Универзитет у Београду

Електротехнички факултет



Формална и симулациона анализа

алгоритама за дисеминацију података

у блокчејн мрежи

Дипломски рад

|  |  |
| --- | --- |
| Ментор: | Кандидат: |
| проф. др Јелица Протић | Лука Милетић 2014/0036 |

Београд, септембар 2018.

Садржај

[Садржај 2](#_Toc524604483)

[1. Увод 3](#_Toc524604484)

[2. Дисеминација података у блокчејн мрежи 4](#_Toc524604485)

[2.1. О Тендерминт технологији 4](#_Toc524604486)

[2.2. Дефиниције 5](#_Toc524604487)

[2.2.1. Реплицирање детерминистичких коначних аутомата 5](#_Toc524604488)

[2.2.2. Мемпул компонента 9](#_Toc524604489)

[2.2.3. Модел система 11](#_Toc524604490)

[2.2.4. Дисеминација клијентске трансакције и постизање консензуса 13](#_Toc524604491)

[3. Формална анализа алгоритама 17](#_Toc524604492)

[3.1. Тренутно решење 17](#_Toc524604493)

[3.1.1. Објашњење алгоритма 17](#_Toc524604494)

[3.1.2. Анализа перформанси 19](#_Toc524604495)

[3.1.3. Дискусија решења 19](#_Toc524604496)

[3.2. Алгоритам за дисеминацију заснован на клијенту 20](#_Toc524604497)

[3.2.1. Објашњење алгоритма 20](#_Toc524604498)

[3.2.2. Анализа перформанси 22](#_Toc524604499)

[3.2.3. Дискусија решења 22](#_Toc524604500)

[3.3. Кластеровање стабала 22](#_Toc524604501)

[3.3.1. Објашњење алгоритма 23](#_Toc524604502)

[3.3.2. Анализа перформанси 32](#_Toc524604503)

[3.3.3. Дискусија решења 33](#_Toc524604504)

[3.4. PPP HEAP PSS 33](#_Toc524604505)

[3.4.1. Објашњење алгоритма 34](#_Toc524604506)

[3.4.2. Анализа перформанси 48](#_Toc524604507)

[3.4.3. Дискусија решења 49](#_Toc524604508)

[4. Симулациона анализа алгоритама 50](#_Toc524604509)

[4.1. Поставка симулатора 50](#_Toc524604510)

[4.1.1. Сервери у симулацији 51](#_Toc524604511)

[4.1.2. Клијент у симулацији 54](#_Toc524604512)

[4.1.3. Одређивање метрика у симулацији 55](#_Toc524604513)

[4.2. Резултати симулације и анализа добијених резултата 59](#_Toc524604514)

[5. Закључак 67](#_Toc524604515)

[Литература 68](#_Toc524604516)

[Списак скраћеница 70](#_Toc524604517)

[Списак слика 71](#_Toc524604518)

[Списак табела 72](#_Toc524604519)

1. Увод

Овај рад представиће формалну и симулациону анализу алгоритама за дисеминацију података на примеру Тендерминт технологије [1]. Тендерминт представља врсту блокчејн шеме. Блокчејн шеме обезбеђују децентрализацију и непорецивост. Анализа алгоритама уско је повезана са Мемпул (енг. *Mempool*) компонентом Тендерминта, која је одговорна за дисеминацију података у мрежи. Она је тренутно имплементирана тако да гарантује одређени исход, али производи велику редундантност. Ослања се на P2P (*Peer to peer*) комуникацију између сервера у дистрибуираној мрежи. У овом документу биће приказана формална, а затим и симулациона анализа неколико алгоритама са циљем да се реши проблем редундантности који постоји у тренутној имплементацији Мемпула у Тендерминту.

Уопштено гледано, постоје три различита приступа када су у питању алгоритми за дисеминацију. Први је заснован на томе да главну улогу у дисеминацији има клијент. Други је базиран на томе да се врши класификација сервера у мрежи у структуриране групе. Трећи се ослања на потпуно динамичко повезивање сервера, чиме се формира насумични граф мреже. Алгоритми анализирани у овом раду се заснивају на та три приступа.

У поглављу 2. биће објашњено шта је Тендерминт, са освртањем на принципе који су битни за овај рад. Након тога, биће дате дефиниције појмова који су релевантни за боље разумевање проблема који постоји, као и приложених решења. У овом поглављу биће дефинисан и модел система на коме се заснивају приложена решења.

У поглављу 3. овог документа биће представљена формална анализа тренутног алгоритма који се користи за дисеминацију у Тендерминту, заједно са још три предложена решења. Свако решење биће праћено анализом перформанси, као и дискусијом валидности тог решења у мрежи у којој може да дође до отказа. У поглављу 4. биће детаљно објашњена симулациона анализа. Симулациона анализа има за циљ да прикаже недостатке тренутног решења, али и предности једног од алгоритама предложених у поглављу 3. Показаће се да је предложени алгоритам далеко погоднији за употребу, те да стога може имати и реалну примену у оквиру Тендерминта. Рад ће бити закључен у поглављу 5. У закључку биће дат резиме свега што је урађено, истакнути доприноси овог рада, и наведене напомене за даља истраживања у вези алгоритама за дисеминацију.

1. Дисеминација података у блокчејн мрежи
   1. О Тендерминт технологији

Блокчејн представља модел складиштења података заснован на децентрализацији и непорецивости. У овом моделу, подаци су дистрибуирани на већем броју сервера, чиме се елиминише SPOF (*Single Point Of Failure*) проблем. Додатно, подаци су потпуно отпорни на нежељене модификације од стране трећег лица.

Блокчејн је привукао доста пажње првенствено због употребе унутар Биткоин (енг. *Bitcoin*) система трансакција [2]. Биткоин је криптовалута која се ослања на дистрибуирање информација о клијентским трансакцијама у већи број блокчејн ланаца. У Биткоину користе се различити механизми криптографије да би се обезбедила тајност. Суштински гледано, блокчејн је уланчана листа блокова који садрже клијентске трансакције [3]. Везе између блокова моделиране су користећи криптографске хеш функције. Из разлога што је таква уланчана листа реплицирана на великом броју сервера у дистрибуираној мрежи, мора постојати јасно дефинисан протокол на основу ког сви сервери једногласно могу да донесу консензус, и одлуче шта ће бити следећи блок трансакција у блокчејну.

Евидентно је да је блокчејн добио на популарности захваљујући Биткоину. Међутим, ова идеја може да се примени у много генералнијем контексту. Блокчејн не мора да се користи само за финансијске трансакције, већ може бити коришћен и за било какве трансакције које садрже неке инструкције [4].

Идеја о томе да трансакција може бити било шта користи се и у Тендерминту. Детаљна спецификација Тендерминта може се наћи у [16]. Тендерминт је блокчејн отпоран на отказе [5]. Протокол за консензус који се користи у Тендерминту, који не захтева „рударење“ као консензус протокол који се користи у Биткоину, служи да успостави јасан редослед трансакција у блокчејну у присуству малициозних трећих лица (*Byzantine generals problem* [6]).

Протокол за консензус представља једну од две основне компоненте Тендерминта. У овом раду, консензус алгоритам биће већином посматран као црна кутија, узевши у обзир чињеницу да сви детаљи његове имплементације нису битни за анализу алгоритама за дисеминацију. Само они елементи консензус алгоритма који су уско повезани са алгоритмом за дисеминацију биће објашњени.

Друга основна компонента Тендерминта јесте Мемпул. Мемпул се користи за памћење клијентских трансакција које још увек нису део блокчејна, као и за дисеминацију тих трансакција ка осталим серверима. Може се рећи да се у њему чувају трансакције за које још увек није постигнут консензус. Мемпул компонента представља улаз консензус алгоритма. Излаз консензус алгоритма јесте блок трансакција који ће бити додат у блокчејн.

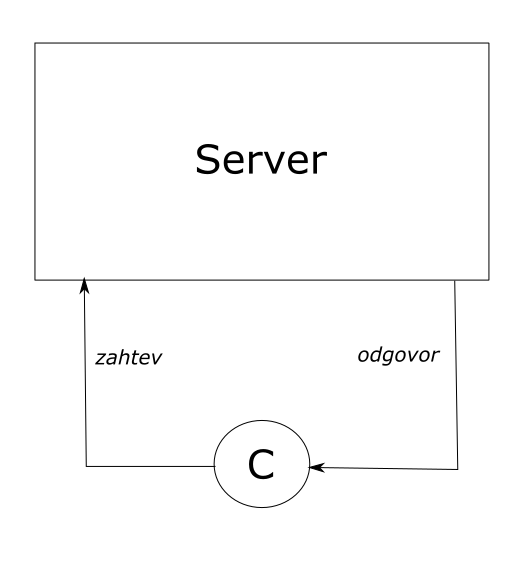
Дакле, Тендерминт је представљен дистрибуираном мрежом сервера где сваки сервер има своју реплику блокчејна. Поред тога што се у блокчејну чувају информације о трансакцијама клијената, те трансакције такође бивају извршене у тренутку када се додају у блокчејн. Из перспективе клијента, битно је да се трансакција дода у блокчејн и да буде извршена. Тендерминт је настао као приватна блокчејн мрежа, али тренутно се тежи томе да та мрежа постане јавна. Јавни блокчејн Тендерминта зове се *Cosmos*. Информације о њему могу се наћи у [17].

