Industrijski komunikacioni protokoli u infrastrukturnim sistemima

Projektna dokumentacija

Load Balancer

- Dušan Borovićanin PR56/2020
- Aleksa Bugarinović PR14/2020

Uvod

LoadBalancer je distribuirani sistem dizajniran za efikasno upravljanje radnim zadacima i raspodelu opterećenja između radnika (workers). Sastoji se od tri ključne komponente: LoadBalancer-a, Radnika (Workers) i Klijenta (Client). LoadBalancer koordinira radnicima i upravlja redom radnih zadataka, dok klijenti komuniciraju sa LoadBalancer-om kako bi poslali i dobili rezultate svojih radnih zadataka. Koristi se za ravnotežu opterećenja između više servera kako bi se poboljšala dostupnost, skalabilnost i performanse sistema.

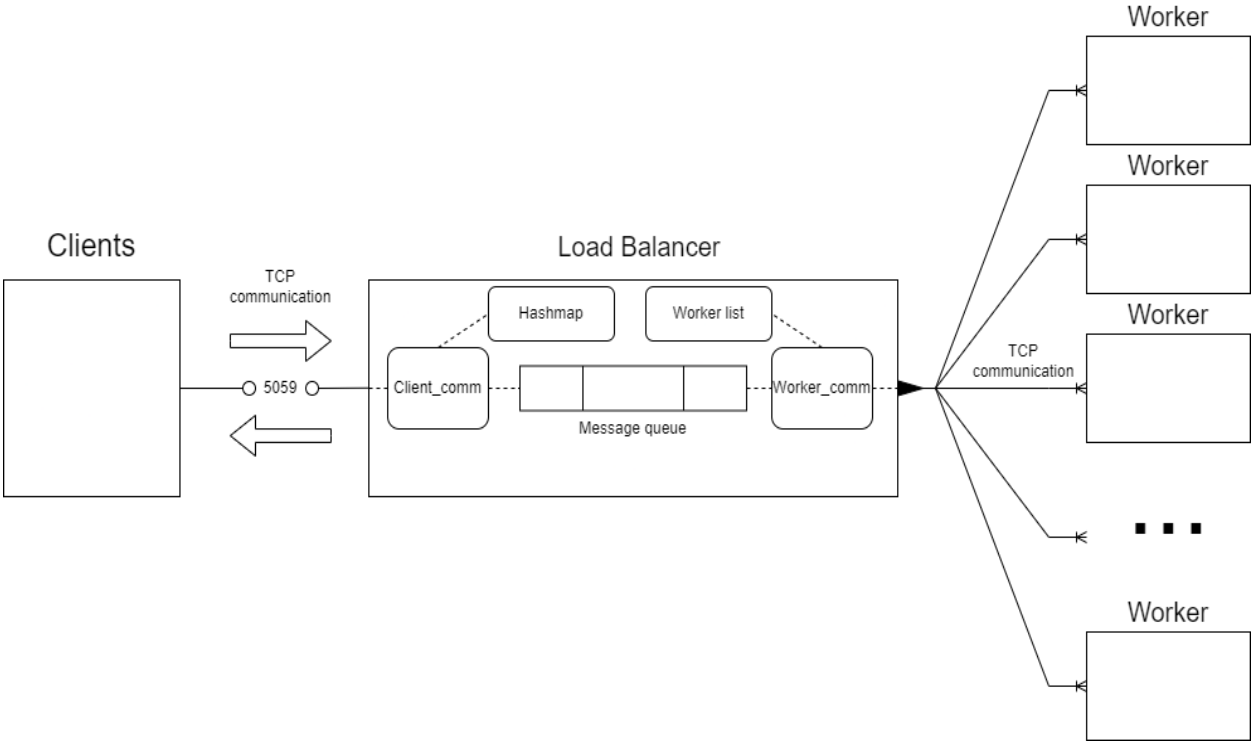
Neki od glavnih ciljeva load balancera uključuju :

1. **Povećanje dostupnosti**: Load balancer igra ključnu ulogu u poboljšanju dostupnosti sistema. Ako jedan od servera doživi kvar ili postane nedostupan, load balancer može preusmeriti zahteve korisnika na druge ispravne servere. Ovo povećava otpornost sistema na kvarove i smanjuje vreme nedostupnosti.
2. **Povećanje skalabilnosti**: Load balancer omogućuje jednostavno dodavanje ili uklanjanje servera iz klastera kako bi se prilagodila potrebama rastućeg opterećenja. To olakšava horizontalno skaliranje, gdje se kapacitet sistema povećava dodavanjem novih resursa, umesto vertikalnog skaliranja, gdje se pojedini resursi nadograđuju.
3. **Optimizacija performansi**: Load balancer može optimizovati performanse sistema usmeravanjem zahteva prema serverima sa najmanjim opterećenjem ili najbližim korisnicima. To može uključivati pravilno raspoređivanje zahteva ili implementaciju dodatnih tehnika, poput SSL offloadinga ili keširanja sadržaja.
4. **Sigurnost**: Load balancer može pružiti nivo sigurnosti tako da filtrira ili obustavlja određene vrste podataka kroz mrežu, prati i sprečava određene vrste napada, odnosno pruža SSL terminaciju radi bolje sigurnosti komunikacije.
5. **Praćenje i dijagnostika**: Load balancer može pružiti informacije o stanju sistema, praćenje performansi i dijagnostiku problema. Ovo pomaže administratorima u održavanju i optimizaciji sistema.

