UNIVERZITET U NOVOM SADU
FAKULTET TEHNIČKIH NAUKA

PROJEKAT IZ PREDMETA

"Industrijski komunikacioni protokoli u elektroenergetskim sistemima"

Advanced Heap Manager (AHM)

Profesori: Branislav Atlagić
Ranko Popović

Asistent: Ognjen Jelisavčić

Studenti: Đorđe Radivojević PR106/2016
Vukašin Radić PR119/2016
Dejan Jovanić PR128/2016

Novi Sad, 2020. godine

Sadržaj

1. Uvod

Heap manager predstavlja komponentu koja rukuje dinamički alociranom memorijom. Zatraženu memoriju stavlja na raspolaganje korisniku, a prethodno je zauzima na memorijskom segmentu koji se naziva heap.

Predmet ovog projekta jeste Advanced Heap Manager (AHM u daljem tekstu) koji omogućava uporedni rad na konfigurabilnom broju heap-ova i na taj način rešava problem heap contention-a.

Heap contention predstavlja jednovremeno pristupanje heap-u od strane više niti, što dovodi do usporavanja programa. Pored AHM-a projekat sadrži i client-server arhitekturu koja je zadužena za obavljanje funkcionalnosti putem računarske mreže, kao i testove koji demonstriraju samu implementaciju.

Ciljevi ovog projekta su sledeći:

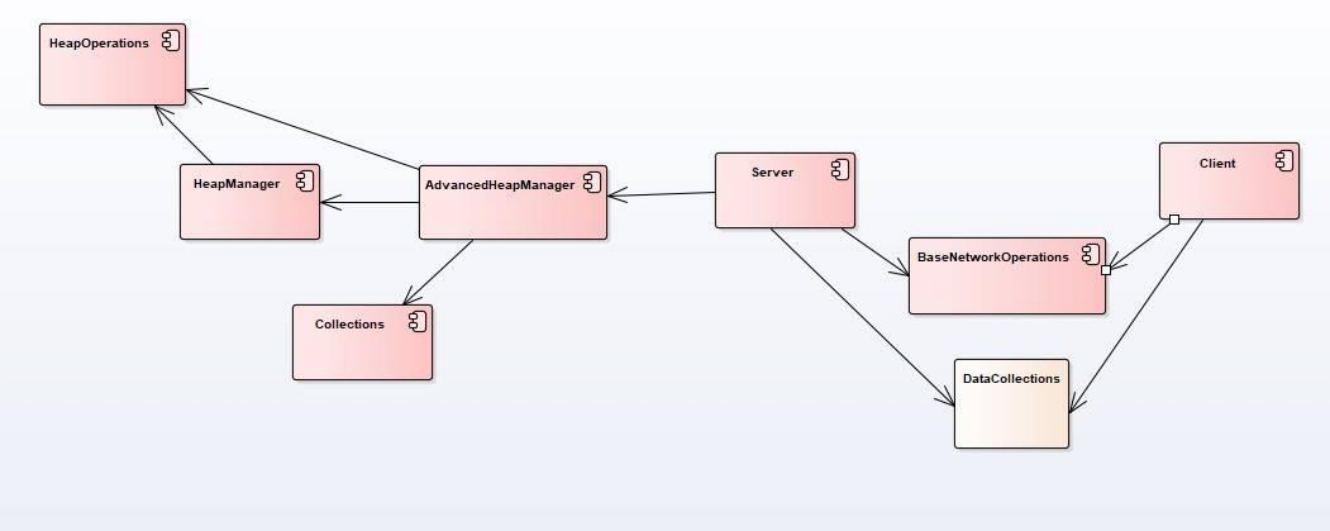
1. Rešavanje problema heap contention-a
2. Poboljšati perfomanse (vreme) koje je potrebno za alokaciju i dealokaciju memorije
3. Podržati uspešnu alokaciju i dealokaciju veće količine memorije
4. Pokazati da se implementirano rešenje ponaša onako kako se od njega očekuje, čak i u uslovima povećanih klijentskih zahteva

Funkcionalnosti implementirane u sklopu projekta su:

1. Kreiranje i razmena poruka nasumične dužine kroz računarsku mrežu
2. Alokacija (zauzimanje) memorije uz pomoć AHM-a
3. Dealokacija (oslobađanje) memorije uz pomoć AHM-a

2. Arhitektura rešenja

Slika 1. prikazuje dijagram komponenti koje čine projekat.



Slika 1. Arhitektura projekta

2.1. Heap manager arhitektura

Heap manager implementiran u ovom projektu se sastoji iz tri komponente:

- HeapOperations
- HeapManager
- AdvancedHeapManager

2.1.1. HeapOperations

Ova komponenta služi za direktan rad sa Heap-ovima, tj. za direktno rukovanje memorijom.

Ona izlaže operacije stvaranja i uništavanja Heap-ova, kao i operacije za alokaciju i dealokaciju memorije preuzete iz prethodno napravljenih Heap-ova.

2.1.2. HeapManager

HeapManager komponenta izlaže osnovnu funkcionalnost Heap manager-a. Njen osnovni zadatak jeste da drži sve napravljene Heap-ove na okupu i da na zahtev korisnika vrati trenutno selektovan Heap.

Selekcija Heap-a se vrši prema Round Robin tehnici, koja nalaže da sledeći izabrani element bude sledeći element u nizu, čime se na jednostavan način vrši dobra raspodela posla, odnosno Heap-ova. Definiše strukturu HeapManager, koja služi za centralizovanje podataka samog manager-a.

Komponenta izlaže funkcionalnosti inicijalizovanja i deinicijalizovanja strukture HeapManager-a i dodavanja novih Heap-ova u manager. Takođe, definiše i funkcije za odabir sledećeg Heap-a, kao i automatsko alociranje memorije date veličine iz sledećeg selektovanog heap-a, uz vraćanje Heap-a iz kojeg je preuzeta memorija. Pored toga, definise i funkciju za vraćanje memorije iz preuzetog Heap-a.

2.1.3. AdvancedHeapManager

Primaran cilj ove komponente je da pruži jednostavan interfejs korisnicima za korišćenje implementiranog Heap manager-a. Drugi zadatak ove komponente jeste da prati poreklo alociranih memorijskih blokova. Definiše strukturu rečnika, koji omogućava „wrap“-ovanje Heš tabele, uz kontrolisanje pristupa iz različitih niti. U rečnik unosi par [pokazivač na blok, Heap]. Komponenta pruža funkcije za inicijalizaciju i deinicijalizaciju manager-a, kao i funkcije advanced_malloc() i advanced_free() za alociranje, odnosno dealociranje memorijskih blokova.