* 1. Дефиниције

У овом потпоглављу, пре свега дате су дефиниције појмова који су неопходни за анализу алгоритама за дисеминацију у Мемпулу Тендерминта. Затим је дефинисан модел система, довољно једноставан да буде лако разумљив, а с друге стране довољно комплексан да се на њему могу вршити анализе одговарајућих алгоритама.

* + 1. Реплицирање детерминистичких коначних аутомата

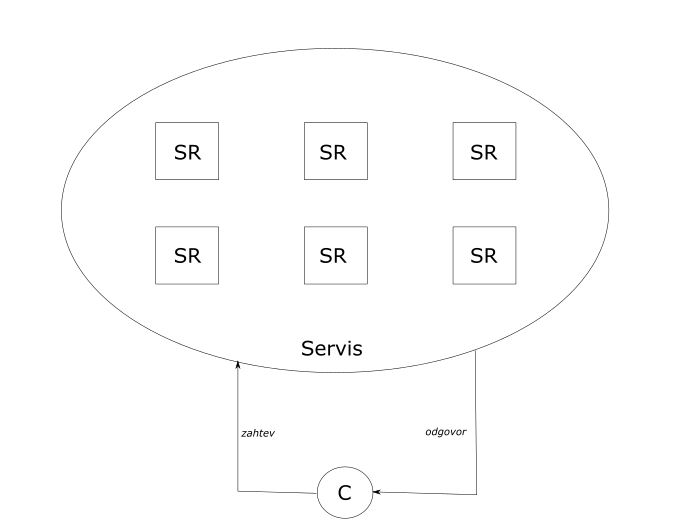
Узмимо у обзир стандардну клијент-сервер архитектуру, приказану на слици 2.2.1. Сервер обезбеђује сервис за клијента. Да би клијент искористио сервис, мора да пошаље захтев, а затим да чека на одговор.



Слика 2.2.1 – Стандардна клијент-сервер архитектура

* ***C*** – клијент

Премда је ово најједноставнији приступ у имплементацији клијент-сервер архитектуре, његова главна мана је у томе што постоји само један сервер. У случају да дође до отказа тог сервера, цео сервис постаје недоступан клијенту. Овакав сервер може се сматрати SPOF делом система. Из тог разлога, усвојићемо напреднији и поузданији приступ у имплементирању клијент-сервер архитектуре, приказан на слици 2.2.2. Овај приступ заснован је на реплицирању детерминистичких коначних аутомата - SMR (*State Machine Replication*).



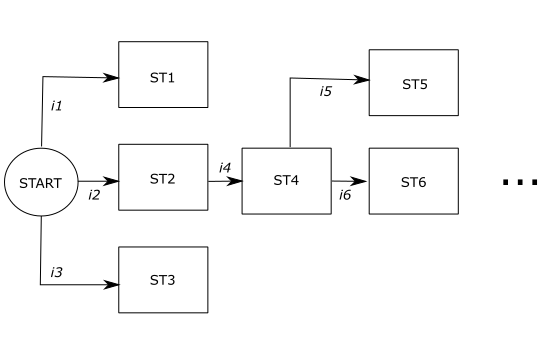
Слика 2.2.2 – Реплицирање сервера

* ***SR*** – Реплика сервера (енг. *Server replica*)
* ***C*** – Клијент

На слици 2.2.2 може се приметити да је клијенту и даље доступан само један сервис. С друге стране, постоји много реплика сервера који пружају тај сервис. Све оне су потпуно транспарентне за клијента. Из тог разлога, мора се дефинисати протокол који координира интеракцију клијента са репликама сервера. Дискусија оваквог протокола превазилази оквире овог рада.

Размотримо следећу ситуацију - Клијент се конектовао на реплику сервера и комуницирао са њом да би слао своје захтеве и добио одговоре. Временом, та реплика сервера постала је неисправна (нпр. отказала је и постала недоступна). Малопре поменути протокол морао би да повеже клијента са неком другом репликом сервера, која би наставила да враћа одговоре клијенту, а да притом то све буде транспарентно за клијента. Питање које се поставља је – Како ће нова реплика сервера знати шта је стара радила? Проблем се усложњава због чињенице да све реплике сервера могу потенцијално да опслужују и различите клијенте.

Решење које SMR приступ дефинише је – Свака реплика сервера биће имплементирана као детерминистички коначни аутомат, који се састоји од стања и прелаза. Кад год клијент пошаље захтев на извршавање, захтев ће бити извршен на свим репликама сервера, тако да ће све оне колективно моћи да пређу у исто следеће стање. Овај приступ такође претпоставља да су прелази између стања на свим репликама сервера апсолутно исти. Као последица тога, када се догоди ситуација објашњена малопре, како је нова реплика сервера у истом стању у ком је била и стара, моћи ће лако да одговори клијенту на исправан начин.



Слика 2.2.3 – Детерминистички коначни аутомат унутар реплике сервера

* ***ST*** – Стање
* ***I*** – Улаз (енг. *Input*)

Пример детерминистичког коначног аутомата унутар реплике сервера дат је на слици 2.2.3. Важно је приметити да аутомат мора да буде детерминистички. Када би био недетерминистички, постојала би могућност да аутомат пређе у различита стања за исте улазе. Стога, стања на аутоматима потенцијално не би била усаглашена на свим репликама сервера.

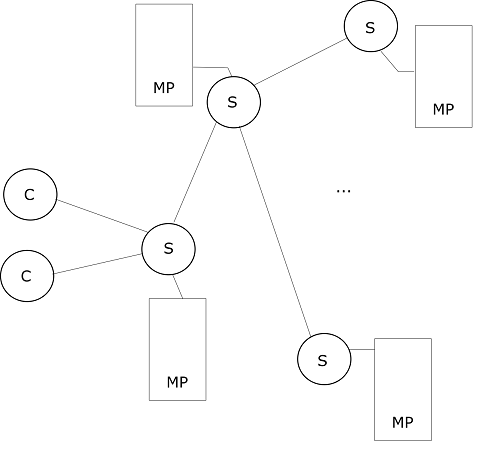
Како је наведено у [7], кључ за имплементацију SMR je:

* **Координација реплика** – све реплике примају и обрађују исту секвенцу захтева. Ово даље може да се декомпонује на:
  + - **Договор** – свака исправна реплика сервера прима сваки захтев
    - **Редослед** – свака исправна реплика сервера обрађује захтеве које прима у истом релативном редоследу.

SMR приступ омогућава отпорност на отказе ослањајући се на претпоставку да се у систему никад неће појавити више од неког броја неисправних/малициозних реплика сервера. У [7], тврди се да овај приступ гарантује исправно функционисање уколико је број неисправних реплика мањи или једнак трећини броја свих реплика у систему. Ово је један од главних принципа на којима почива блокчејн. Постизање консензуса и једногласна одлука о томе шта је следећи блок трансакција није ништа друго него колективно одлучивање о томе шта ће бити следеће стање свих детерминистичких коначних аутомата.

* + 1. Мемпул компонента

Поново, претпоставићемо клијент-сервер архитектуру са SMR приступом, тако да сваки сервер има своју реплику детерминистичког коначног аутомата, односно блокчејн.



Слика 2.2.4 – клијент-сервер архитектура заједно са Мемпул компонентом

* ***S*** – Сервер (реплика сервера)
* ***C*** – Клијент
* ***MP*** – Мемпул (*Memory pool*)

