УНИВЕРЗИТЕТ У НОВОМ САДУ
ФАКУЛТЕТ ТЕХНИЧКИХ НАУКА У НОВОМ САДУ

Данијел Јовановић PR55-2020
Вук Огњановић PR51-2020

ИНДУСТРИЈСКО КОМУНИКАЦИОНИ ПРОТОКОЛИ У ИНФРАСТРУКТУРНИМ СИСТЕМИМА

ПРОЈЕКАТ

- Примењено софтверско инжењерство (ОАС) –

Нови Сад, 2023.

САДРЖАЈ

ОПИС РЕШАВАНОГ ПРОБЛЕМА И ЦИЉЕВИ

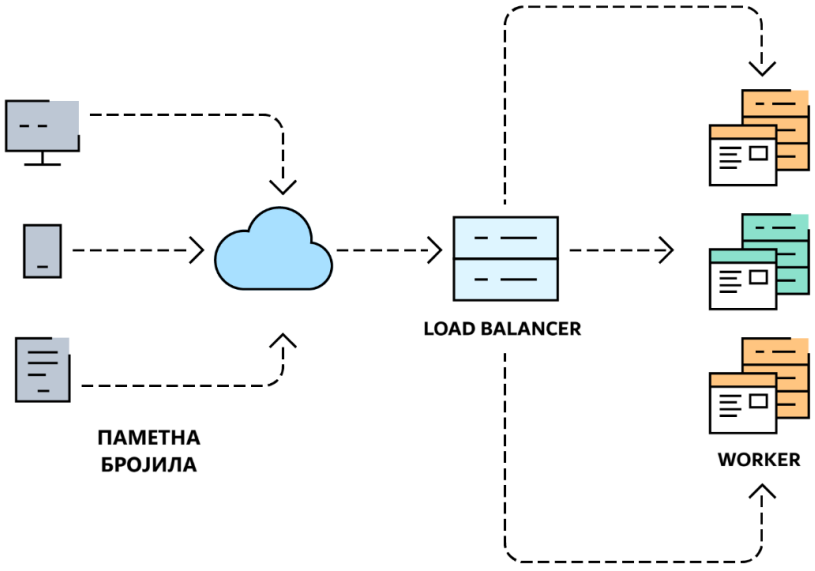
Проблем свакодневнице који се годинама уназад провлачи је све већи и већи обим података које треба пренети и обрадити. До пре пар година, свако домаћинство је имало по једно паметно бројило (узимајући фактор да су паметна бројила у својим првим фазама дистрибуције за широке масе била скупља у односу на конвенциона бројила). Временом, цена паметних бројила као и њихова заступљеност и евидентна заступљеност донела је многе предности али и увела нове изазове.

Новитет који је стигао са паметним бројилима је тај да је количина података коју бројила достављају велика. Све те податке потребно је да неко обради (нпр. сервис за обраду података). Обрада података није нимало наиван процес. У процесу обраде кључно је што брже и ефикасније добијене податке обрадити и прећи на обраду нове серије података. Проблем настаје када се број клијената односно паметних бројила која шаљу податке повећа. Тада традиционалне архиктетуре и методологије нису нимало ефикасне и могу довести до пада сервиса за обраду података.

Како би се изашло у коштац са проблемом преоптерећења и потенцијалног пада сервиса и загушења комуникационих линкова потребно је било оптерећење распоредити што равномерније, а саме податке обрадити што ефективније. У том контексту уведен је и имплементиран ***Load Balancing*** алгоритам расподеле оптерећења.

Претпоставка је да систем располаже са **N** процеса који представљају Worker-е W_R (процеси који обрађују податке који им се проследе). Сваки W_R у једном тренутку може вршити обраду једне серије података. Како постоји **N** таквих процеса у једном тренутку идеално се може обрадити **N** серија података. Али, како W_R може само обрађивати податке, ту на сцену ступа нови процес **Load Balancer** - L_B.

