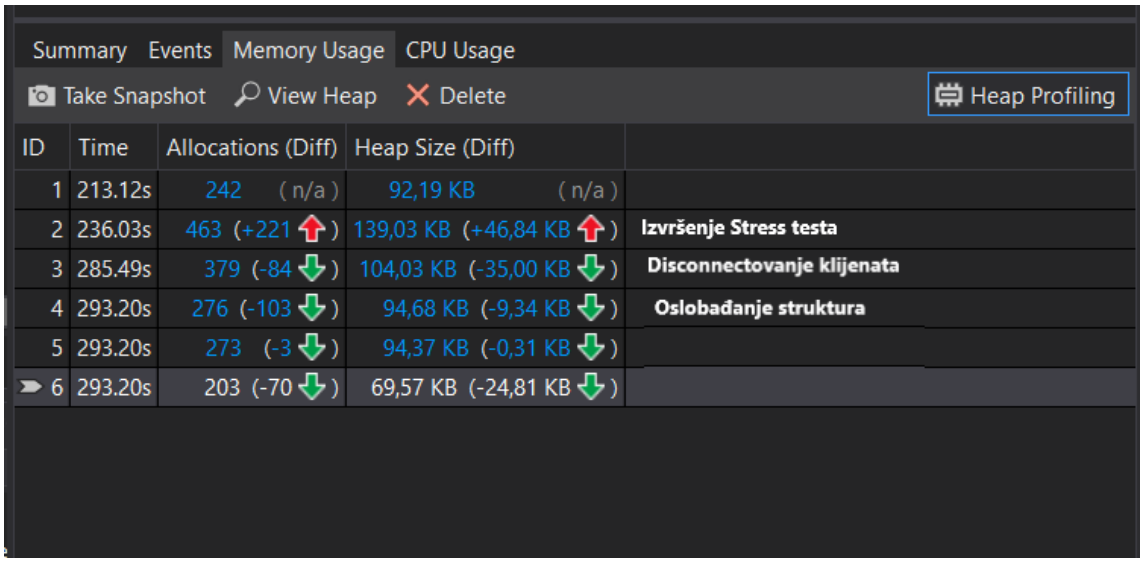
Stress testovi služe za proveru rada i performansi aplikacije kada je pod opterećenjem. Ovakve slučajeve je teško testirati ručno, i potrebno je programski opteretiti sistem do granica pucanja kako bi se neke od grešaka ispoljile. Stress Testovima se proverava stabilnost i otpornost programa na otkaze.

S obzirom da opisani program radi sa fajlovima, i opslužuje teoretski neograničen broj klijenata, u ovom slučaju smo odabrali da testiramo rad Servera, i koliko efikasno opslužuje klijente, i kapacitete fajlova koji se prenose preko mreže.