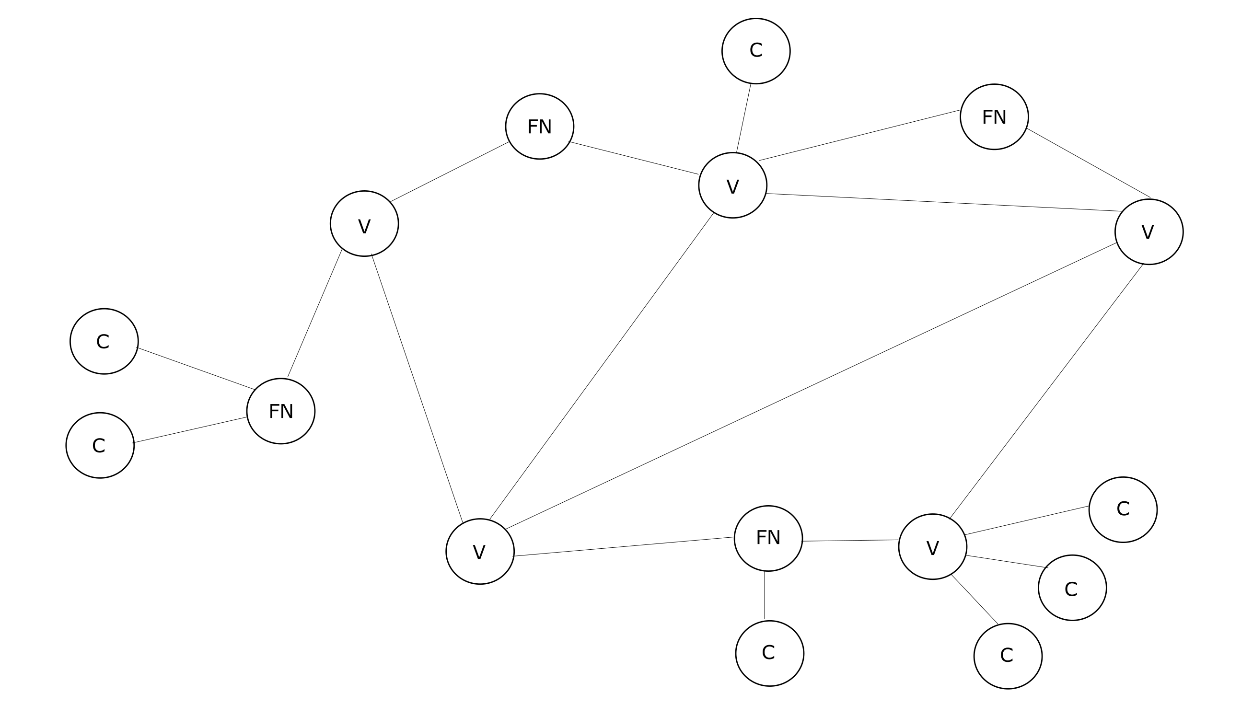
Клијент може да пошаље захтев ка серверу, да би добио одговор. Према SMR приступу, то значи да сваки сервер мора да прими захтев, а затим да га обради. Међутим, као што је илустровано на слици 2.2.4, клијент је конектован на само један сервер. Са друге стране, сервери међусобно нису повезани сваки са сваким, већ је сваки сервер повезан са подскупом осталих реплика. Ово је једно од јако битних ограничења у Тендерминту, где се ради о WAN (*Wide area network*), а не LAN (*Local area network*) мрежи, где сервери нису део само једног административног домена. Из тог разлога, немогуће је да сваки сервер буде повезан са свим осталима.

Да би клијентски захтев дошао до свих сервера у мрежи, мора се користити одређени алгоритам за дисеминацију, који ће клијентску трансакцију да дистрибуира по мрежи, тако да она стигне до свих сервера. Алгоритми за дисеминацију података у дистрибуираним системима се у литератури често називају и „протоколи трачарења“ (енг. *Gossip communication protocols*) или „епидемични протоколи“ (енг. *Epidemic protocols*) [8]. Дисеминација клијентске трансакције представља једну од две улоге Мемпул компоненте у Тендерминту.

Алгоритам за дисеминацију се заснива на периодичном прослеђивању одређене информације ка суседним серверима (енг. *peers*). Често је претпоставка да један сервер зна за све друге сервере у мрежи, тако да може периодично да одабере одређени подскуп њих и да им пошаље поруку. Као што је напоменуто малопре, у Тендерминту ово није случај, из разлога што сваки сервер зна само за подскуп осталих. Стога, у Тендерминту би алгоритам за дисеминацију захтевао да сервер периодично шаље поруке свима у том подскупу. Овакво периодично слање порука доводи до веома брзог ширења информација кроз мрежу. С друге стране, постоји редундантност у пријему порука – могуће је да сервер прими поруку која садржи информацију о нечему што он већ зна. С једне стране, ово представља сувишно оптерећење за сервер. С друге стране, редундантност је добра у случају отказа неког од сервера или губитка поруке у току преноса, јер иста порука по више различитих путева у мрежи може да дође до истог сервера.

Друга улога Мемпул компоненте јесте да складишти трансакције које још увек нису део блокчејна. Када клијентска трансакција дође до сервера, она свакако бива дисеминрана по мрежи. Међутим, јасно је да не може да буде обрађена истог тренутка, из разлога што мора бити постигнут консензус од стране свих сервера, који ће ту трансакцију укључити у блокчејн. Док се то не деси, клијентска трансакција чува се унутар посебне RAM (*Random access memory*) меморије сервера. Ова меморија садржи трансакције које још увек нису део блокчејна. Када се донесе консензус и одлучи се који је следећи блок трансакција који ће бити додат у блокчејн (односно које је следеће стање коначног аутомата), трансакције из тог блока се бришу из RAM меморија свих сервера. У оквиру Тендерминт мреже, трансакције се, уместо у RAM меморији, чувају на диску.

* + 1. Модел система



Слика 2.2.5 – Модел система

* ***C*** - Клијент
* ***FN*** – Стандардни сервер (енг. *Full node*)
* ***V*** – Валидатор сервер (енг. *Validator node*)

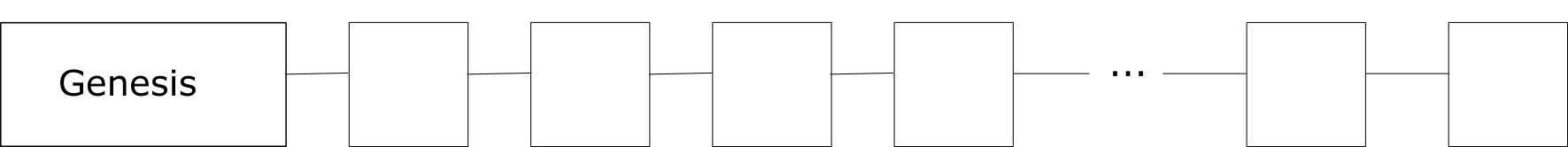
Претпоставимо клијент-сервер архитектуру која се састоји од више врста чворова (енг. *node*), приказану на слици 2.2.5. Нека су ти чворови означени са FN и V. У реалном систему, чвор може бити представљен или једним сервером, или већим бројем сервера (нпр. унутар организације). Без губитка општости, у даљем тексту се претпоставља да је сваки чвор представљен једним сервером, па ће се из тог разлога разликовати две врсте чворова:

* + FN сервер
  + V сервер

Сваки V сервер је такође и FN сервер. Међутим, супротно не важи – FN сервер може и не мора бити V сервер. На слици 2.2.5, сваки сервер означен као FN није и V сервер.

Нека постоји скуп од ***m*** FN сервера. Сервери у моделу система нису део једног административног домена. Као последица тога, потпуна мрежна повезаност (енг. *full-mesh connectivity*) између њих не може бити спроведена. Из тог разлога, сваки FN сервер у мрежи може бити повезан само са подскупом других FN сервера. Овај подскуп назваћемо ***подскуп суседа*** (енг. *peer subset*). Везе између сервера приказане на слици 2.2.5 су бидирекционе, тако да је могуће успоставити двосмерну комуникацију. Подскуп суседа је подложан променама како време пролази, дакле није статички.

Претпоставка је да је минимална величина подскупа суседа 1. Стога, не може постојати FN сервер који није повезан ни са једним другим сервером. Када би мрежа била моделирана помоћу графа, такав граф би увек био повезан. Сваки FN сервер има јавну IP (*Internet protocol*) адресу. Сваки FN сервер има RAM меморију која је део Мемпул компоненте. У овој меморији се чувају трансакције које још увек нису део блокчејна. У оквиру система постоји блокчејн који је реплициран на сваком FN серверу. Илустрација блокчејна дата је на слици 2.2.6.



Слика 2.2.6 – Блокчејн на сваком серверу

Блокчејн се састоји од уланчане листе блокова трансакција. Везе су моделиране хеш функцијама, то јест сваки блок садржи хеш функцију претходног блока. Први блок у блокчејну назива се ***блок генезе*** (енг. *Genesis block*). Сви блокови у блокчејну су потомци блока генезе.

Модел система објашњен у овом потпоглављу је поједностављен у односу на модел система који се користи у Тендерминту. У Тендерминту постоји још неколико врста чворова. Међутим, сви принципи који се тичу функционисања Мемпул компоненте су потпуно исти као и у Тендерминту. Стога, алгоритми који су предложени у овом документу могу се врло лако прилагодити архитектури мреже која постоји у Тендерминту.