Ciljevi zadatka

Cilj zadatke jeste razvijanje servisa koji vrši obradu merenja pametnih brojila. Korisnik se povezuje na server odnosno Load Balancer koji sluša na portu 5059 i zatim mu šalje potrošnju sa svog električnog brojila izraženu u kW-h. Dovoljno je poslati samo brojnu vrednost potrošnje. Load Balancer dobijene vrednosti merenja prosleđuje radniku odnosno Worker-u. Worker podatke obrađuje tako što na osnovu primljenih podataka prvo određuje zonu u kojoj je korisnik (zelena, plava ili crvena), a zatim računa potrošnju električne energije izražene u RSD. Poruku koji Worker treba da dostavi klijentu prvo šalje Load Balancer-u, a Load Balancer je prosleđuje na krajnju destinaciju, odnosno klijentu.

Dizajn



Komunikacija između klijenata i Load Balancer-a, kao I između Worker-a i LoadBalancer-a se odvija preko TCP protokola. Load Balancer pri pokretanju kreira, odnosno inicijalizuje hashmap-u, listu radnika (slobodnih i zauzetih) i queue, odnosno red. U hashmap-i se čuvaju podaci o klijentima prijavljenim na Load Balancer. Liste Worker-a (slobodni i zauzeti) sadrže informacije o aktivnim radnicima. Queue služi za čuvanje poruke koju klijent odnosno radnik šalje. Kada je Load Balancer pokrenut, on je spreman da prihvati nove klijente na portu 5059.

Strukture podataka

Programski jezik C ne sadrži strukture podataka koje su potrebne za optimalno rešenje ovog problema. Za potrebe ove aplikacije kreirane su sledeće tri strukture podataka :

1. **HashMap** – struktura podataka koja omogućava efikasno mapiranje ključeva na vrednosti. Ključevi moraju biti jedinstveni, a svaki ključ je povezan s određenom vrednošću. Implementacija HashMap obično uključuje upotrebu heš funkcije za mapiranje ključeva na određene lokacije u memoriji.

```c
unsigned int hash(char* name) {
    unsigned int hash_val = 5381;
    int c;

    while ((c = *name++)) {
        hash_val = ((hash_val << 5) + hash_val) + c;
    }

    return hash_val % MAX_ELEM;
}
```

Ova implementacija hash funkcije koristi jednostavan algoritam poznat kao "djb2" hash algoritam i često koristi zbog svoje jednostavnosti i relativno dobre distribucije heš vrednosti.

Aplikacija koristi HashMap-u za čuvanje podataka o klijentima. Korišćenje baš ove strukture podataka je je pogodno iz više razloga :

1. **Brza pretraga i pristup podacima** : Heširanje ključeva omogućava direktno mapiranje ključa na odgovarajuće mesto u memoriji, što rezultuje brzim pristupom podacima.
2. **Jedinstveni ključevi** : Hash mape zahtevaju jedinstvenost ključeva, što znači da svaki klijent može biti identifikovan jedinstvenim ključem. Ovo je korisno u situacijama gde je potrebno brzo prepoznavanje i pristupanje podacima o određenom klijentu.
3. **Optimizacija prostora** : Hash mape omogućavaju efikasno upravljanje prostorom, jer se podaci smeštaju samo na mesta gde je to potrebno, a ne u kontinuiranom nizu kao u **nizovima**.
4. **Dinamičko proširivanje** : Implementacija hash mape omogućava dinamičko proširivanje kako bi se nosile sa povećanjem broja klijenata bez značajnog gubitka performansi.

2. **List** - struktura podataka koja omogućava skladištenje i organizaciju podataka u linearnom redosledu. Za razliku od niza (array), gde su elementi smešteni u kontinuiranom bloku memorije, elementi liste su smešteni na različitim mestima u memoriji i povezani pokazivačima.

   Liste su fleksibilne i omogućavaju efikasno dodavanje i brisanje elemenata, ali imaju i nešto složeniju strukturu i troše više memorije od **nizova**. Izbor između nizova i povezanih listi u C-ju zavisi od specifičnih zahteva aplikacije i operacija koje se izvode nad podacima.

```
void init_list(list** l) {
    *l = (list*)malloc(sizeof(list));
    (*l)->head = NULL;
    (*l)->current = NULL;
    (*l)->tail = NULL;
    InitializeCriticalSection(&(*l)->cs);
}
```

Ovu strukturu podatka u aplikaciji koristimo za čuvanje informacija o slobodnim i zauzetim radnicima. Korišćenje baš ove structure podataka pogodno je iz više razloga :

1. **Dinamičko dodavanje i uklanjanje radnika** : Liste omogućavaju lako dodavanje i uklanjanje elemenata, što je korisno kada se radnici priključuju ili odjavljuju iz sistema, kakav slučaj imamo i u ovoj aplikaciji. Upravljanje slobodnim i zauzetim radnicima zahteva dinamičko upravljanje listom, a liste nude efikasne operacije dodavanja i brisanja čvorova.
2. **Efikasno praćenje slobodnih radnika** : Liste omogućavaju efikasno praćenje slobodnih radnika. Kada se radnik oslobodi (postane slobodan), može se jednostavno dodati na kraj liste slobodnih radnika. Ovo omogućava brz pristup prvom slobodnom radniku prilikom dodele zadatka.
3. **Brza pretraga i raspodela radnika** : Povezane liste pružaju brz pristup elementima, što je važno za brzu pretragu i raspodelu radnika. Na primer, Load Balancer može brzo pronaći slobodnog radnika tako što će uzeti prvi element iz liste slobodnih radnika.