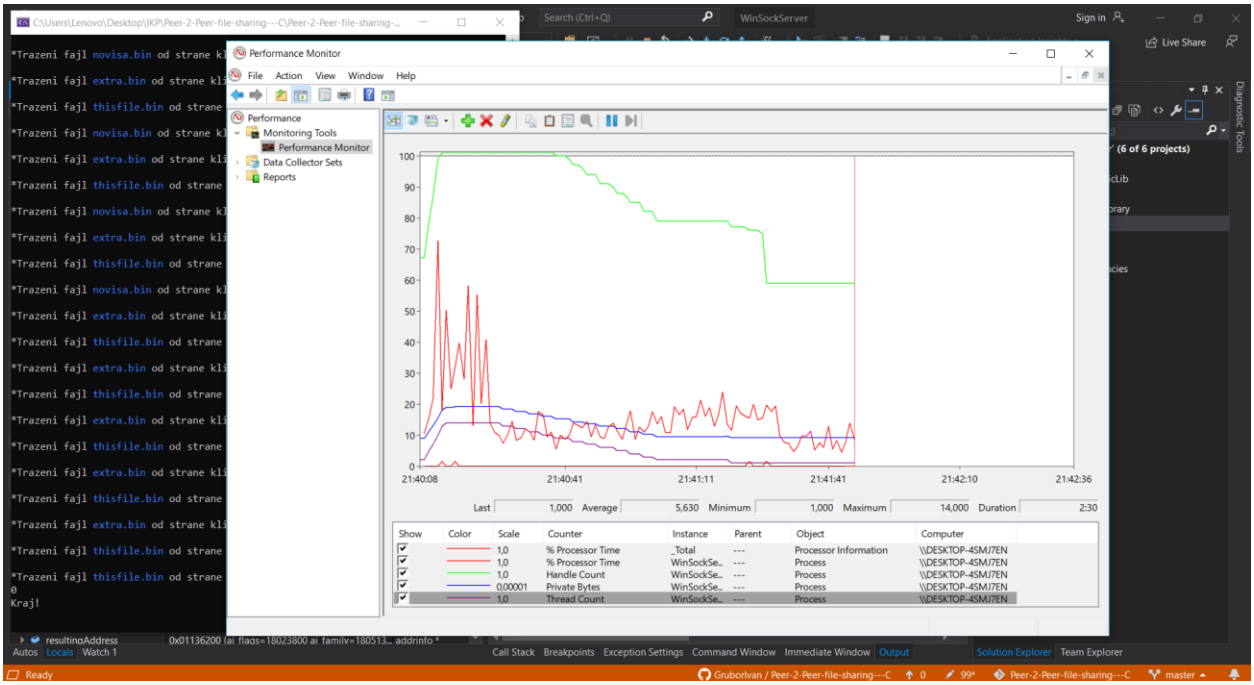
Stress Test #1

Ovaj stress test je baziran na testiranju Servera, i njegove sposobnosti da opsluži sve klijente koji su konektovani, isporučujući im sve fajlove koje traže. Svrha je provera kako server reaguje pod pritiskom kad primi mnogo zahteva u sekundi.
Test je koncipiran tako da svaki klijent nakon što se pokrene neprestano šalje zahteve Serveru za različite fajlove. Radi po principu random, i nasumično bira fajlove koje isporučuje.
Test je realizovan pokretajući 15 Klijentskih instanci, koji istovremeno šalju zahteve.

Ovim Stress testom je takođe testiran konkurentni rad, i rad sa Kritičnim sekcijama u okviru struktura podataka koje rukuju operacijama na Serveru.



Snimak Heap-a pri izvršenju prvog stress testa.



Performance monitor grafik prilikom izvršenja stress testa

Stress Test #2

Ovaj stress test proverava rad Peer-to-Peer komunikacije pri velikom broju zahteva. Testira komunikaciju izmedju klijenata, kad ih Server preusmeri na Peer-to-Peer konekciju.

Zaključak stress testova

Zaključci testova

Server odlično rukuje klijentskim zahtevima. Veoma brzo odgovara na zahteve zbog toga što se komunikacija sa svakim klijentom dešava u zasebnom Threadu. Strukture podataka su obezbeđene kritičnom sekcijom, I ne postoje preplitanja pri pristupu resursima na Serveru. Nisu se pokazali nikakvi nedostaci programu prilikom izvršenja testa.

Peer-to-Peer komunikacija radi bez propusta uprkos pojačanom broju Peer-to-Peer zahteva, I odlično reaguje na opterećenje.

Mana zbog koje je dovođenje programa do granica mogućnosti otežano je što je implementirano da ukoliko server pošalje jednom klijentu određeni fajl, neće mu slati opet isti fajl, jer ga taj Klijent već ima kod sebe. Ova implementacija je smišljena pre nego što smo razmišljali o potencijalnim stress testovima. Iz ovog razloga je test odrađen sa tako velikim brojem klijenata, I svakako je proveren slučaj kad dva klijenta istovremeno pristupaju istom fajlu. Takodje je testirana Peer-to-Peer komunikacija.

Potencijalna unapređenja

Rezultati stress testa su pokazali da Server dobro podnosi opterećenje kada istovremeno mora da odgovori na više zahteva.