На основу слике 1, L_B је процес који има приступ свим подацима који су пристигли од стране клијената и има увид у тренутно заузеће процеца за обраду података W_R. На основу тих података његов задатак је једноставан: **заузети што већи број процеса и распоредити заузеће на начин како би се обрадила што већа количина података и смањило оптерећење у јединици времена.**



Слика 1: Load Balancing алгоритам

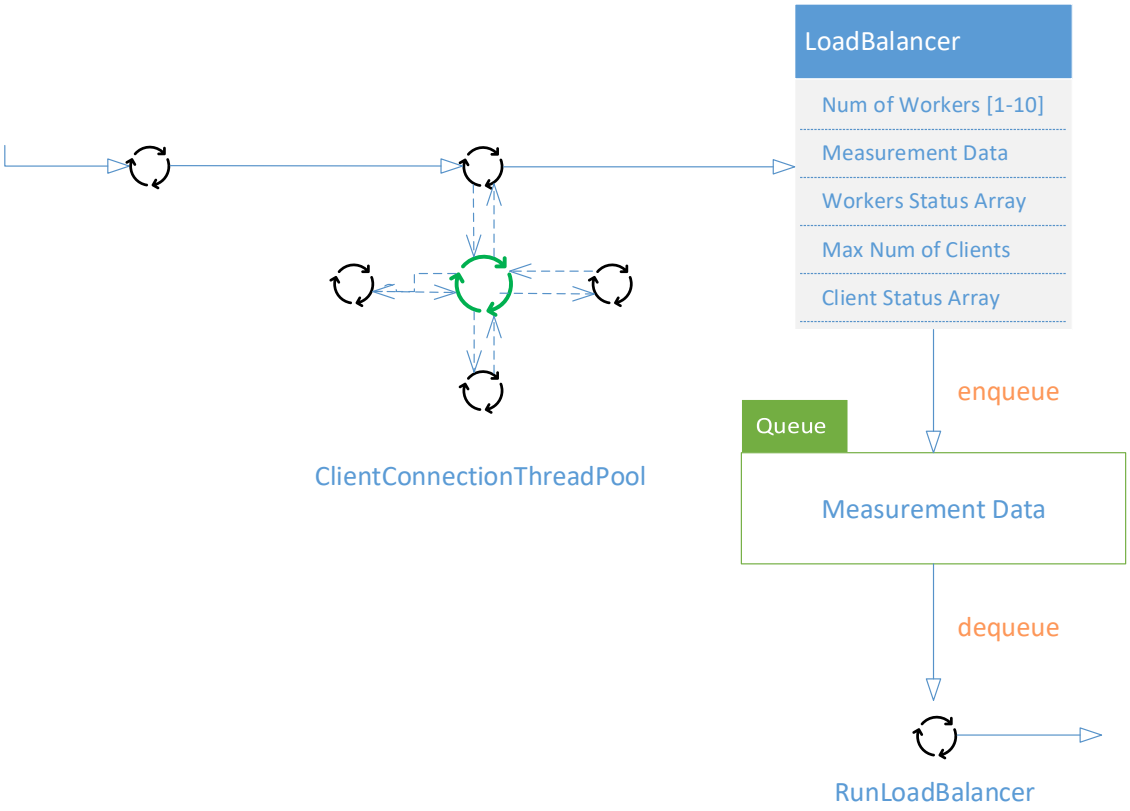
ОПИС РЕШЕЊА ПРОБЛЕМА

ДИЗАЈН СИСТЕМА

Први део система, описан на слици 2, се састоји од три главне компоненте: ***Client Handler*** за управљање конекцијама клијената усмерених ка сервису, ***Thread Pool*** за рад са клијентима односно смештање пристиглих мерења као и ***Load Balancer***-а који управља нитима за обраду мерења. Ове компоненте међусобно сарађују како би на што је могуће оптималнији начин обрадиле захтеве и истовремено распоредили задатке коришћењем ***Load Balancing*** алгоритма што је равномерније могуће међу одређеним бројем нити за обраду података.

Client Handler

Client Handler је одговор за управљање долазних веза клијената. Ослушкује захтеве клијената, успоставља везе и иницира поступке руковања комуникацијом. Када веза буде успостављена, ***Client Handler*** препушта повезане клијенте ***Thread Pool***-у на даљу обраду.