4. **Jednostavna implementacija u C-ju:** Liste imaju relativno jednostavnu implementaciju u jeziku C, što ih čini prikladnim izborom za mnoge implementacije, uključujući i implementaciju ove aplikacije. Pravilno korišćenje pokazivača omogućava efikasno upravljanje memorijom i promenama u listi.

3. **Queue** – (red) struktura podataka koja se koristi za organizaciju elemenata prema principu "prvi unutra, prvi napolje" (First In, First Out - **FIFO**). Ovo znači da prvi element koji je dodat u red će biti prvi koji će biti uklonjen.

   Red se često implementira korišćenjem niza (array) ili povezane liste, i može se koristiti u različitim situacijama gde je važno održavati redosled dolaska elemenata.

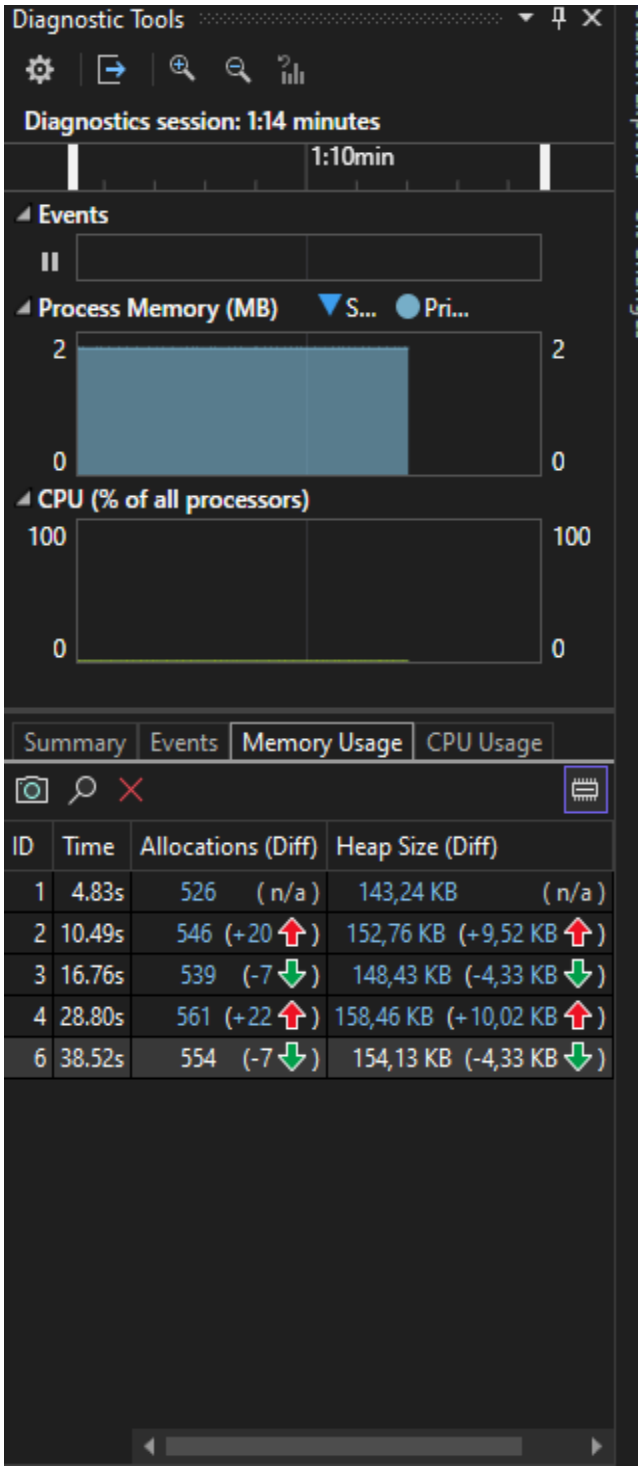
```
void create_queue(int capacity) {
    queue* newQueue = (queue*)malloc(sizeof(queue));
    newQueue->front = 0;
    newQueue->rear = capacity - 1;
    newQueue->capacity = capacity;
    newQueue->currentSize = 0;

    //create array of messages
    newQueue->messageArray = (messageStruct**)malloc(sizeof(messageStruct*) * capacity);

    for (int i = 0; i < capacity; i++) {
        newQueue->messageArray[i] = NULL;
    }

    InitializeCriticalSection(&newQueue->cs);
    hSemaporeQueueEmpty = CreateSemaphore(0, capacity, capacity, NULL);
    hSemaporeQueueFull = CreateSemaphore(0, 0, capacity, NULL);

    q = newQueue;
}
```

Korišćenje reda (Queue) u aplikaciji LoadBalancer kod čuvanja poruka koje se šalju između klijenta i radnika može biti pogodno iz nekoliko razloga:

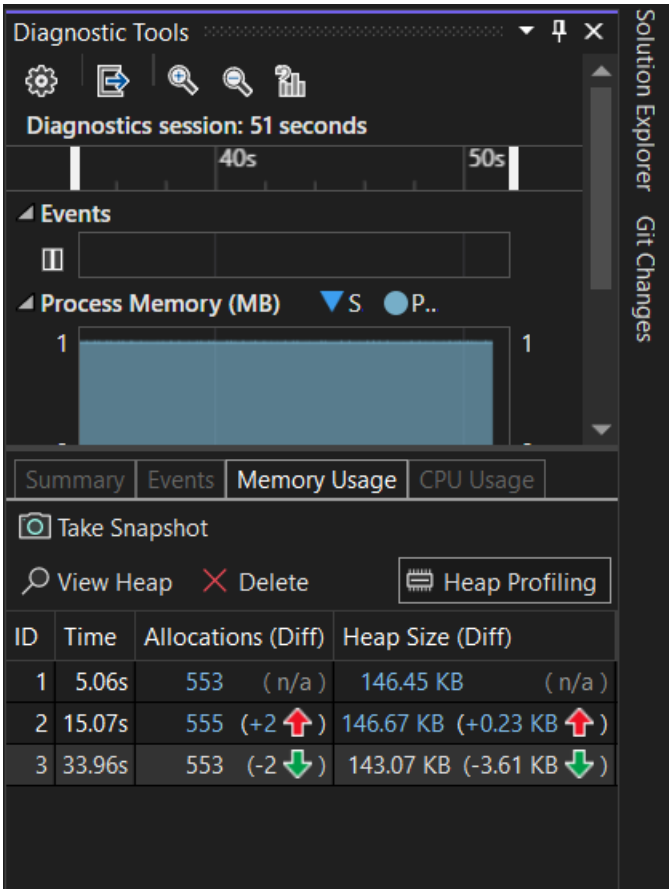
1. **Princip FIFO (First In, First Out)** : Red omogućava čuvanje poruka prema principu "prvi unutra, prvi napolje" (FIFO). Ovo znači da poruke koje stignu prve u red će biti prve poslate radnicima. Ovaj redosled je često važan u scenarijima gde se poruke šalju u specifičnom redosledu i gde je bitno održati stabilnost poruka.
2. **Ravnoteža opterećenja (Load Balancing)** : Red služi kao centralno skladište poruka koje čekaju na obradu od strane radnika. Load balancer može ravnomerno raspoređivati poruke radnicima iz reda, omogućavajući efikasno upravljanje opterećenjem i izbegavajući preopterećenje određenih radnika.
3. **Pouzdana dostava**: Korišćenje reda može poboljšati pouzdanost sistema jer poruke koje stignu u red ostaju tamo dok se ne dostave radnicima. Ako jedan radnik privremeno postane nedostupan ili ne može trenutno da obradi poruku, poruka će ostati u redu dok se radnik ne vrati u funkcionalno stanje.
4. **Lakše praćenje i dijagnostika** : Korišćenje reda olakšava praćenje i dijagnostiku sistema. Load balancer i radnici mogu pristupiti redovima kako bi videli koje poruke čekaju obradu, što može biti korisno za analizu performansi i rešavanje problema.

Sve ove karakteristike čine **red** pogodnim za upotrebu u aplikacijama LoadBalancer, posebno u situacijama gde se obrada poruka vrši asinhrono i gde je potrebno održavati određeni redosled poruka.