Као резиме, у наставку су наведене све претпоставке усвојене у моделу система:

1. FN сервери имају јавне IP адресе
2. Сервери нису повезани тако да је повезан сваки са сваким, већ сваки сервер има свој подскуп суседа.
3. FN сервер може да мења свој подскуп суседа динамички.
4. Минимална величина подскупа суседа је 1.
5. Мрежа није отпорна на отказе. Сервери у мрежи су потпуно поуздани.
   * 1. Дисеминација клијентске трансакције и постизање консензуса

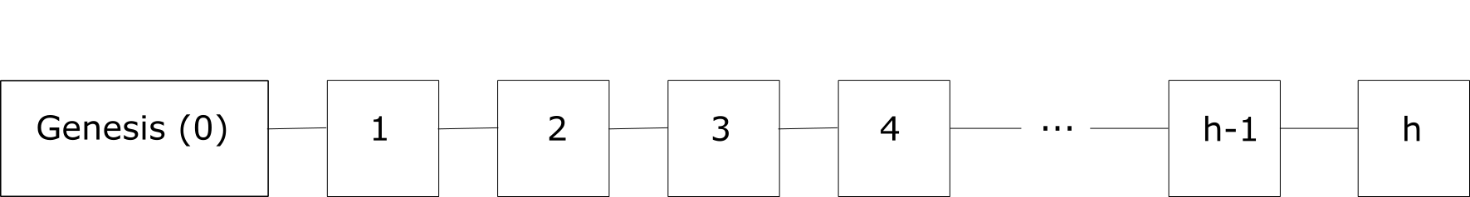
У овом потпоглављу биће објашњени они елементи консензус алгоритма Тендерминта који су уско повезани са алгоритмом за дисеминацију клијентске трансакције. Консензус алгоритам Тендерминта је детаљно објашњен у [1].

Као што је речено раније, клијент жели да његова трансакција буде обрађена од стране Тендерминт мреже. Трансакција може бити било који скуп података који по извршавању доноси неку корист клијенту (нпр. новчана трансакција). Имплементационо, трансакција се посматра само као низ бајтова. Из клијентске перспективе, битна су два догађаја:

* + Трансакција мора бити извршена. <1>
  + Доказ о извршењу трансакције мора бити смештен унутар блокчејна. <2>

Да би то постигао, клијент може да се конектује на FN сервер по избору. По успостављању конекције, клијент може да пошаље трансакцију том серверу. У било ком тренутку, клијент може да одлучи да се дисконектује са конектованог сервера, и повеже се са неким другим FN сервером.

Када FN сервер прими трансакцију, најпре проверава њену валидност. Клијент може да пошаље трансакцију која је невалидна по неким критеријумима Тендерминта. Оваква трансакција се одбацује и не обрађује се. Ако је трансакција валидна, сервер је складишти унутар своје RAM меморије. Додатно, трансакција бива дисеминирана кроз мрежу, најпре ка свима у подскупу суседа одговарајућег сервера. Када суседи приме трансакцију, поступак се понавља. На овај начин трансакција се дистрибуира до свих у мрежи. Као што је речено у претходном потпоглављу, на сваком серверу постоји блокчејн, приказан на слици 2.2.7.



Слика 2.2.7 – Блокчејн на сваком серверу

Сваки блок у блокчејну описан је редним бројем, који се у Тендерминту назива ***висина*** (енг. *height*). Нека је тај број означен са ***h***. Сваки блок садржи трансакције које не само да су део трајне историје, већ и бивају извршене при додавању у блокчејн. Ово задовољава клијентске услове <1> и <2>.

Да би нов блок био додат у блокчејн, скуп валидатор сервера (V сервера) мора да постигне консензус за сваку висину блокчејна ***hb***. Висина блокчејна представља редни број блока који ће бити додат у блокчејн, односно број блокова у блокчејну изузев блока генезе.

Тај скуп V сервера у Тендерминту се назива ***скуп валидатора*** (енг. *validator set*). Скуп валидатора је скуп FN сервера који се бира при сваком извршавању консензус алгоритма, дакле када год је потребно додати нови блок у блокчејн. Нека је величина овог скупа означена са ***n***. Важи да је ***n < m***, где је ***m*** број свих сервера у мрежи. Кад год се неки FN сервер дода у скуп валидатора, он постаје V сервер. Блок генезе садржи информације о првом скупу валидатора икада.

Може се рећи да се висина блокчејна мења при сваком извршавању консензус алгоритма. Дакле, за сваку нову висину блокчејна, мења се и скуп валидатора. Скуп валидатора одређује се од стране неког спољашњег процеса. Ова информација преузета је из [18]. Алгоритми за одређивање скупа валидатора превазилазе оквире овог рада.

Међутим, скуп валидатора може и кроз више извршавања консензус алгоритма да остане исти. Како је наведено у [1], скуп валидатора се бира тако да се што више искористи њихова ***моћ гласања*** (енг. *voting power*). Сваком FN серверу придружен је број који представља његов удео у доношењу коначне одлуке при гласању о томе шта је следећи блок трансакција. Из тог разлога, могуће је да се скуп валидатора никад неће ни мењати, на пример у мрежи где моћ гласања одређеног броја сервера далеко надмашује моћ гласања осталих сервера. Како ово није општи случај, оваква претпоставка не сме се усвојити. Тендерминтов консензус алгоритам гарантује успешност уколико је моћ гласања неисправних процеса мања од трећине укупне моћи гласања свих процеса у мрежи.

Даље, свако извршавање консензус алгоритма састоји се од ***k*** рунди, где се у свакој рунди бира нови ***предлагач*** (енг. *proposer*) из скупа валидатора. Предлагач је V сервер чија је дужност да предложи одређени број трансакција из своје RAM меморије. Он то ради тако што направи блок од тих трансакција, и предложи га осталима у скупу валидатора. Ако тај блок буде усвојен и постигне се консензус за њега, неће бити више рунди у тренутном извршавању консензус алгоритма. Нова рунда се иницира само онда када у претходној рунди није могла бити донета једногласна одлука. Такође, постоји вероватноћа да кроз све рунде консензус алгоритма предлагач остане исти, али ни ово не важи у општем случају.

Овим долазимо до потребног, али не и довољног услова у вези Мемпул компоненте и дисеминације трансакција из ње – када FN сервер прими клијентску трансакцију код себе, он мора да је дисеминира тако да се гарантује (или бар тврди са великом вероватноћом) да ће та трансакција временом доћи до RAM меморије бар једног FN сервера који ће бити члан скупа валидатора (дакле, V сервер) и предлагач у некој рунди ki на некој висини блокчејна hbj у будућности**.** Овај след догађаја довешће до тога да трансакција буде предложена, а самим тим и извршена и додата у блокчејн.

Услов није довољан из три разлога. Први је тај да максимални број трансакција у предложеном блоку може бити мањи од броја трансакција у RAM меморији предлагача. Уколико је то случај, чак и да трансакција дође до неког предлагача, може се десити да он не може да је предложи из разлога што за њу нема места у предложеном блоку, и да притом тај предлагач више никад у будућности не буде предлагач. Други разлог је да је скуп валидатора динамички и подложан променама, дакле није познат унапред. Трећи разлог је да је предлагач у свакој рунди потенцијално различит сервер из скупа валидатора, тако да је и он непознат унапред. Из наведеног, **може се закључити да је потребно да клијентска трансакција доспе до свих сервера у мрежи, да би се гарантовали догађаји <1> и <2>.**

Мемпул компонента се сматра уским грлом система. RAM меморија садржи трансакције које нису део блокчејна, и што се брже празни, брже ће се трансакције извршавати. У случају да се та меморија препуни, клијентска трансакција неће моћи да се изврши. Штавише, овако нешто може довести и до отказа сервера. Из тог разлога, битно је да се RAM меморија што брже празни. Као што је речено, она се празни кад год се нека трансакција коју садржи дода у блокчејн. Да би се трансакција додала у блокчејн, неопходно је да што брже дође до свих сервера у мрежи, да би неко од предлагача из скупа валидатора могао да је предложи. На ту брзину директно утиче алгоритам дисеминације трансакција.

Као резиме, у наставку су дате чињенице које се тичу консензус алгоритма, а које су релевантне за анализу алгоритама за дисеминацију, која следи у наредним поглављима:

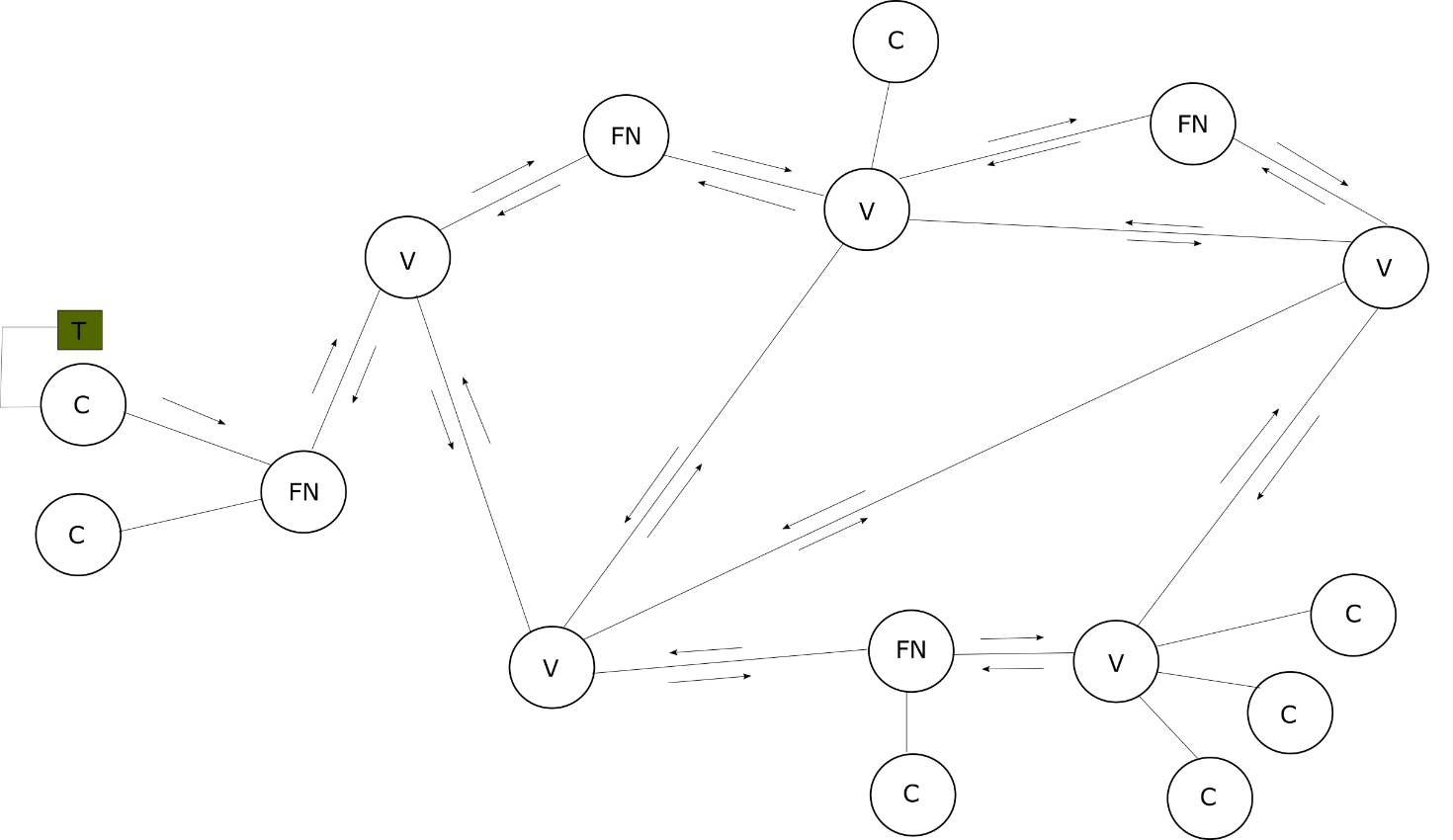
1. Скуп валидатора је подскуп скупа FN сервера – ***n < m***.
2. Скуп валидатора је динамички – потенцијално се мења при сваком извршавању консензус алгоритма.
3. Предлагач у свакој рунди бира се динамички – потенцијално се мења у свакој рунди.
4. Формална анализа алгоритама

У овом поглављу, пре свега биће представљена формална анализа тренутног алгоритма за дисеминацију који се користи у Тендерминту. Након тога, биће приложена формална анализа три различита алгоритма, од којих се сваки заснива на једном од приступа који су поменути у уводу овог документа. Анализа сваког алгоритма биће праћена псеудокодом, дискусијом о перформансама, и начинима функционисања алгоритма у случају мреже која би требало да буде отпорна на отказе.

* 1. Тренутно решење

У овом потпоглављу биће приложена дискусија о тренутном алгоритму дисеминације трансакција који се користи у Тендерминту.

* + 1. Објашњење алгоритма



Слика 3.1.1 – Тренутни алгоритам за дисеминацију трансакција у Тендерминту

* ***C*** - Клијент
* ***FN*** – Стандардни сервер (енг. *Full node*)
* ***V*** – Валидатор сервер (енг. *Validator node*)
* ***T*** – Клијентска трансакција

Претпоставимо да клијент жели да његова трансакција буде обрађена од стране Тендерминт мреже, на начин који је објашњен у претходном поглављу, као што је илустровано на слици 3.1.1. Да би то урадио, клијент шаље своју трансакцију ка FN серверу са којим се повезао. Псеудокод тренутног алгоритма за дисеминацију који се користи у Тендерминту дат је у наставку. Овај алгоритам извршава се на сваком FN серверу.

Табела 3.1.1 – Тренутно решење – псеудокод

|  |
| --- |
| 1. upon receive(T transaction, Node sender) { 2. // Node = {C, FN, V} 4. **if** (sender == C) { 5. // u Tendermintu ovaj uslov ne postoji 7. bool valid = checkTx(transaction); 8. **if** (valid == **false**) { 9. **return**; 10. } 11. } 13. bool isInMyMempool = checkMempool(transaction); 14. **if** (isInMyMempool == **true**) { 15. **return**; 16. } 18. addMempool(transaction);  21. **for** (Node node : getPeerSubset()) { 22. node.send(transaction, self); 23. } 25. } |

Кад год FN сервер прими трансакцију, било од клијента, било од некога из свог подскупа суседа, позива се функција ***receive(T, Node)***. Унутар функије, прво се проверава валидност трансакције позивом функције ***checkTx(T)***. Имплементација функције ***checkTx(T)*** у програмском језику *Go* може се наћи у [19]. С обзиром на то да је претпоставка у нашем моделу система та да је мрежа потпуно поуздана, сматрамо да један сервер увек има потпуно поверење у све сервере из његовог подскупа суседа. Из тог разлога, провера валидности трансакције врши се само уколико је трансакцију послао клијент. Битно је приметити да се у реалном систему у Тендерминту, провера валидности увек врши, из разлога што је могуће да је неки од сервера неисправан или малициозан. Уколико је трансакција невалидна, не ради се ништа. Уколико је трансакција валидна, проверава се да ли је она већ унутар RAM меморије посматраног сервера. Уколико јесте, примљена је редундантна порука и не ради се ништа. Уколико није, трансакција се додаје у скуп трансакција које још увек нису део блокчејна. Након тога се та иста трансакција шаље свим серверима у подскупу суседа.

* + 1. Анализа перформанси

Ако претпоставимо да је мрежа сервера моделирана повезаним графом који има ***e*** чворова и ***v*** грана, тада је број порука које се размене за једну клијентску трансакцију једнак ***2v***, као што је приказано на слици 3.1.1. Оптималан број порука које се размене за једну клијентску трансакцију, где би сваки сервер „чуо“ нову информацију само једном, био би једнак ***e-1*** (не узимајући у обзир слање трансакције од стране клијента ка FN серверу). Број редундантних порука које сваки сервер у мрежи прими је јако велики, што ће бити показано у резултатима симулације. Ово је последица тога да се свака примљена порука напросто прослеђује даље кроз мрежу. На ово додатно утичу циклуси који постоје у графу мреже.

* + 1. Дискусија решења

Тренутни алгоритам за дисеминацију који се користи у Тендерминту гарантује да ће порука стићи до свих сервера у мрежи, уколико је минимална величина подскупа суседа једнака 1, то јест нема неповезаних сервера. Како порука гарантовано стиже до свих сервера, тиме су задовољени услови <1> и <2>. Самим тим што се порука шаље свим суседима, овакав алгоритам је отпоран на отказе у мрежи. Међутим, из свега наведеног може се закључити да тренутно решење које се користи у Тендерминту доводи до пријема великог броја редундантних порука, што ће резултати симулације и показати.

Једноставно побољшање тренутног решења било би да се, у току слања трансакције серверима у подскупу суседа, порука не шаље серверу који ју је претходно послао, уколико се заиста ради о поруци примљеној од стране суседног сервера, а не клијента. Горе наведена ***receive(T, Node)*** функција, сада би изгледала као у псеудокоду датом у наставку.

Табела 3.1.2 – Једноставно побољшање тренутног решења

|  |
| --- |
| 1. //... 2. **for** (Node node : getPeerSubset()) { 3. **if** (node != sender) { 4. node.send(transaction, self); 5. } 6. } 7. //... |

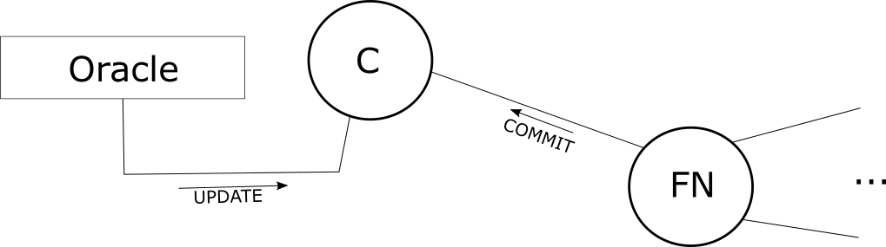
* 1. Алгоритам за дисеминацију заснован на клијенту

Алгоритам за дисеминацију објашњен у овом потпотглављу заснован је на првом од три приступа наведена у уводном делу овог рада. Идеја је да је за дисеминацију одговоран само клијент.

* + 1. Објашњење алгоритма

У литератури може се наћи варијанта оваквог алгоритма у којој би клијент своју трансакцију слао свим FN серверима у мрежи [9]. Наравно, ово би захтевало да клијент познаје IP адресе свих FN сервера у мрежи. Овакав приступ би сигурно задовољио услове <1> и <2>. У оваквој ситуацији, клијент би био оптерећенији јер је одговоран за дисеминацију, али овакав приступ није фокусиран на оптимизацију искоришћења ресурса на клијенту.