Слика 2: Дијаграм дизајна система I

Client Thread Pool

Client Thread Pool динамички додељује нити како би ефикасно обрадио истовремене захтеве клијената. Са конфигурабилним максималним бројем нити, ова компонента обезбеђује оптимално коришћење ресурса. ***Client Status Array*** прати статус обраде сваког клијента, док подаци о мерењу бележе садрже клијентске податке у којима се налази датум мерења и вредност мерења.

Load Balancer

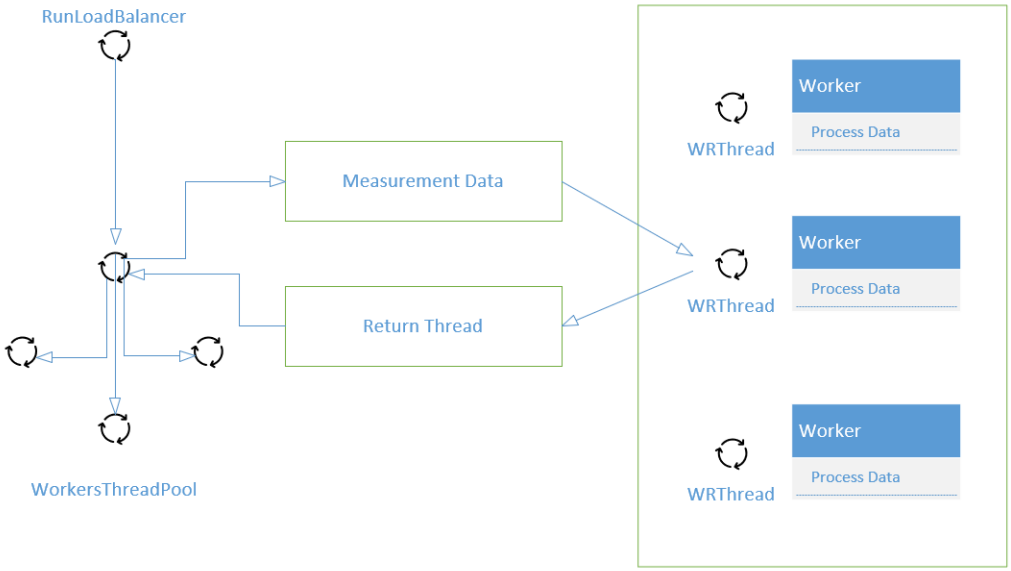
У средишту система налази се ***Load Balancer***, који преузима одговорност за дистрибуцију захтева клијената међу доступним нитима за обраду података. Конфигурабилне особине попут броја радника и максималног броја клијената омогућавају прилагодљивост у подешавању понашања система.

Када нит ***RunLoadBalancer*** преузме податке из реда, почиње слање података ка компонентама ***Worker*** (слика 3).

Load Balancer периодично провера да ли се у ***Queue*** налази барем један пристигли податак о мерењу и ако се налази провера да ли у ***Thread Pool***-у има слободних нити. Ако има слободних нити, иницира покретање процеса за обраду и наставља даљу периодичну проверу реда.

WorkersThreadPool

WorkersThreadPool је компонента одговорна за управљање ***Thread Pool***-ом за обраду података. Он омогућава конкурентно обављање више задатака.



Слика 3: Дијаграм дизајна система II

Worker

Када се нит добије из ***Workers Thread Pool***-а, нит ***RunLoadBalancer*** прослеђује податке клијента ***Worker*** компоненти на обраду. ***Worker*** компонента је дизајнирана да обрађује податке од клијената, извршавајући обраду података.

Ток обраде

Worker компонента обрађује додељени задатак користећи добијену нит. По завршетку задатка, ***Worker*** враћа нит у ***WorkersThreadPool***. Овај приступ обезбеђује ефикасно управљање нитима, избегавајући исцрпљивање ресурса и оптимизацију перформанси.

Сигнализација Load Balancer-у

Истовремено, по завршетку задатка, **Worker** компонента сигнализира ***Load Balancer***-у да је задатак завршен. Овакав механизам сигнализације одржава ажурирано стање доступности ***Worker***-а и омогућава ***Load Balancer***-у да доноси одлуке о дистрибуцији задатака на основу стања радника у реалном времену.