2.1.5. Razlozi za odabir dizajna

Razlozi za odabir prethodno opisanog dizajna su sledeći:

1. Izdvajanjem HeapOperation komponente se olakšava promena u implementaciji rada sa memorijom, uz smanjenje prostiranja promena. Takođe, onemogućava se korisniku da koristi nepodržane operacije.

2. HeapManager komponenta je potpuno nezavisna komponenta. To omogućava ponovno iskorišćavanje komponente u drugim projektima. Izdvajanjem odvojene komponente se takođe olakšava razvoj celog projekta, jer je sastavljanje testova lakše, bug-ovi se lakše uočavaju i izmene se manje propagiraju.

2.2. Network (client-server) arhitektura

Network arhitektura implementirana u ovom projektu sastoji se od sledećih komponenti:

- Client
- Server
- BaseNetworkOperations
- DataNetworkCollections

2.2.1. Client

Komponenta koja modeluje ponašanje više klijenata uz pomoć niti. Klijenti simuliraju slanje poruka nasumičnih veličina, koje će se na serveru smeštati na heap koristeći AHM. Takođe, svaki od klijenata prihvata odgovor servera, koji je u vidu poruke nasumične veličine. Klijenti alociraju potrebnu količinu memorije, za smeštanje poruke koja predstavlja odgovor servera i na kraju dealociraju zauzetu memoriju. Alokacija i dealokacija potrebne memorije vrši se uz pomoć malloc() i free() funkcija koje su propisane C standardom.

2.2.2. Server

Komponenta koja prihvata i uspostavlja konekcije za svakog klijenta i smešta ih u zasebnu nit. Nakon toga, za poruku nasumične veličine, koju prihvati od klijenta, alocira potrebnu količinu memorije, generiše novu poruku nasumične veličine koju kao odgovor vraća klijentu i na kraju dealocira zauzetu memoriju. Server koristi implementirane funkcije koje su dostupne u okviru AHM-a.

2.2.3. BaseNetworkOperations

Komponenta koja se bavi komunikacijom između klijenta i servera. Nudi detekciju stanja socket-a, tj. proveru da li je socket u stanju read, write ili u stanju greške, kao i operacije slanja i primanja veće količine podataka.

2.2.4. DataNetworkCollections

Komponenta koja omogućava generisanje poruke nasumične veličine i popunjava je brojevima od 0 – 9 radi eventualnog prikaza.

2.2.5. Razlozi za odabir dizajna

Razlozi za odabir prethodno navedenog dizajna su sledeći:

1. Client i Server predstavljaju logički odvojene celine. Više client-a koriste usluge server-a koji se oslanja na implementaciju AHM-a pa ih je bilo neophodno razdvojiti na posebne komponente. Obe ove komponente raspolažu wrapp-ovanim metodama za slanje i prijem poruka koje omogućavaju slanje i prijem veće količine podataka.

2. NetworkOperations predstavlja komponentu koja obuhvata operacije vezane za razmenu podataka preko mreže i izdvojeno je u posebnu logičku celinu u cilju smanjenja redundancije koda i eventualne ponovne upotrebe ili proširenja funkcionalnosti vezanih za mrežnu komunikaciju.

3. DataNetworkCollections jeste komponenta koja se bavi samim podacima koji se razmenjuju preko mreže. Ideja je da ovaj projekat bude odvojena celina u cilju eventualnog dodavanja struktura podataka koje bi se slale preko mreže i na taj način obuhvatio kompletnu logiku vezanu za podatke.

3. Strukture podataka

Strukture podataka koje se koriste u okviru projekta AHM su:

- HeapManager
- HashTable
- HashNode
- Dictionary

3.1. HeapManager

```
typedef HANDLE Heap;

typedef struct manager_struct {
    Heap* heap_array; ///< niz heap-ova
    int heap_size; ///< maksimalna velicina "default" heap-a.
    int heap_count; ///< broj trenutno koriscenih heap-ova.
    int max_heaps; ///< maksimalan broj heap-ova.
    int current_heap; ///< index trenutno selektovanog heap-a.
    CRITICAL_SECTION manager_mutex; ///< propusnica kriticne sekcije
} HeapManager;
```

Slika 2. Izgled strukture HeapManager

HeapManager predstavlja strukturu koja lokalizuje sve potrebne podatke za funkcionisanje Heap manager-a.

Definise kritičnu sekciju (manager_mutex) koja omogućava da sve operacije nad HeapManager-om budu thread-safe.

Heap_array polje predstavlja niz svih mesta za heap-ove. Manager je sposoban da u run-time-u dodaje nove heap-ove, stoga se definiše polje heap_count koje govori koliko heap-ova je smešteno u manager. Heap-ovi mogu beskonačno da rastu, ali mogu i da budu ograničene veličine, a ta veličina se nalazi u heap_size polju.
Current_heap predstavlja indeks trenutno korišćenog Heap-a u nizu Heap-ova.

3.2. HashTable

```
typedef struct hash_table {
    HashNode** _table; ///< Bucket-i. Niz pokazivaca na elemente.
    int size; ///< Trenutna velicina tabele.
    int minimal_size; ///< Minimalna velicina tabele. Potrebno da bude prost broj.
    int entries; ///< Broj elemenata u tabeli.
    void*(*bucket_list_allocating_function)(int); ///< Pokazivac na funkciju koja alocira memoriju za bucket-e.
    void(*bucket_list_free_function)(HashNode**); ///< Pokazivac na funkciju koja oslobadja memoriju za bucket-e.
    void(*node_free_function)(HashNode*); ///< Pokazivac na funkciju koja oslobadja memoriju za jedan element.
    void*(*node_allocate_function)();///< Pokazivac na funkciju koja alocira memoriju za jedan element.
} HashTable;
```

Slika 3. Izgled strukture HashTable

HashTable struktura predstavlja implementaciju hash tabele. Tabela unosi kao vrednosti par pokazivač Heap, a kao ključ se koristi vrednost pokazivača.

Hash tabela je realizovana kao tabela ulančavanja. Ona sadrži niz bucket-a koji sadrže sve elemente na istom indeksu (vrednost hash-a za svaki element modulirana veličinom tabele), tako da svaki bucket predstavlja pokazivač na listu elemenata. Tabela se širi i skuplja u zavisnosti od faktora popunjenosti. Ako je faktor popunjenosti >= 0.75, dolazi do dupliranja tabele, dok kod faktora popunjenosti <= 0.25 tabela se skuplja za polovinu.