Идеја приложена овде још је ефикаснија од оне предложене у [9]. Претпоставка је да клијент код себе има локални модул, који се у литератури често назива и *Oracle* [10], као што је приказано на слици 3.2.1. У нашем моделу система, овај модул способан је да прати промене у скупу валидатора. Кад год се деси нека промена у скупу валидатора, клијент бива обавештен о томе. Клијент сада може своју трансакцију да пошаље само онима у тренутном скупу валидатора.



Слика 3.2.1 – Илустрација алгоритма за дисеминацију заснованог на клијенту

* ***C*** - Клијент
* ***FN*** – Стандардни сервер (енг. *Full node*)

Табела 3.2.1 – Алгоритам за дисеминацију заснован на клијенту – псеудокод

|  |
| --- |
| 1. Set<Node> currentVS = EMPTYSET; 2. Set<T> uncomitted = EMPTYSET; 4. upon receive(UPDATE, Set<Node> newVS) { 5. **for** (Node node : newVS) { 6. **if** (!currentVS.contains(node)) { 7. node.send(uncomitted); 8. } 9. } 11. currentVS = newVS; 12. } 14. upon receive(COMMIT, T transaction) { 15. uncomitted.remove(transaction); 16. } 18. upon send(T transaction) { 19. **for** (Node node : currentVS) { 20. node.send(transaction); 21. } 23. uncomitted.add(transaction); 24. } |

Клијент локално има два скупа – тренутни скуп валидатора и скуп трансакција које још увек нису ***комитоване*** (односно додате у блокчејн). Када клијент жели да пошаље своју трансакцију на извршавање, позива се функција ***send(T)***, која одговарајућу трансакцију шаље свима у скупу валидатора, а затим је додаје у скуп некомитованих трансакција. Када је за неку трансакцију постигнут консензус, одговарајући FN сервер послаће ***COMMIT*** поруку клијенту. По пријему ***COMMIT*** поруке, клијент уклања одговарајућу трансакцију из скупа некомитованих трансакција. Поред тога, кад год се деси промена у скупу валидатора, локални модул клијента (*Oracle*) шаље ***UPDATE*** поруку клијенту. По пријему ове поруке, клијент шаље своје некомитоване трансакције само онима из новог скупа валидатора који нису у старом скупу валидатора. На овај начин некомитоване трансакције ће увек доћи до свих у скупу валидатора. Након слања ових трансакција, ажурира се тренутни скуп валидатора.

* + 1. Анализа перформанси

Уколико величину скупа валидатора означимо са ***n***, тада је, у идеалном случају, број порука које се размене за једну клијентску трансакцију једнак ***n***. Уколико трансакција не буде додата у блокчејн пре него што дође до промене у скупу валидатора, захтева се да она буде послата разлици скупова ***newVS*** и ***currentVS (newVS \ currentVS)***. Дакле, максимални број порука које ће се разменити за једну клијентску трансакцију зависи од консензус алгоритма. Уколико се за послату трансакцију постигне консензус у првом извршавању алгоритма након што је она послата, неће бити потребе за поновним слањем те трансакције по пријему ***UPDATE*** поруке. Што се тиче броја редундантних порука које прими сваки сервер у мрежи, он је једнак 0. Поруке ће примити само они који су у скупу валидатора, и то само једном.

* + 1. Дискусија решења

Алгоритам за дисеминацију заснован на клијенту гарантује да ће некомитована трансакција увек доћи до свих у скупу валидатора. Самим тим, задовољени су услови <1> и <2>. У случају неисправних сервера у скупу валидатора, алгоритам ће и даље исправно радити, с обзиром на то да се трансакција шаље свима у скупу валидатора. Ако претпоставимо да број неисправних валидатора никад неће премашити ***n/3***, и да се предлагач бира са униформном расподелом, увек ће доћи до предлагања клијентске трансакције.

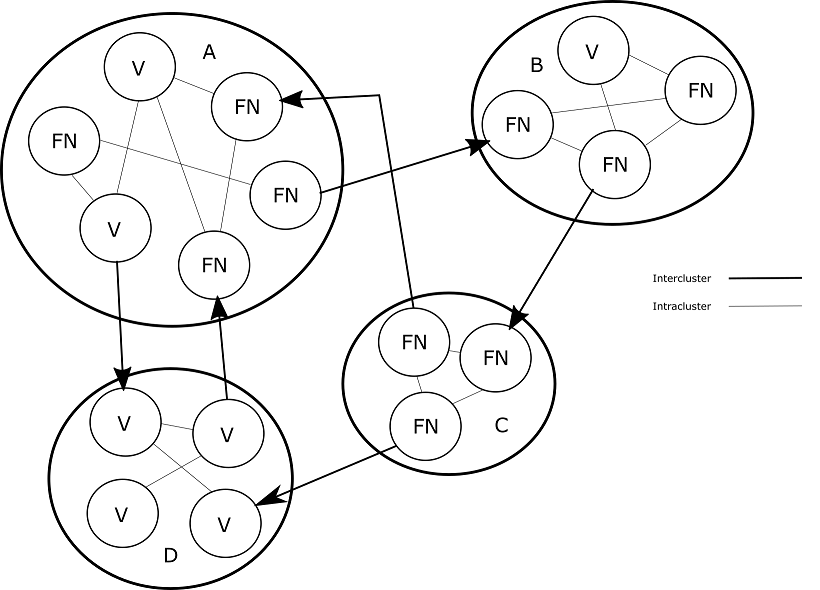
Важно је приметити да овакав алгоритам није лак за имплементацију. Поставља се питање како имплементирати локални модул клијента – *Oracle*.С друге стране, разматрање идеалног решења је корисно као референца према којој се може мерити редундантност у решењима која се користе у реалним системима.

* 1. Кластеровање стабала

Алгоритам објашњен у овом потпоглављу заснива се на другом приступу - класификацији сервера у мрежи у структуриране групе. Као што је напоменуто раније, сервери у моделу система нису део једног административног домена. Они су део велике дистрибуиране мреже, где потпуна мрежна повезаност не може бити успостављена. Из тог разлога, FN сервер може бити повезан само са подскупом других сервера. Овај подскуп назвали смо *подскуп суседа.* Претпоставили смо да је минимална величина подскупа суседа 1. Стога, уколико бисмо мрежу моделирали графом, тај граф би увек био повезан. Поставља се питање да ли можемо да распоредимо у одређене групе оне сервере у мрежи који су „близу“ једни другима по неким критеријумима. Те групе у литератури називају се ***кластери*** (енг. *cluster*).

* + 1. Објашњење алгоритма

Идеја алгоритма предложеног овде комбинује два решења предложена у [11] – ***Flat membership server based protocol*** ([11] – 2.3.1) и ***Hierarchical membership protocol*** ([11] – 2.3.2). Међутим, та два решења примењена директно на наш модел система не би побољшала перформансе алгоритма за дисеминацију у значајној мери. Стога, у даљем тексту се предлаже додатна модификација комбинације два поменута решења, са циљем да се смањи редундантност при размени порука.