ИМПЛЕМЕНТАЦИЈА СИСТЕМА

Серверска Апликација

Сервер апликација игра главну улогу у успешном повезивању са клијентима и управљању њиховим захтевима. Користећи ***socket***-е, постигнута је ефикасност у прихватању и обради више клијентских веза. ***Thread Pool*** за клијенте омогућава ефикасно управљање и праћење клијентских веза.

Load Balancer

Load Balancer је срж система, а имплементација користи основни ***Round-Robin*** приступ за дистрибуцију задатка између доступних нити за обраду података. **T*hread Pool*** је одговоран за оптимално коришћење ресурса, а ***Load Balancer*** за расподелу оптерећења, креацију и деактивацију нити као и ефикасним управљањем корисничким захтевима за прекид рада свих сервиса.

Queue

За сигурну комуникацију између нити за обраду података и ***Load Balancer***-а, имплементиран је ***Queue*** чије су операције атомичне природе односно у току смештања података (***Enqueue***) није могуће избацити податке из реда (***Dequeue***) и обрнуто.

Handlers

Имплементирани су ***handler***-и како би се подаци мерења добијени од клијената сачували у ***Queue*** и нити за обраду података. ***Handler*** клијената управља клијентским везама и процесуира податке мерења и смешта их у ***Queue***, док ***handler*** нити за обраду података процесуира податке добијене од ***Load Balancer***-а. Ови ***handler***-и омогућавају управљање нитима као и ефикасну обраду корисничког захтева за прекид рада свих сервиса.

Client

Клијент се путем ***TCP*** конекције повезује на сервер и шаље податке док је конекција активна. Подаци садрже датум у опсегу од 2020-2023. године и измерену вредност. Подаци који се шаљу на обраду се генеришу насумично.

СТРУКТУРЕ ПОДАТАКА

Queue

Queue се користи за олакшавање комуникације између ***Load Balancer*** и нити за обраду података. Ова структура података обезбеђује безбедан и ефикасан пренос података у конкурентом окружењу.

```
typedef struct Queue {
    Node* head;              // Pointer to the head of the queue
    Node* tail;              // Pointer to the tail of the queue
    CRITICAL_SECTION lock;   // Critical section for synchronization
    int size;                // Size of the queue
    int shutdown;            // Shutdown flag to stop processing data
} Queue;
```

Листинг 1: Структура података Queue

Избор ***Queue*** структуре података произилази из потребе за синхронизованим приступом подацима у конкурентном окружењу. Критична секција обезбеђује да операције попут додавања и избацивања буду атомичне, чиме се спречава неконзистентност података.

```
void InitializeQueue(Queue* queue);
void Enqueue(Queue* queue, MeasurementData* newData);
struct MeasurementData* Dequeue(Queue* queue);
void ClearQueue(Queue* queue);
void DestroyQueue(Queue* queue);
int QueueSize(Queue* queue, int dataProcessed);
int ShutDown(Queue* queue);
void SetShutDown(Queue* queue);
```

Листинг 2: Методе за рад са структуром података Queue

Структура клијентских података за обраду

Клијент користи структуру ***MeasurementData*** како би енкапсулирао генерисане вредности мерења и датуме. Структура се динамички алоцира за сваку итерацију и након тога се ослобађа, ефикасно управљајући меморијом.

```
typedef struct MeasurementData {
    char date[11];                  // Date in DD.MM.YYYY format
    unsigned int measurementValue;  // Measured value
} MeasurementData;
```

Листинг 3: структура за смештање измерених података

ТЕСТИРАЊЕ И РЕЗУЛТАТИ ТЕСТОВА

Тестирање решења је есенцијално како би се провериле структуре података, брзина рада решења, заузеће меморије. У ту сврху посебно су написана два теста:

- **Провера структуре података (Queue) у граничним случајевима**
- **Провера пропусности система у проактивном режиму рада**

ПРОВЕРА СТРУКТУРЕ ПОДАТАКА У ГРАНИЧНИМ СЛУЧАЈЕВИМА

Процес тестирања укључује неколико корака који су неизоставни у провери рада програмског решења. Тест иницијално креира ***Queue*** а затим га алоцира простор у радној меморији и поставља ***Queue*** у задато почетно стање (ред је празан). Намена ***Queue*** је да касније у току теста, подаци буду чувани до обраде.