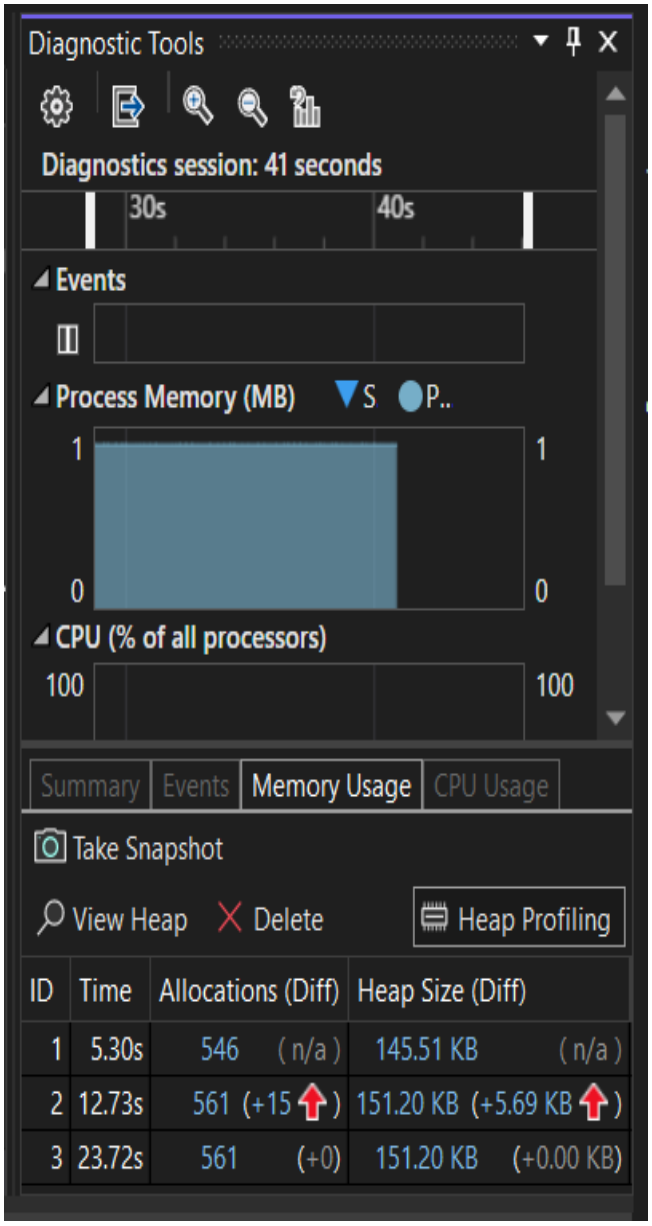
Rezultati testiranja



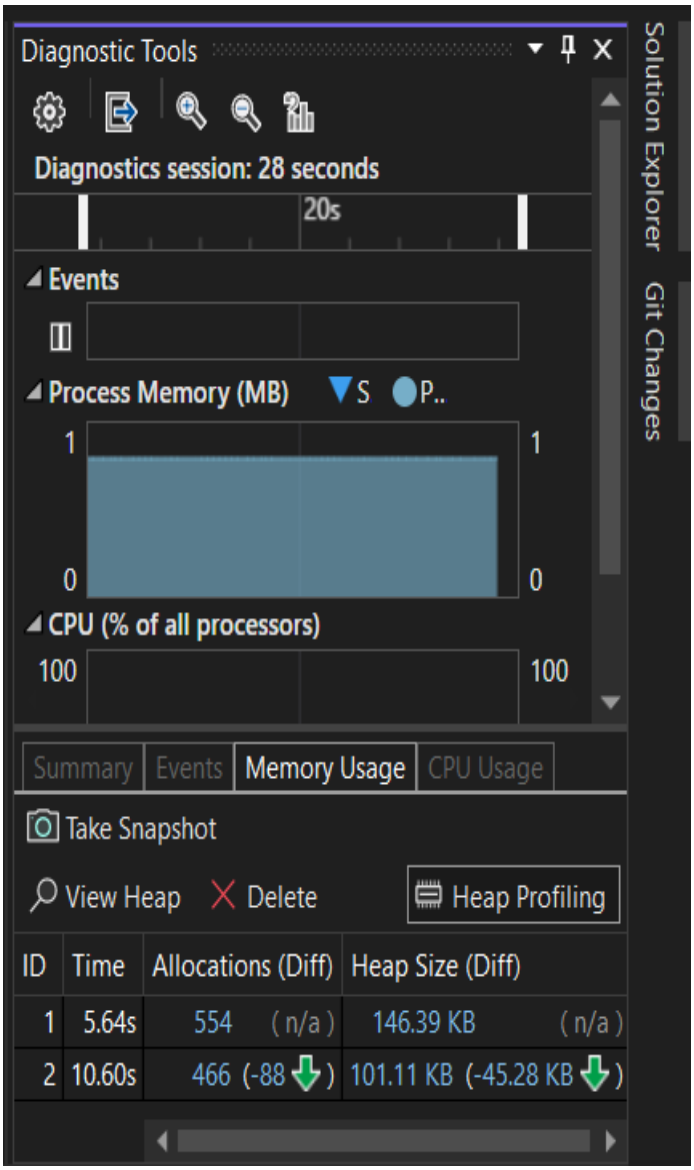
1. Pokrenut program u standardnom režimu
2. Dodat novi radnik
3. Uklonjen radnik (postoji već jedan slobodan)
4. Dodat novi radnik
5. Uklonjen radnik (postoji već jedan slobodan)