Слика 3.3.1 – Подела мреже у кластере – *Hierarchical membership protocol*

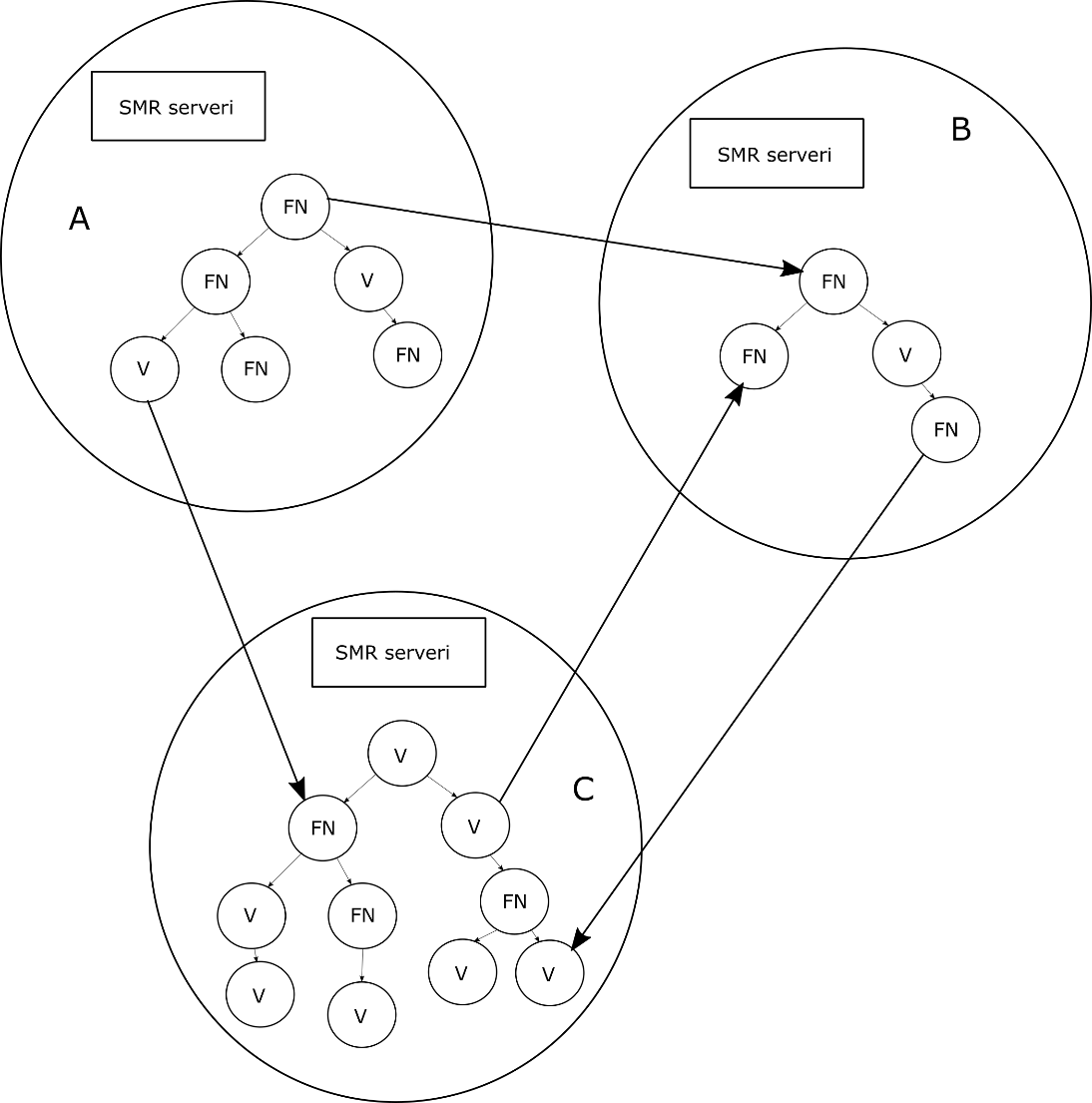
* ***FN*** – Стандардни сервер (енг. *Full node*)
* ***V*** – Валидатор сервер (енг. *Validator node*)

Пре свега, претпоставимо да је мрежа сада подељена у кластере. Како је речено у [11], сервери се групишу у кластере према географској удаљености или некој метрици која се тиче близине у мрежи (нпр. кашњење повратног путовања (енг. *round trip delay*) или број мрежних хопова између сервера). Кластерима може да управља неки спољашњи процес [11]. У [11], приложен је математички доказ који показује да је потребан само мали број веза (линкова) између кластера да би они били повезани. На слици 3.3.1, постоје четири кластера – *A*, *B*, *C* и *D*. Такође, разликоваћемо два типа линкова:

* + ***Интеркластерни*** (енг. *Intercluster*) – представљају линкове између различитих кластера, унидирекциони су.
  + ***Интракластерни*** (енг. *Intracluster*) – представљају линкове између сервера унутар истог кластера, бидирекциони су.

Сервер може бити одговоран само за један интеркластерни линк – ово је такође рестрикција представљена у [11].

Како број интеркластерних линкова не мора бити велики, смањили смо број грана у графу мреже. Такође, груписањем сервера према неком критеријуму „близине“, успоставили смо бржу комуникацију између њих. Без обзира на то, редундантност при пријему порука и даље постоји, како унутар кластера, тако и унутар целе мреже. Најпре ћемо размотрити решавање проблема редундантности унутар кластера, а затим и унутар целе мреже.



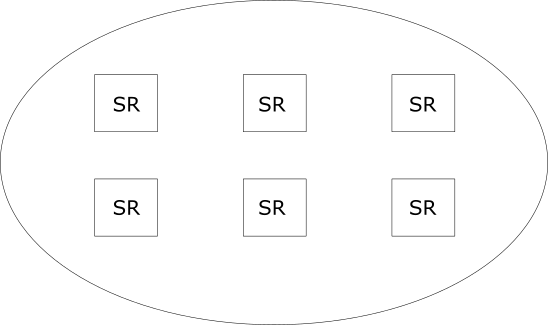
Слика 3.3.2 – Решавање проблема редундантности унутар кластера – *Flat membership server based protocol*

* ***FN*** – Стандардни сервер (енг. *Full node*)
* ***V*** – Валидатор сервер (енг. *Validator node*)

Узрок редундантности приликом дисеминације поруке кроз мрежу јесу циклуси у графу мреже. Из тог разлога, предлажемо структуру података која не садржи циклусе, и чији су линкови унидирекциони – стабло. Када би сервери унутар кластера били повезани тако да формирају стабло, као што је приказано на слици 3.3.2, сваки сервер примио би поруку само једном. Међутим, постављају се два питања:

1. Ко формира стабло унутар кластера?
2. Како пропагирати поруку навише у стаблу, уколико користимо унидирекционе линкове?

Да бисмо одговорили на прво питање, приступ објашњен у [11], *Flat membership server based protocol*, може се применити.



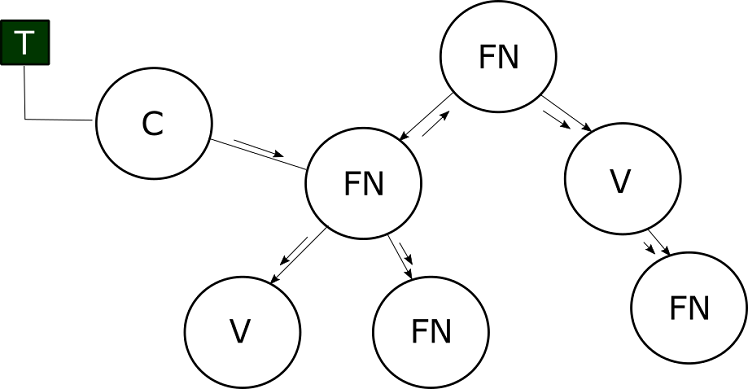
Слика 3.3.3 – SMR сервери који се користе за формирање стабала

* ***SR*** – Реплика сервера (енг. *Server replica*)

Нека постоји скуп SMR сервера, где сваки сервер има своју реплику детерминистичког коначног аутомата, као што је илустровано на слици 3.3.3. Ове сервере потребно је разликовати од FN и V сервера. Када год нови FN сервер жели да постане део мреже, што може да се деси потпуно асинхроно и стога је динамички процес, он шаље захтев скупу SMR сервера који је задужен за „најближи“ кластер тог FN сервера. С обзиром да се ради о реплицирању коначних аутомата, FN сервер није свестан постојања скупа SMR сервера. За њега је све то потпуно транспарентно. Из његове перспективе, он контактира један сервер са захтевом да постане део кластера. Одговорност је свих сервера у скупу SMR сервера да постигну консензус о томе где у стаблу ће се налазити нови FN сервер. Након постизања консензуса, SMR сервери моћи ће колективно да пређу у следеће стање аутомата.

Поред тога што би морали да одговарају на захтеве FN сервера који желе да постану део кластера, SMR сервери би такође морали да одржавају стабло што ефикаснијим (тако да је оно балансирано или чак комплетно). Препорука је да се стабло мења динамички с времена на време, тако да одређени чвор не остане корени чвор, унутрашњи чвор или лист дуго времена. Листови у стаблу пропагирају поруке много мање него остали чворови, и због тога су мање искоришћени. Ово је лоше уколико они имају велики пропусни опсег (енг. *bandwidth*). Периодична реконструкција стабла оптималније искоришћава пропусне опсеге чворова у стаблу [12]. Реплицирање сервера елимише SPOF проблем. Међутим, између њих мора постојати одговарајућа синхронизација и стога консензус алгоритам који би се на њима извршавао мора бити пажљиво одабран.

Да бисмо дали одговор на друго питање, размотрићемо ситуацију илустровану на слици 3.3.4.



Слика 3.3.4 – Пропагација клијентске трансакције при коришћењу стабала

* ***C*** - Клијент
* ***FN*** – Стандардни сервер (енг. *Full node*)
* ***V*** – Валидатор сервер (енг. *Validator node*)
* ***T*** – Клијентска трансакција

Када FN сервер прими клијентску трансакцију, он може лако да је пропагира низ сопствено подстабло. Међутим, како обезбедити да порука стигне до остатка главног стабла? Да бисмо решили овај проблем, претпоставићемо да сваки чвор у стаблу садржи информације о свом родитељу (нпр. садржи IP адресу родитеља). Стога, када FN сервер приказан на слици 3.3.4 прими клијентску трансакцију, он може да је пропагира ка свом родитељу и својим потомцима. Ово се понавља рекурзивно. Могу се усвојити следећа правила:

* + Када сервер у стаблу прими поруку од детета или клијента, он мора да је проследи ка свом родитељу – ***дисеминација навише*** (енг. *upward gossiping*).
  + Сервер увек мора да проследи поруку низ сопствено подстабло – ***дисеминација наниже*** (енг. *downward gossiping*).

Псеудокод оваквог решења дат је у наставку.