Tabela se skuplja samo ako je veća od minimalne veličine koja se prosleđuje pri inicijalizaciji. Zbog bolje distribuiranosti elemenata, tabela mora biti veličine prostog broja, o čemu se sama tabela brine.

Tabeli je pri inicijalizaciji takođe potrebno proslediti pokazivače na funkcije za alociranje i oslobađanje tabele, odnosno jednog pojedinačnog elementa.

3.3. HashNode

```
typedef struct hash_node {
    void* key; ///< Kljuc prema kojem se pretrazije HashTable.
    void* value; ///< Vrednost koja se smesta u HashTable.
    struct hash_node* next; ///< Pokazivac na sledeci element u bucket-u.
} HashNode;
```

Slika 4. Izgled strukture HashNode

HashNode predstavlja strukturu koja čini jedan element u hash tabeli. Za potrebe projekta, ključ u tabeli je pokazivač na memorijski blok, a vrednost je Heap u kojem se nalazi alocirana memorija.

3.4. Dictionary

```
typedef struct dict {
    HashTable* _table; ///< pokazivac na HashTable objekat.
    Heap _dict_heap; ///< Handle privatnog heap-a recnika koji sadrzi hash tabelu u sve njene elemente. Koristi
    CRITICAL_SECTION _cs; ///< Propusnica kriticne sekcije, koja omogucava thread-safe operacije nad recnikom.
} Dictionary;
```

Slika 5. Izgled strukture Dictionary

Dictionary je struktura koja predstavlja „wrapper“ za hash tabelu. S obzirom da hash tabela nije „thread-safe“, dictionary u sebi ima kritičnu sekciju, što daje „thread-safety“ svim operacijama nad tabelom. On takođe u sebi ima svoj privatan Heap, u koji se smeštaju svi podaci u tabeli (bucket-i i svi elementi).

3.5. Razlozi za korišćenje struktura podataka

Postoji više razloga za korišćenje gore navedenih podataka:

1. Hash tabela se koristi zbog brzog pristupa unikatnim elementima. To je naročitno pogodno za ovaj projekat, s‘obzirom da su pokazivači na memorijske blokove unikatni, a potreban je brz pristup pri oslobađanju prethodno zauzetog memorijskog bloka.

2. HashNode se koristi kako bi se ključ i vrednost koji se koriste u okviru same Hash tabele, objedinili u jedan element koji se čuva u tabeli, čime se olakšava pretraga tabele. Takođe, svaki HashNode sadrži i pokazivač na sledeći element u bucket-u, čime se automatski dobija pokazivač na narednu listu elemenata.

3. Dictionary se koristi kako bi se obezbedila “thread-safe” tabela tj. kako bi sve operacije nad tabelom bile “trade-safe”. Takođe, uvođenjem Dictionary-a, razvijena je optimalna struktura podataka, što značajno doprinosi perfomansama ovog projekta.

4. Rezultati testiranja AHM-a

Kako bi se proverilo da li je AHM uspešno implementiran, rešenje je testirano iz dva testna okruženja:

1. Test aplikacija koja direktno koristi AHM i meri vreme potrebno da se alocira i dealocira 2GB memorije iz 1, 2, 5, 10 i 50 niti respektivno.

2. Testni server koji od više klijenata prima poruke nasumične veličine i šalje im odgovore nasumične veličine, pri čemu koristi AHM. Za potrebe testiranja, broj klijenata je ograničen na 10, dok je maksimalna veličina poruke 100MB.

4.1. Razlozi za odabir navedenih testova

Postoji više razloga zašto su izabrani ovi testovi:

1. Prvi test treba da pokaže razliku između podrazumevanih malloc() i free() funkcija i advanced_malloc() i advanced_free() funkcija. Razlika se ogleda u vremenu koje je potrebno za alociranje i dealociranje 2GB memorije

2. Drugi test treba da pokaže kako se sistem ponaša kada se veća količina podataka šalje putem mreže, od strane više klijenata. Pri tome, server koristi AHM za smeštanje primljenih podataka.

3. Veća količina memorije koja je potrebna da se zauzme u oba testa, izabrana je kako bi se prikazale značajne razlike u vremenu koje je potrebno za alociranje i dealociranje memorije. Naravno, razlike su primetne i prilikom alociranja i dealociranje manje količine memorije, ali ne u većoj meri, što će biti prikazano u okviru drugog test scenarija.

4.2. Test aplikacija koja direktno koristi AHM

Tabela 1 prikazuje vreme koje je potrebno da se alocira i dealocira 2GB memorije iz 1, 2, 5, 10 i 50 niti respektivno. Memorija je prvo alocirana i dealocirana uz pomoć funkcija malloc() i free() koje su propisane C standardom, a zatim uz pomoć advanced_malloc() i advanced_free() funkcija, koje su implementirane u okviru AHM-a.

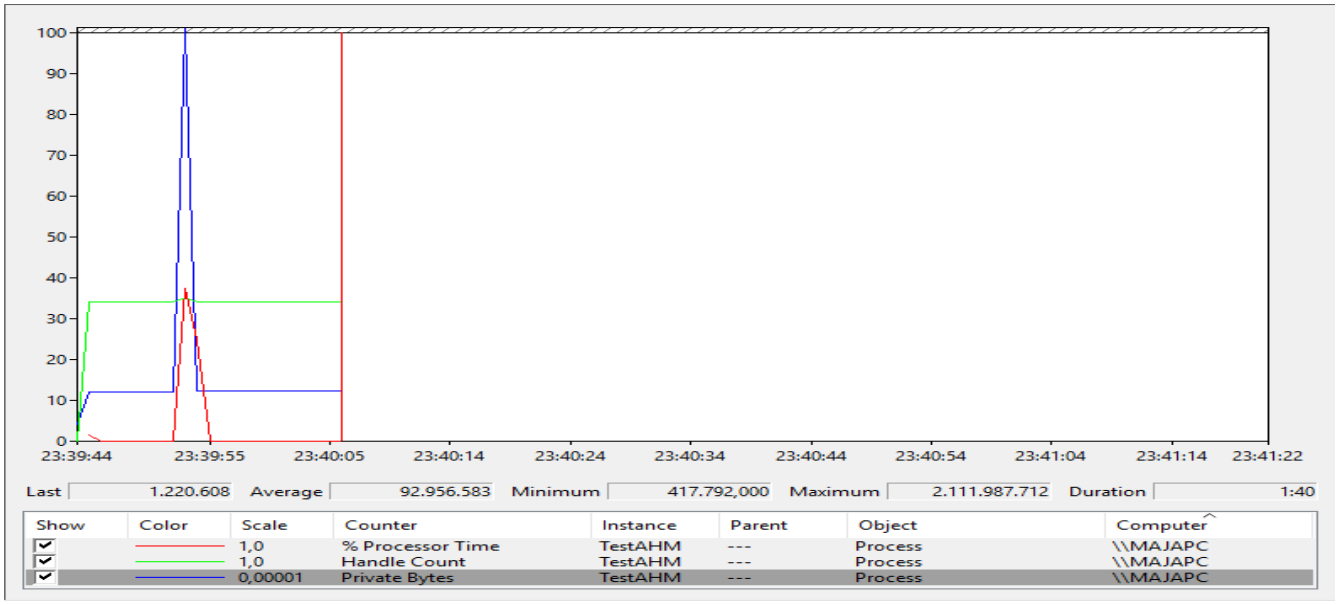
Redni broj testa	Malloc(), free()		Advanced_malloc(), advanced_free()	
	Broj tredova	Vreme [s]	Broj tredova	Vreme [s]
1	1	3.190	1	0.700
2	2	3.493	2	1.274
3	5	5.041	5	3.297
4	10	22.930	10	8.314
5	20	135.567	20	16.619
6	50	24.591	50	4.862