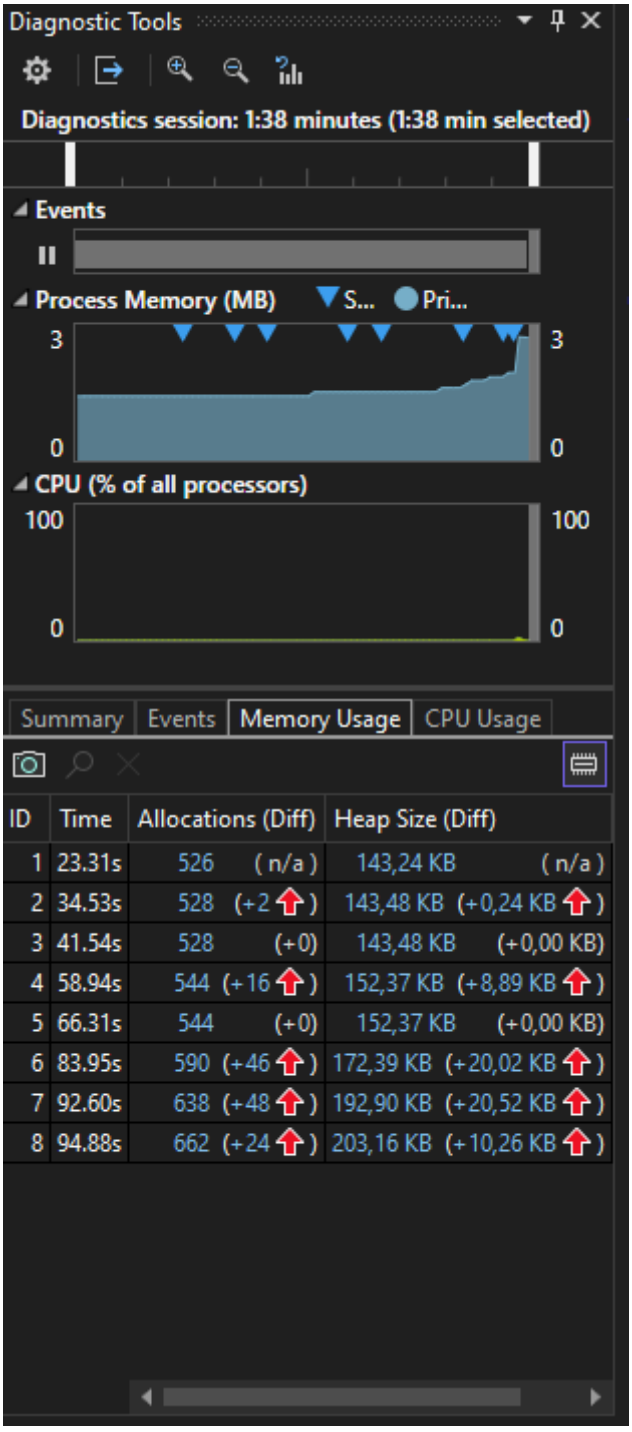
Dodavanje i brisanje liste



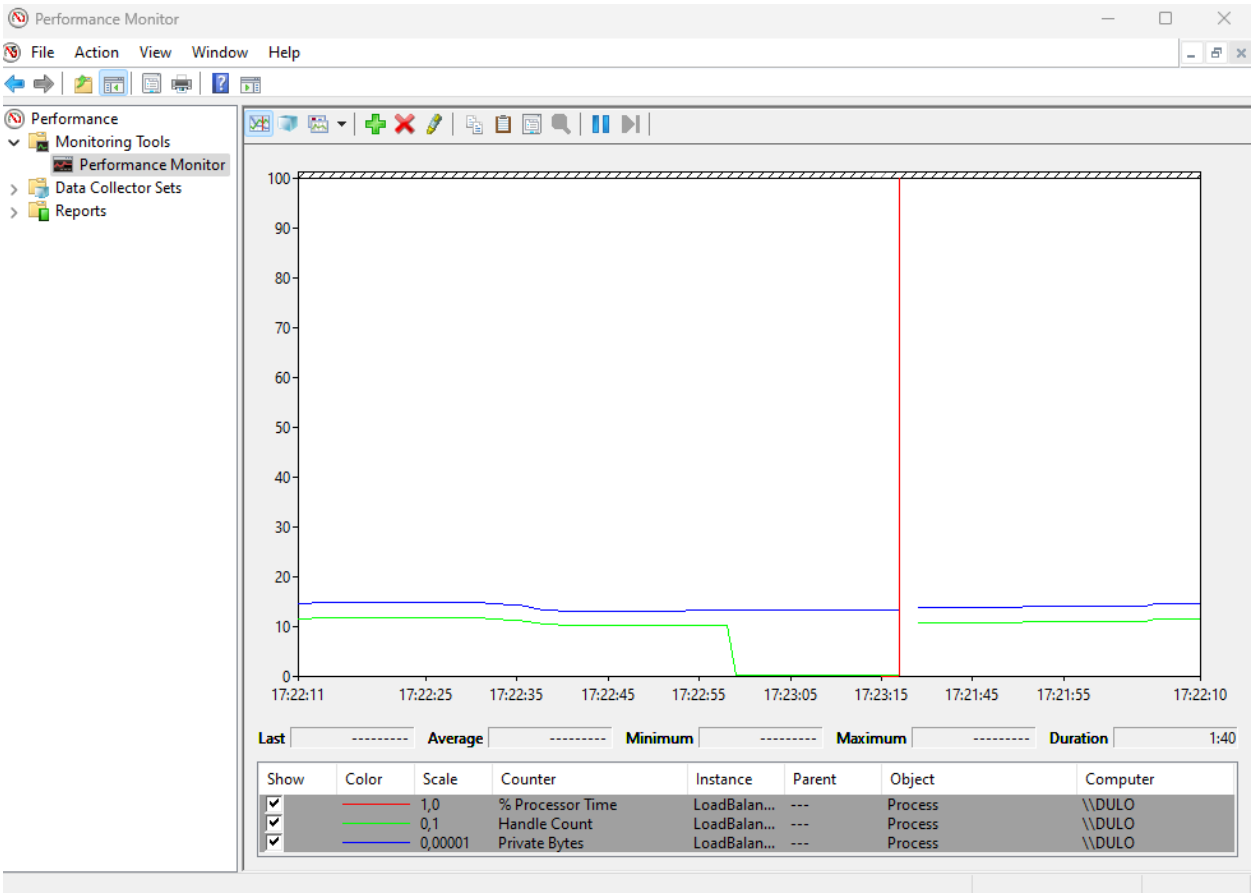
Dodavanje HashTable



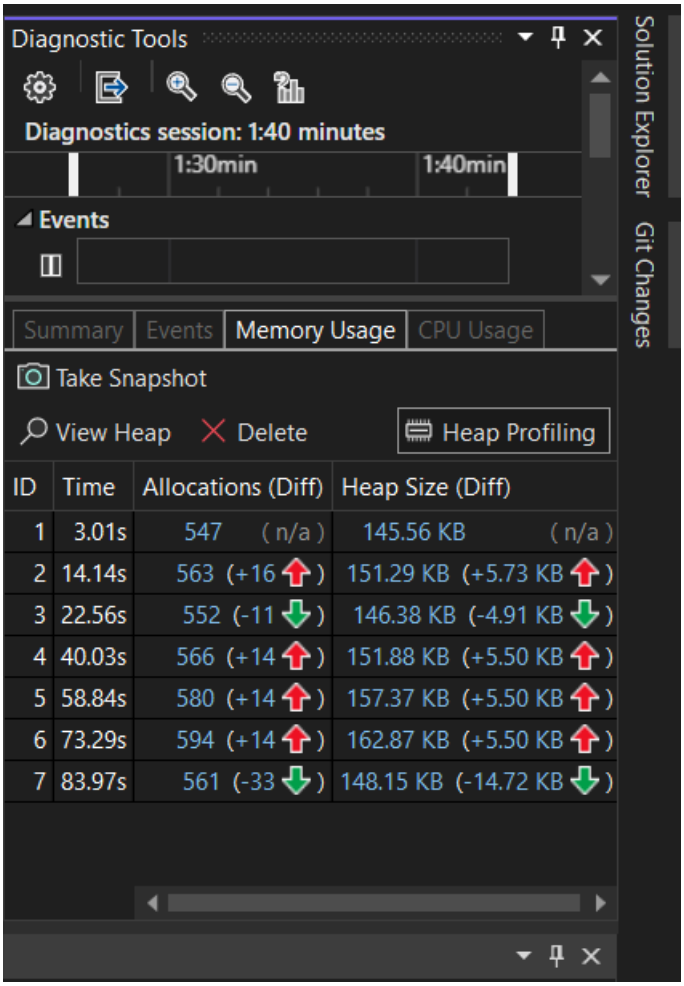
Oslobađanje memorije aplikacije



Dodavanje novih klijenata aplikacije koji šalju poruku svake sekunde

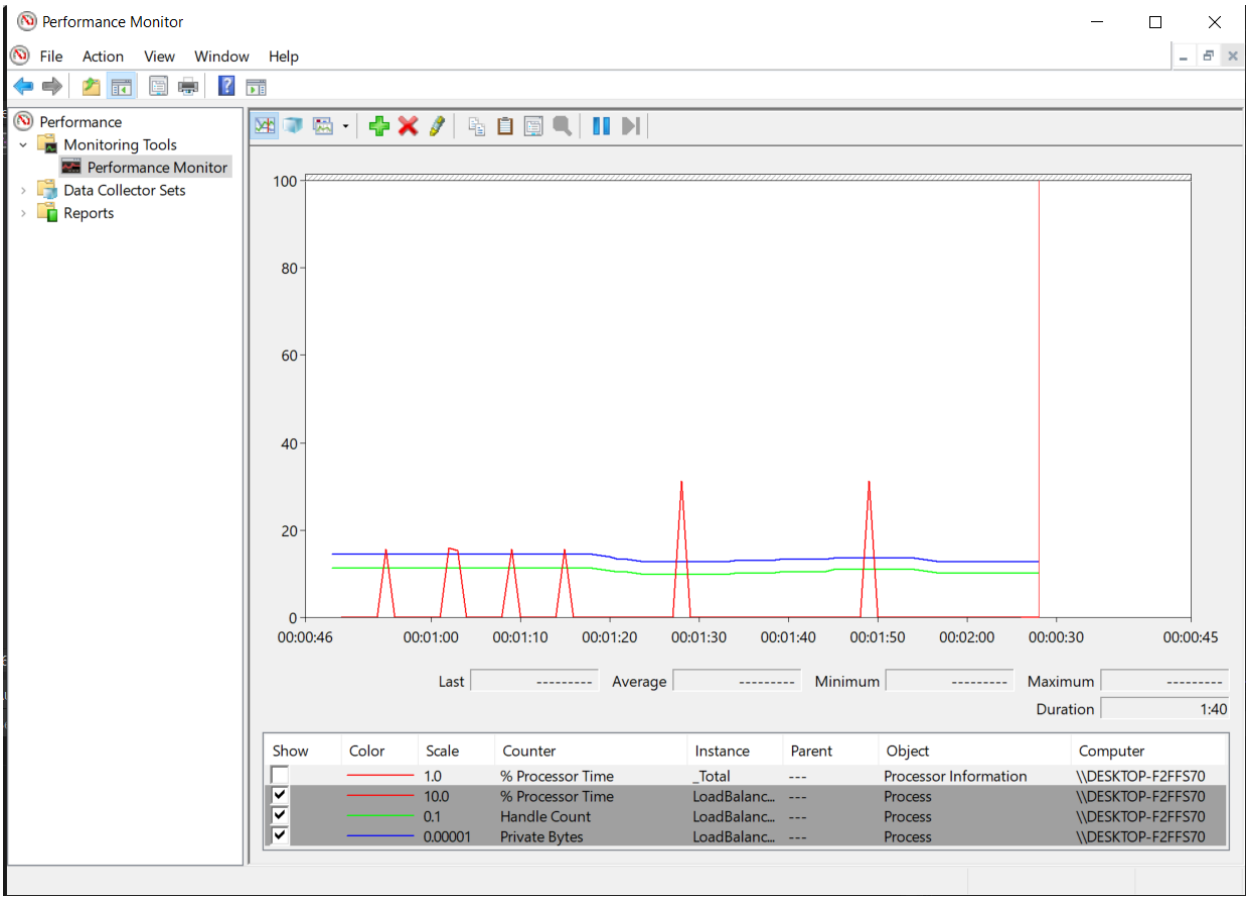


Pokretanje 5 klijenata i zatim gašenje aplikacije

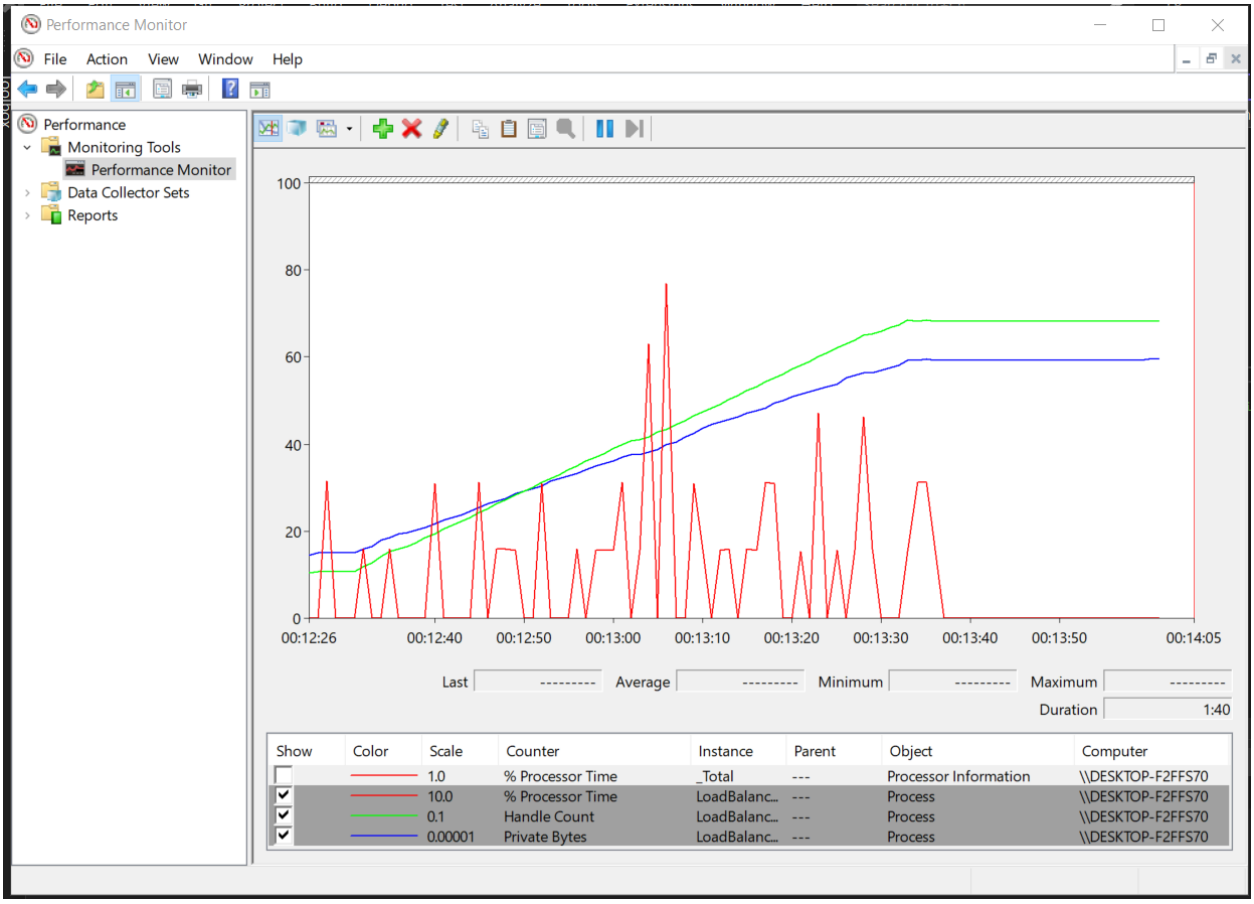


Uključivanje i gašenje klijenata

Stress test



5 klijenata koji šalju poruke na 500ms, zatim su ugašeni i upaljeno je još 4 klijenta



5 klijenata koji šalju poruke na svakih 200ms

Zaključak

Pregledom rezultata urađenih testova nad aplikacijom možemo doći do zaključka da postoji povremeno curenje memorije. Curenje memorije je najizraženije kod Worker komponente. Pri gašenju radnika se ne oslobodi ista količina memorije koja je zauzeta pri kreiranju istog.

Po rezultatima stress testova možemo zaključiti da se povećanjem broja klijenata i brojem poruka koje Load Balancer jako brzo puni red sa poruka, te je potrebno pokrenuti sve više i više radnika, što znatno utiče na opterećenje memorije i procesora računara.

Potencijalna unapređenja

Prema drugom stress testu, u kome je pokrenuto 5 klijenata koji šalju poruke na svake 0.2s možemo videti da se veoma brzo povećava korišćenje procesorske memorije. Potrebno je ograničiti broj radnika, odnosno ograničiti klijenta na broj poruka koji može da pošalje u određenom vremenskom intervalu.

Trebalo bi prekontrolisati Worker komponentu i proveriti gde dolazi do curenje memorije pri gašenju istih.