Queue је тренутно празан, те је сада потребно започети процес алокације меморије за податке о мерењима који ће бити смештени у ред. Тест почиње да алоцира (резервише) делове меморије за податке а затим се редом стављају у " ***Queue***. У случају грешке, као што је недостатак меморије за алокацију, ***Queue*** ће бити исражњен а сви претходни заузети меморијски сегменти ослобођени.

Битно је напоменути да у току процеса алокацирања меморије, програм води рачуна колико је меморије заузето и колико је података креирано у измереној количини меморије.

```
-------------------------------------------------------------------------------------------
[Option Runner]: Running Queue Stress Test...
-------------------------------------------------------------------------------------------

[Queue Allocator]:   Memory allocated after   10000 allocations:       160000 bytes
[Queue Allocator]:   Memory allocated after   20000 allocations:       320000 bytes
[Queue Allocator]:   Memory allocated after   30000 allocations:       480000 bytes
[Queue Allocator]:   Memory allocated after   40000 allocations:       640000 bytes
[Queue Allocator]:   Memory allocated after   50000 allocations:       800000 bytes
[Queue Allocator]:   Memory allocated after   60000 allocations:       960000 bytes
[Queue Allocator]:   Memory allocated after   70000 allocations:      1120000 bytes
[Queue Allocator]:   Memory allocated after   80000 allocations:      1280000 bytes
[Queue Allocator]:   Memory allocated after   90000 allocations:      1440000 bytes
[Queue Allocator]:   Memory allocated after  100000 allocations:      1600000 bytes
[Queue Deallocator]: Memory freed up after   10000  deallocations:     160000 bytes
[Queue Deallocator]: Memory freed up after   20000  deallocations:     320000 bytes
[Queue Deallocator]: Memory freed up after   30000  deallocations:     480000 bytes
[Queue Deallocator]: Memory freed up after   40000  deallocations:     640000 bytes
[Queue Deallocator]: Memory freed up after   50000  deallocations:     800000 bytes
[Queue Deallocator]: Memory freed up after   60000  deallocations:     960000 bytes
[Queue Deallocator]: Memory freed up after   70000  deallocations:    1120000 bytes
[Queue Deallocator]: Memory freed up after   80000  deallocations:    1280000 bytes
[Queue Deallocator]: Memory freed up after   90000  deallocations:    1440000 bytes
[Queue Deallocator]: Memory freed up after  100000  deallocations:    1600000 bytes

============================ [Stress Test Checker]: Test Passed ============================

-------------------------------------------------------------------------------------------
[Option Runner]: Running Queue Stress Test Completed
-------------------------------------------------------------------------------------------
```

Листинг 4: Преглед резултата теста за *Queue*

Након успешне алокакције задатог броја елемената, покреће се ослобађање заузете меморије. Као и у процесу алоцирања, тест прати статистику колико је меморије ослобођено и колико података је избрисано из ***Queue***. На основу листинга 4 видљиво је да је тест успешно прошао јер се количина меморије која је била заузета односно алоцирана на крају поново постала доступна оперативном систему.

Важност теста се огледа у томе јер проверава како један део програма управља меморијом, што је кључно за одржавање стабилности програма и спречавање прекомерног коришћења меморије, што на дуже стазе може довести до проблема у раду програма.

ПРОВЕРА ПРОПУСНОСТИ СИСТЕМА У ПРОАКТИВНОМ РЕЖИМУ РАДА

Суштина теста је процена и тестирање пропусности система у проактивном режиму рада. Програм се тестира како би се проценила брзина обраде података и ефикасност система у таквом окружењу где је проток података у великим количинама.

Први корак укључује постављање основних параметара и иницијализацију структуре података која служи за управљање подацима у току теста. Структура података која се користи је ***Queue*** и служи за чување и манипулацију подацима у обради.

У циљу лакше провере пропусности, програм генерише узорке података о мерењима који се затим чувају у ***Queue*** за даљу обраду. Такође, програм врши конфигурацију система и приказује информације о тренутном задатим поставкама (број нити, временски период освежавања статистике...).