Tabela 1. Rezultati merenja

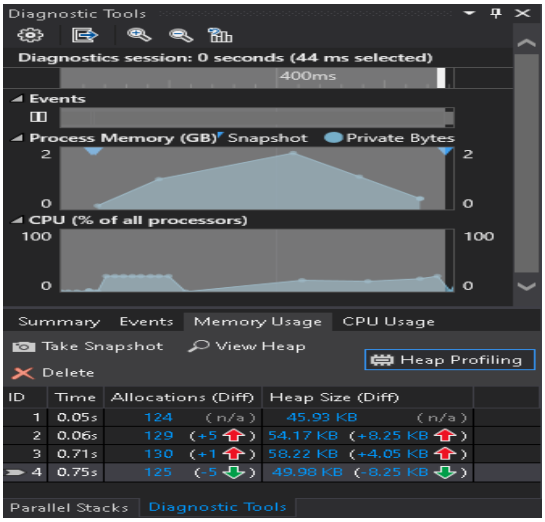
4.2.1. Perfomance Monitor i Visual Studio Profiler

U ovom poglavlju, biće prikazani rezultati zauzimanja 2GB memorije iz 1, 2, 5, 10, 20, 50 niti respektivno u okviru Perfomance Monitor-a i Visual Studio Profiler-a. Testovi koji su razmatrani, odnose se na upotrebu advanced_malloc() i advanced_free() funkcija, koje su implementirane u okviru AHM-a.

4.2.1.1. Zauzimanje 2GB memorije iz jedne niti

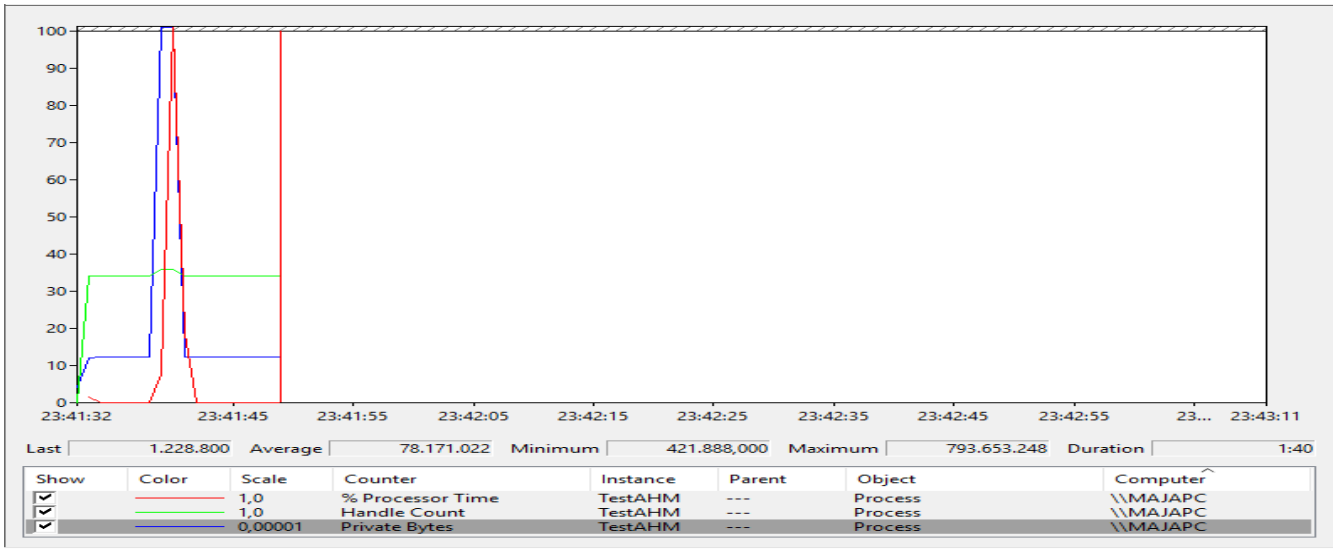


Slika 6. Rezultati Perfomance Monitor-a

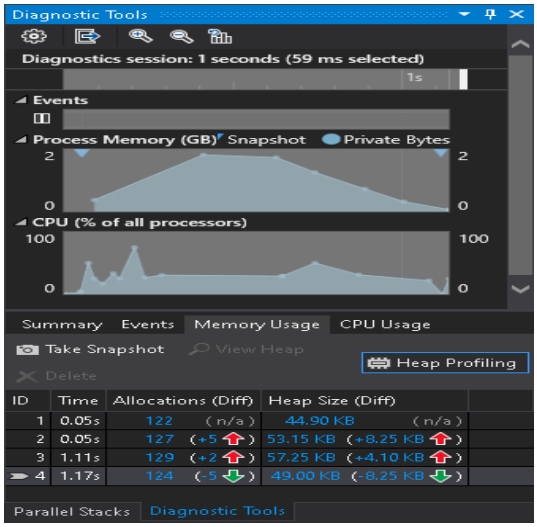


Slika 7. Rezultati Visual Studio Profiler-a

4.2.1.2. Zauzimanje 2GB memorije iz dve niti

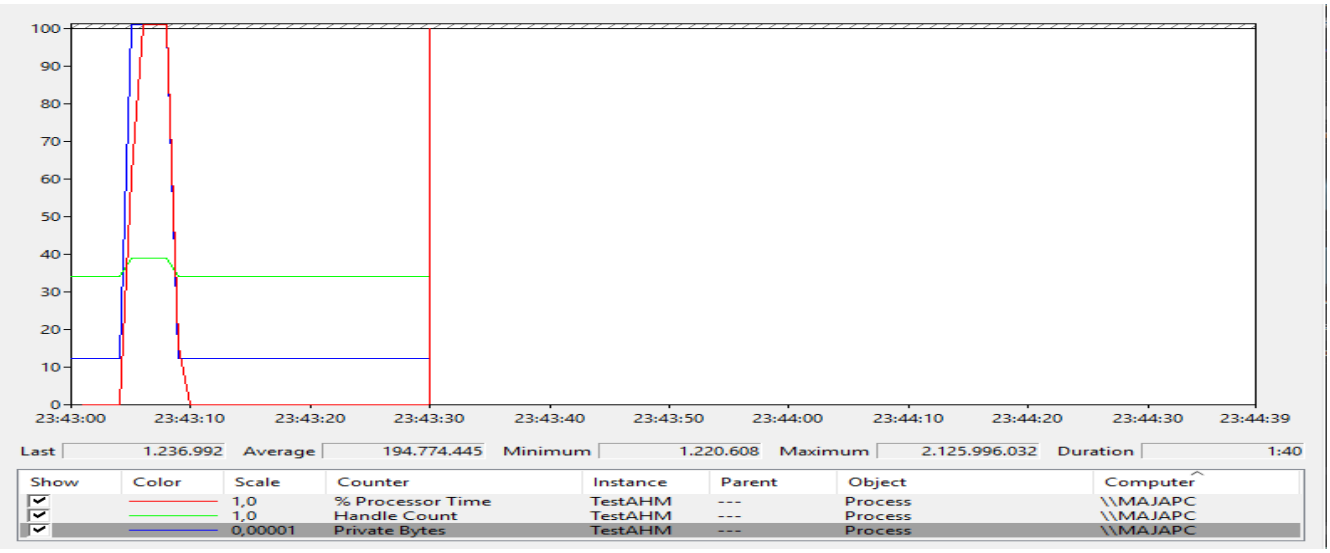


Slika 8. Rezultati Perfomance Monitor-a

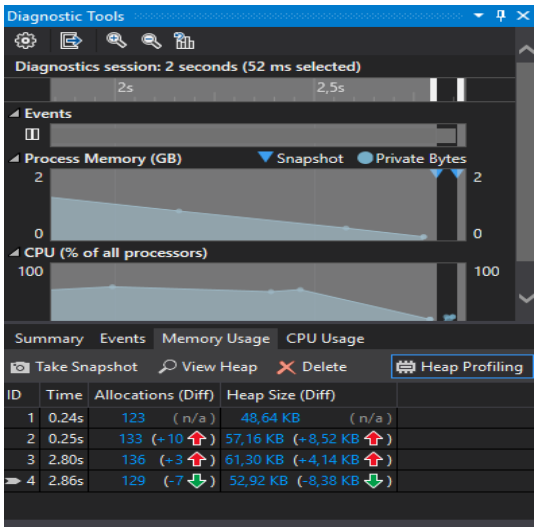


Slika 9. Rezultati Visual Studio Profiler-a

4.2.1.3. Zauzimanje 2GB memorije iz pet niti

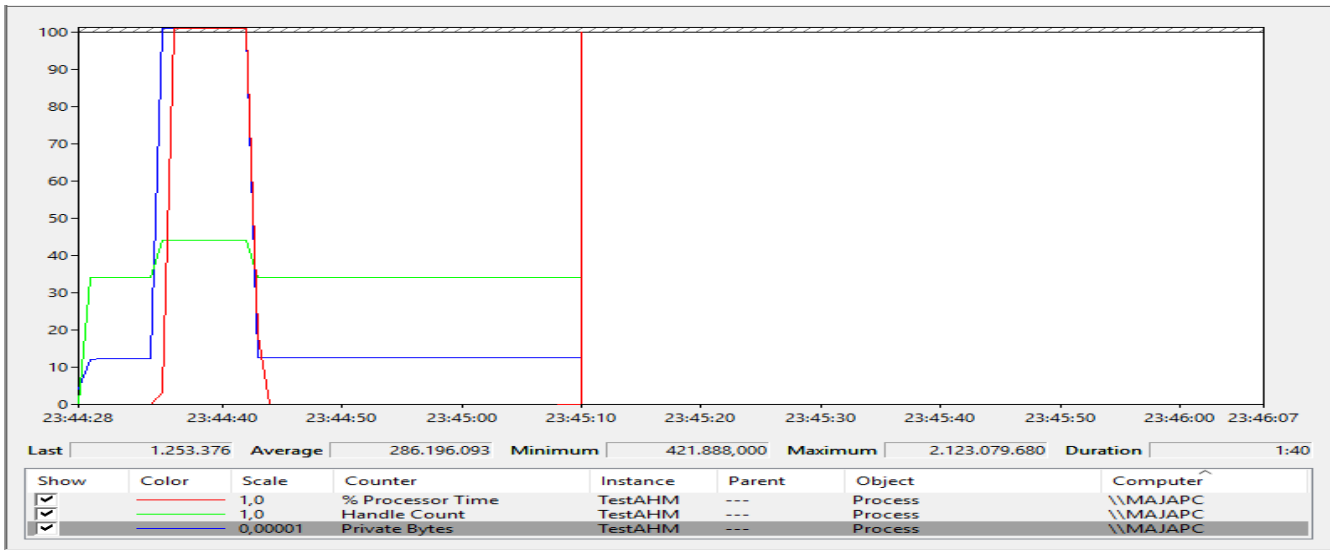


Slika 10. Rezultati Perfomance Monitor-a

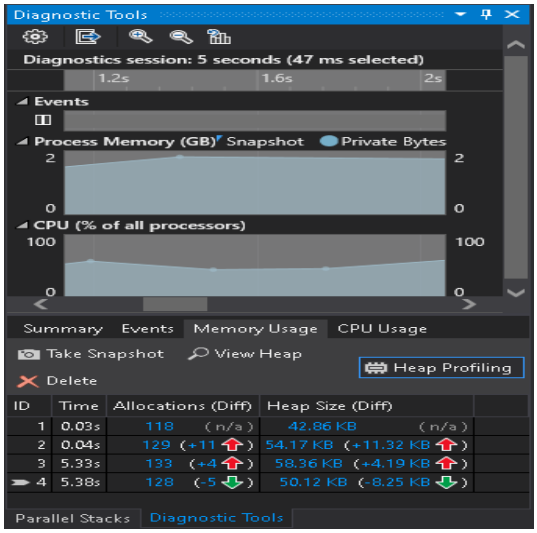


Slika 11. Rezultati Visual Studio Profiler-a

4.2.1.4. Zauzimanje 2GB memorije iz deset niti

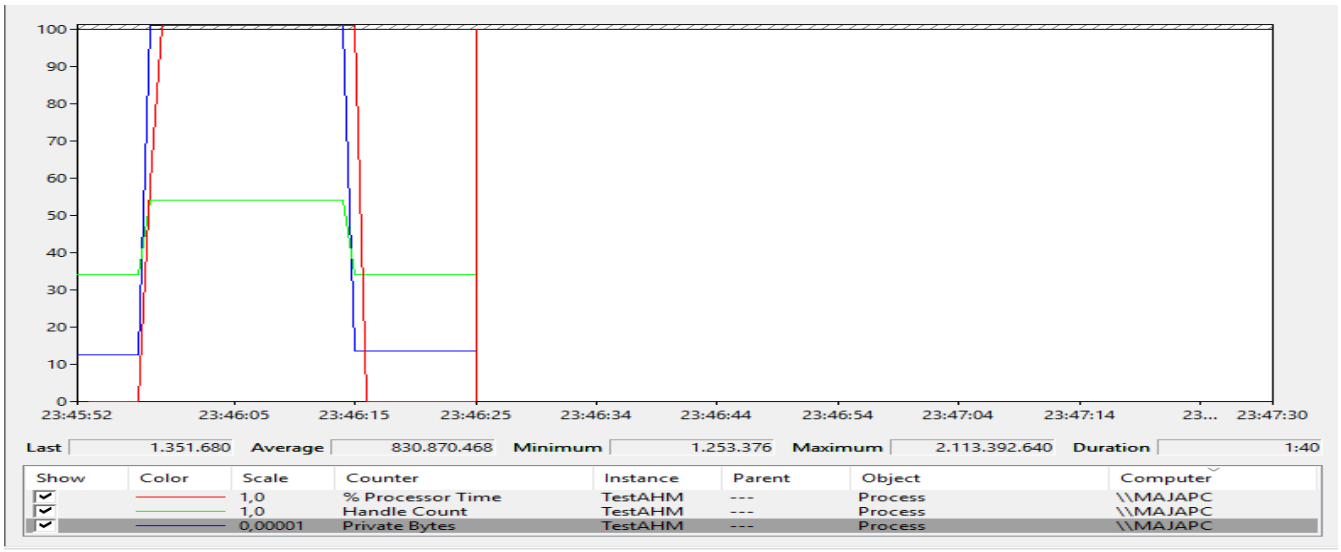


Slika 12. Rezultati Perfomance Monitor-a

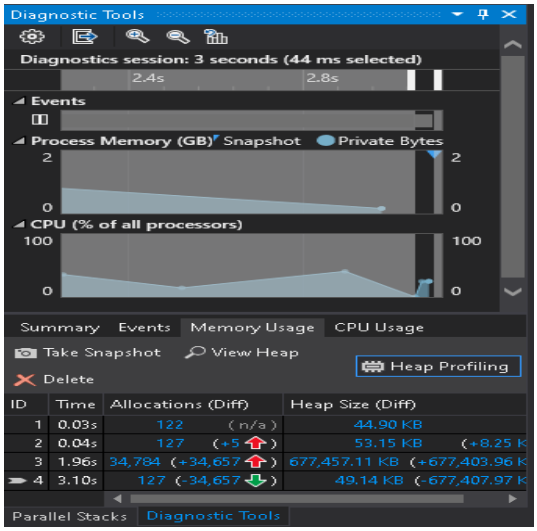


Slika 13. Rezultati Visual Studio Profiler-a

4.2.1.5. Zauzimanje 2GB memorije iz dvadeset niti