Након иницијализације ***Queue*** и чувања генерисаних података, програм креира нити (***Workers***) које обрађују генерисане податке. Заузима се меморија за обраду, постављају се на одговарајуће вредности сви неопходни параметари за рад, и иницира се почетак процеса обраде у систему.

```
-------------------------------------------------------------------------------------
[Option Runner]: Running Bandwidth Stress Test...
-------------------------------------------------------------------------------------

[Configuration Service]: Configuration has been applied successfully

[Test Resource Generator]: Generating test samples of measurement data...
[Test Resource Generator]: Generating test data completed. 100 samples has been added!
[Intelligent Background Service Runner]: Initializating threads...
[Intelligent Background Service Runner]: Threads are up!
[Interaction Service]: Press 'Q' or 'q' to cancel bandwidth test

[Bandwidth Statistics]: Currently in queue: 65 samples
[Bandwidth Statistics]: Bandwidth in the last 15 seconds: 80%
[Bandwidth Statistics]: Currently in queue: 36 samples
[Bandwidth Statistics]: Bandwidth in the last 15 seconds: 80%

[Intelligent Resource Manager]: Gracefully shutting down Load Balancer service...
[Intelligent Resource Manager]: Gracefully shutting down Bandwidth Statistics service...

========================= [Bandwidth Test Checker]: Test Passed =========================

-------------------------------------------------------------------------------------
[Option Runner]: Running Bandwidth Stress Test Completed
-------------------------------------------------------------------------------------
```

Листинг 5: Преглед резултата теста пропусности система

Увидом у листинг 5, систем је успешно и ефикасно одреаговао на гранични случај протока и обраде података и равномерно распоредио оптерећење на доступне нити.

Битно је систем тестирати за променљиве параметре и тиме осигурати да се пропусност у што већем броју тестних случаја понаша конзистентно и предвидиво. За дато програмско решење тест је покретан више пута, забележено је више мерења а затим средња вредност приказана на графику 1.

График 1: Студија анализе пропусности система

Наизглед, резултати не изгледају коректно, али заправо, резултати приказују оптимално стање система. Лако је уочљиво да се систем у једном делу понаша симетрично-засићеним (иако се број нити за обраду повећале, резултати обраде су остали приближно идентични). Градуалним повећањем нити за обраду, засићење криве се превазилази и перформансе постају значајно боље.

Посматрајући случај где је употребљен максималан број нити за обраду података доноси се закључак: **пропусност система је реално-оптимална, оптерећење равномерно подељено на све нити за обраду а сам проток података адекватан**.

Систем је како се број нити за обраду повећавао све мање и мање бивао оптерећен, док је са само једном нити за обраду систем већину времена био максимално оптерећен и обрађивао мању количину података.

ПОТЕНЦИЈАЛНА УНАПРЕЂЕЊА

Одговарајуће унапређење тестирања структура података укључује разматрање различитих граничних случајева и боље анализе у тест процедури. Побољшања у овом подручју би укључивала шире тестирања различитих услова, усмерених на разумевање и реакцију на критичне ситуације. Ово може обухватити изучавање различитих нивоа оптерећења и реакције система у непредвиђеним или екстремним околностима.

Унапређење у управљању меморијом представља један од кључних аспеката. Осмислити и користити ефикасније методе за алокацију и деалокацију меморије, у складу са захтевима система, може значајно побољшати перформансе и оптимизовати коришћење ресурса.

Додатно, дефинисање подешавања која би омогућила контролу и праћење брзине извршавања теста би представљало значајан корак у представљању статистике и активног надгледања перформанси. Ово би допринело бољем разумевању резултата теста и идентификацији потенцијалних слабих тачака система.

Истраживање и употреба аутоматизованих тестова и алата за брже и ефикасније покретање тест процедура могу значајно уштедети време и убрзати развој. Овакав вид приступа омогућава брже откривање и решавање проблематичних делова кода.

Истраживање и имплементација метода за анализу процеса у меморији, које би омогућиле дубље разумевање ресурса и њиховог коришћења у условима великог оптерећења, такође представља значајан корак у напретку тестирања и управљању ресурсима.