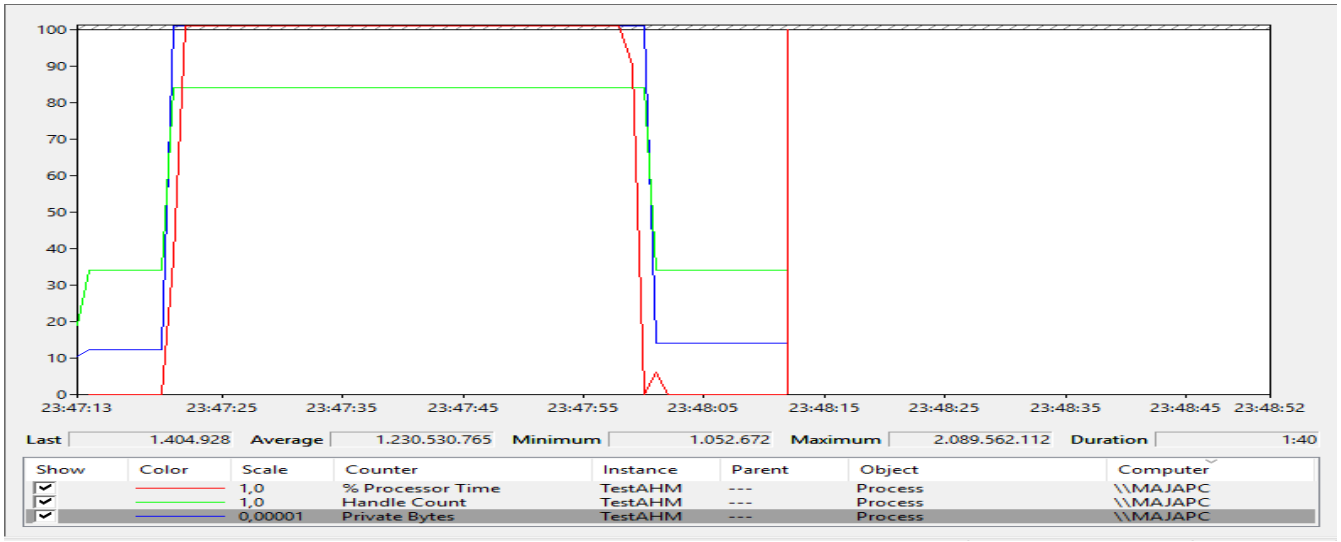


Slika 14. Rezultati Perfomance Monitor-a

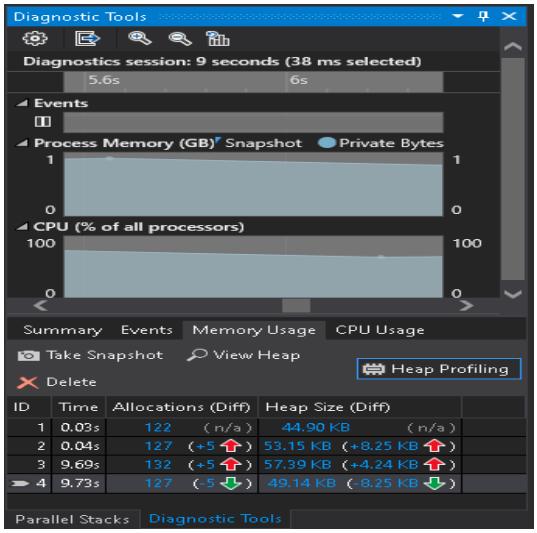


Slika 15. Rezultati Visual Studio Profiler-a

4.2.1.6. Zauzimanje 2GB memorije iz pedeset niti



Slika 16. Rezultati Perfomance Monitor-a



Slika 17. Rezultati Visual Studio Profiler-a

4.3. Testni server koji opslužuje više klijenata i koristi AHM

Tabela 2 prikazuje vreme koje je potrebno da se alocira i dealocira određena količina memorija. Količina memorije zavisi od veličine poruke koju svaki od deset klijenata šalje na server, kao i od veličine poruke koju server šalje klijentima kao odgovor.

Test je realizovan tako što server koristi AHM za alociranje i dealociranje memorije uz pomoć advanced_malloc() i advanced_free() funkcija, dok klijent za alociranje i dealociranje memorije koristi malloc() i free() funkcije koje su propisane C standardom.

Redni broj klijenta	Klijent		Server	
	malloc(), free()		advanced_malloc(), advanced_free()	
	Količina podataka [MB]	Vreme [s]	Količina podataka [MB]	Vreme [s]
1	40	0.055	70	0.002
2	50	0.056	40	0.001
3	40	0.043	40	0.001
4	80	0.095	80	0.003
5	20	0.022	60	0.002
6	90	0.114	70	0.002
7	30	0.044	30	0.001
8	60	0.072	20	0
9	80	0.096	90	0.004
10	50	0.056	80	0.003

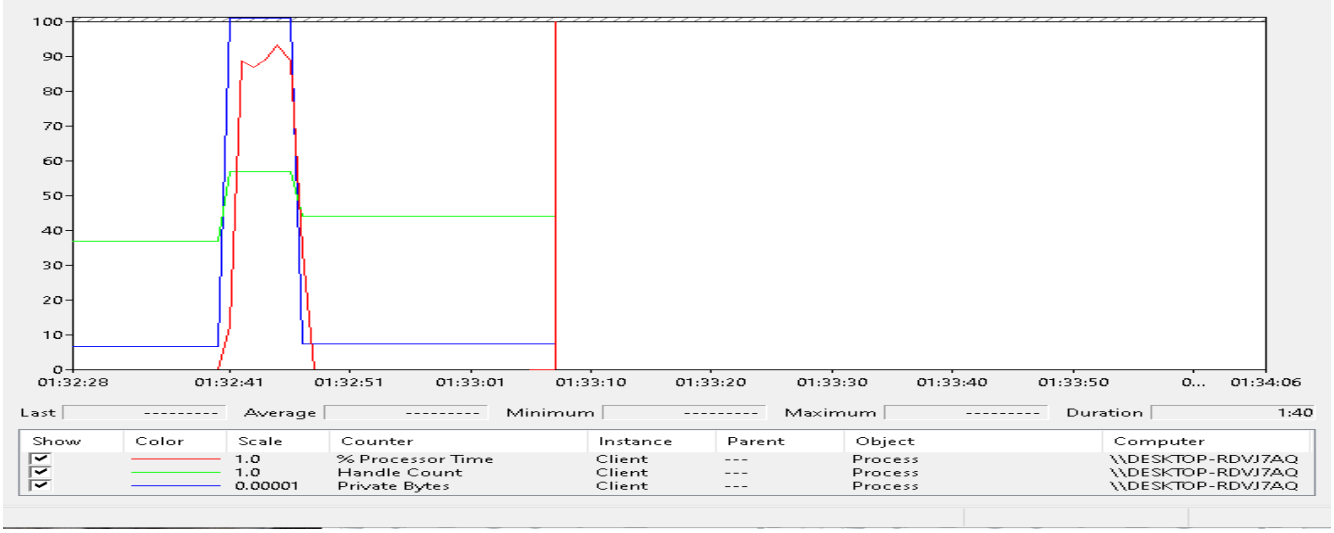
Tabela 2. Rezultati merenja

4.3.1. Perfomance Monitor i Visual Studio Profiler

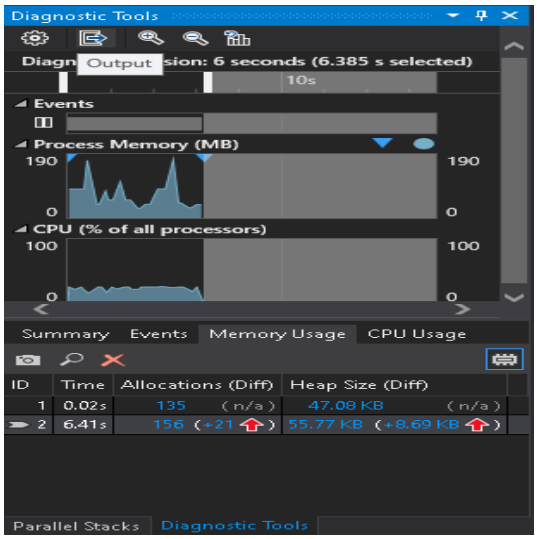
U ovom poglavlju, biće prikazani rezultati alociranja i dealociranje određene količine memorije u okviru Perfomance Monitor-a i Visual Studio Profiler-a. Testovi koji su razmatrani, odnose se na upotrebu advanced_malloc() i advanced_free() funkcija na serverskoj strani, odnosno na upotrebu malloc() i free() funkcija na klijentskoj strani.

Samim tim, posebno će se razmatrati klijentska i serverska strana.

4.3.1.1. Alociranje i dealociranje određene količine memorije na klijentskoj strani

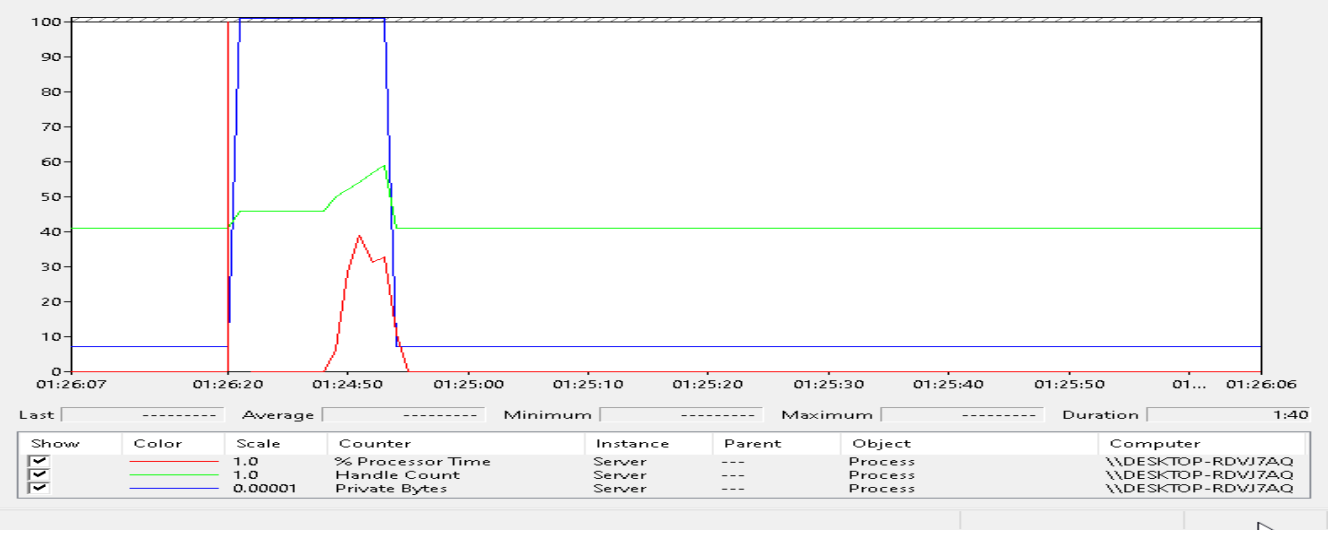


Slika 18. Rezultati Perfomance Monitor-a

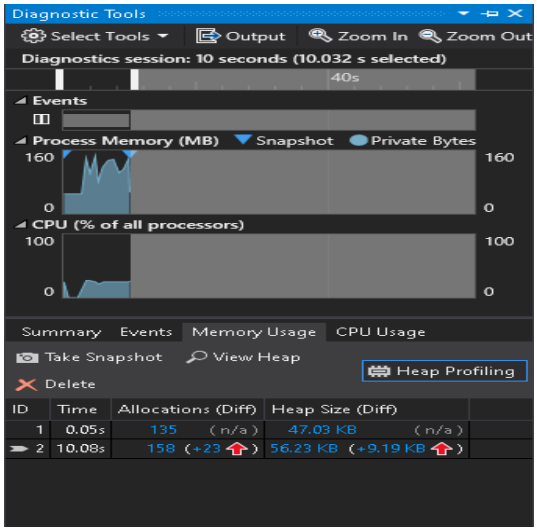


Slika 19. Rezultati Visual Studio Profiler-a

4.3.1.2. Alociranje i dealociranje određene količine memorije na serverskoj strani



Slika 20. Rezultati Perfomance Monitor-a



Slika 21. Rezultati Visual Studio Profiler-a

5. Zaključak

Na osnovu testova prikazanih na prethodnim stranama, može se zaključiti sledeće:

1. Funkcije koje su implementirane u okviru AHM-a su značajno brže prilikom alociranje i dealociranje memorije u odnosu na malloc() i free() funkcije koje su propisane C standardom

2. U toku rada aplikacije postoji zanemarivo curenje memorije

3. Prilikom prenosa poruka od klijenata ka serveru i od servera ka klijentima, ne dolazi do gubitaka podataka (sve poruke uspešno stignu na odredište)

4. Sistem uspešno reaguje kada postoji veći broj klijenata koji šalju veću količinu podataka

Ovakvi rezultati su očekivani iz sledećih razloga:

1. Funkcije za alociranje i dealociranje memorije, koje su implementirane u okviru AHM-a, očekivano su brže od funkcija malloc() i free() koje su propisane C standardom, zato što rešavaju problem heap contention-a. Prilikom pokretanja većeg broja niti, svaka nit dobija memoriju od nekog drugog heap-a na osnovu Round Robin tehnike, za razliku od funkcija malloc() i free() koje isključivo koriste podrazumevani heap procesa

2. S' obzirom da se u test scenarijima zauzima pozamašna količina memorije [2GB, 1GB], može se reći da je zanemarivo curenje memorije od 2-10 KB. Treba napomenuti, da na curenje memorije dosta utiče upotreba standardnih biblioteka

3. Do gubitaka podataka ne dolazi zbog upotrebe TCP transportnog protokola, koji obezbeđuje pouzdanu razmenu podataka

4. U slučaju većeg broja niti, očekivano je da vreme alokacije i dealokacije memorije bude veće. Međutim, za test slučaj sa 50 niti, može se primetiti relativno kratko vreme koje je potrebno za alociranje i dealociranje memorije. Potencijalni razlog za to jeste što sam operativni sistem vrši optimističnu alokaciju memorije. Optimistična alokacija memorije znači da prilikom alokacije dolazi samo do rezervisanja memorijskog prostora, ali ne i samog commit-ovanja memorije od strane operativnog sistema, što dovodi do vreoma brzih alokacija.

6. Potencijalna unapređenja

Rad aplikacije se može potencijalno unaprediti na sledeće načine:

1. HashTable organizovati da radi u principu otvorenog adresiranja

   Kod otvorenog adresiranja se svi elementi nalaze u nizu, što omogućava brži pristup svakom pojedinačnom elementu. Takođe, na taj način se smanjuje memorijsko zauzeće. Mana ovakvog pristupa jeste to što je potrebno imati veoma dobre hash funkcije, jer su kolizije veoma skupe i najčešće je potrebno koristiti više hash funkcija za dobijanje indeksa.

2. Unaprediti hash funkciju

   Iako je hash funkcija koja je implementirana u projektu sasvim zadovoljavajućeg performansa, veoma je teško naći perfektnu hash funkciju. Međutim, verovatno se može naći hash funkcija koja brže računa hash, sa veoma malo kolizija.

3. U HeapManager dodati funkcionalnost izbacivanja Heap-ova iz upotrebe

   Na taj način bi se još više generalizovao rad Heap manager-a.