Табела 3.3.1 - Пропагација клијентске трансакције при коришћењу стабала – псеудокод

|  |
| --- |
| 1. // Pseudokod za FN/V server - bez klastera 3. Node parent = ...; 4. Set<Node> children = ...; 6. upon receive(T transaction, Node sender) { 7. **if** (sender == C) { 8. bool valid = checkTx(transaction); 9. **if** (valid == **false**) { 10. **return**; 11. } 12. } 14. addMempool(transaction); 16. **if** (sender == C || children.contains(sender)) { 17. **if** (parent != nil) { 18. parent.send(transaction, self); 19. } 20. } 22. **for** (Node child : children) { 23. **if** (child == sender) { 24. **continue**; 25. } 27. child.send(transaction, self); 28. } 29. } |

Важно је приметити да сада нема потребе за позивањем ***checkMempool(T)*** функције пре додавања трансакције у RAM меморију сервера. Ово је последица тога да сваки FN сервер прима трансакцију само једном. Ако изузмемо кластере, овакво решење даје идеалне перформансе. Међутим, ако узмемо кластере у обзир, редундантност постоји.

Најпре, објаснићемо како је претходна тврдња истинита. Ако претпоставимо да постоје три кластера, *A*, *B* и *C*, као што је приказано на слици 3.3.2, можемо да приметимо да је мрежа опет моделирана графом, и то цикличним. На пример, следећи интеркластерни линкови формирају циклус:

* + - *B* → *C*
    - *C* → *B*

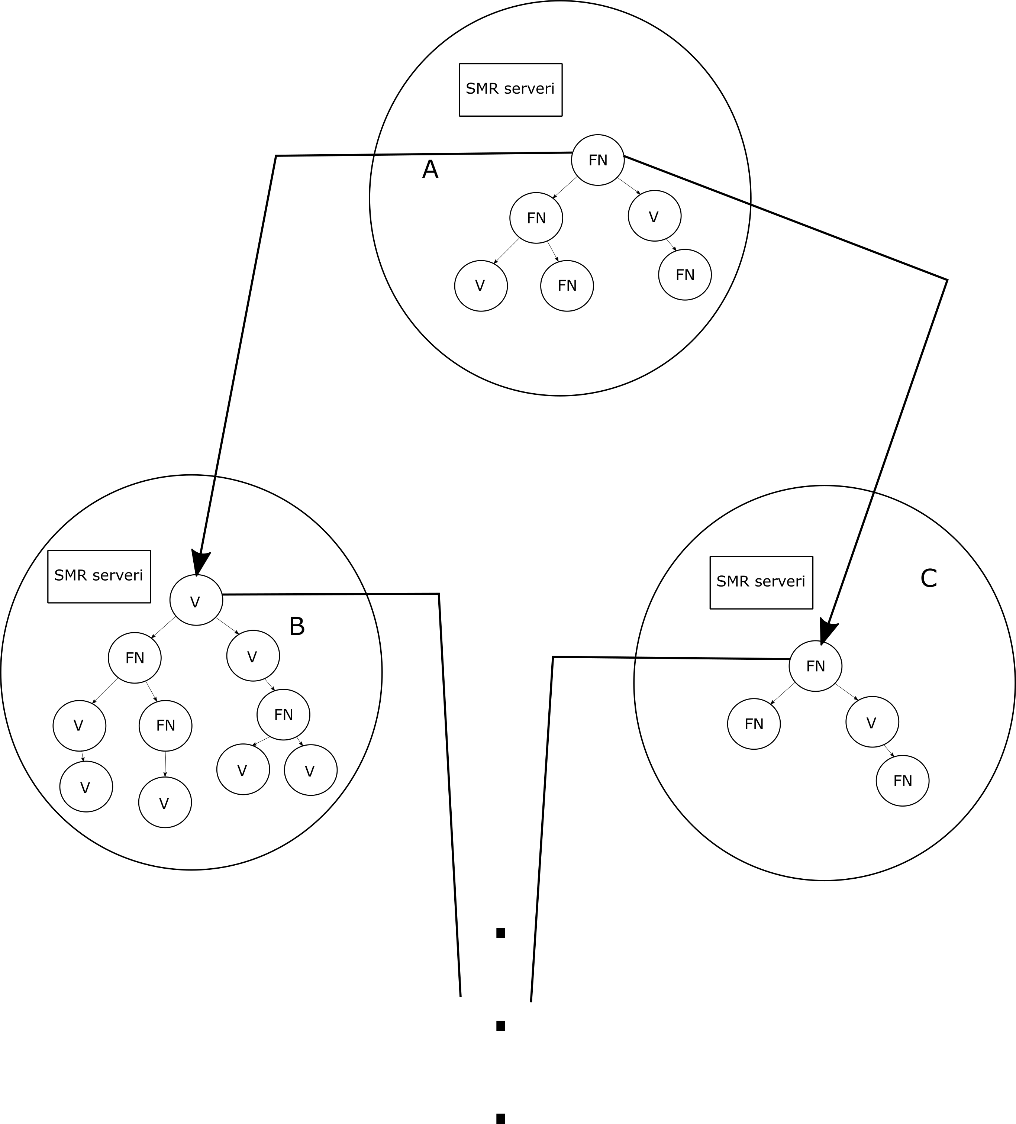
Уколико би постојао клијент у кластеру *B*, који би послао своју трансакцију било ком FN серверу у том кластеру, та трансакција била би дисеминирана локално кроз цео кластер преко интракластерних линкова. Додатно, била би послата у кластер *C* преко интеркластерног линка *B* → *C*. Унутар кластера *C*, трансакција би поново била дисеминирана локално, и затим поново враћена у кластер *B* преко интеркластерног линка *C* → *B*. Да би се избегло додавање дупликата у RAM меморију сервера, овога пута потребно је позвати функцију ***checkMempool(T)***. У даљем тексту дат је псеудокод за FN сервер, с тим што су овог пута кластери узети у обзир.

Табела 3.3.2 – Решавање проблема редундантности у целој мрежи – псеудокод

|  |
| --- |
| 1. // Pseudokod za a FN/V server - sa klasterima 3. Node parent = ...; 4. Set<Node> children = ...; 5. Node interclusterLink = ...; 7. upon receive(T transaction, Node sender) { 8. **if** (sender == C) { 9. bool valid = checkTx(transaction); 10. **if** (valid == **false**) { 11. **return**; 12. } 13. } 15. bool isInMyMempool = checkMempool(transaction); 16. **if** (isInMyMempool == **true**) { 17. **return**; 18. } 20. addMempool(transaction); 22. **if** (sender == C || children.contains(sender) || isFromAnotherCluster(sender)) { 23. **if** (parent != nil) { 24. parent.send(transaction, self); 25. } 26. } 28. **for** (Node child : children) { 29. **if** (child == sender) { 30. **continue**; 31. } 33. child.send(transaction, self); 34. } 36. **if** (interclusterLink != nil) { 37. interclusterLink.send(transaction, self); 38. } 39. } |

Важно је уочити да је у овом случају дисеминација навише потребна и када је пошиљалац из другог кластера. Ова провера врши се позивом функције ***isFromAnotherCluster(Node)***, која би на пример могла да провери да ли IP адреса пошиљаоца припада кластеру примаоца. Такође, нова локална променљива сервера је додата – ***interclusterLink***. Како су интеркластерни линкови унидирекциони, претпоставља се да ова променљива садржи неку вредност само за оне сервере који могу да шаљу поруке у други кластер.

Формирањем стабла унутар кластера у потпуности је елиминисана редундантност. Међутим, у целој мрежи редундантност и даље постоји. Разлог је тај што је граф мреже цикличан. Да би се ово решило, предлаже се коначно решење, које је заправо генерализација идеје да се користе стабла унутар кластера. Илустрација приказана је на слици 3.3.5.



Слика 3.3.5 – Решавање проблема редундантности у целој мрежи – коначно решење

* ***FN*** – Стандардни сервер (енг. *Full node*)
* ***V*** – Валидатор сервер (енг. *Validator node*)

Сада сви кластери у мрежи формирају стабло. У овом случају, чвор у стаблу може бити одговоран за више од једног интеркластерног линка – ово је мање рестриктивно од услова у [11]. Корен сваког стабла унутар кластера садржи интеркластерне линкове (IP адресе) према коренима стабала у кластерима који су деца његовог кластера. Порука се пропагира кроз мрежу на исти начин као што се пропагира унутар једног кластера.

Требало би приметити да рестрикција у [11] није морала да буде ублажена да би алгоритам исправно функционисао. Чвор у стаблу би и даље могао да остане одговоран за само један интеркластерни линк. Ово би довело до тога да већи број чворова унутар истог кластера садржи интеркластерне линкове ка чворовима у другим кластерима. Боље је везати интеркластерне линкове за само један чвор, и то корен. У наставку је дат псеудокод коначног решења када се користи алгоритам за дисеминацију заснован на кластеровању стабала:

Табела 3.3.3 – Кластеровање стабала – псеудокод

|  |
| --- |
| 1. // Pseudokod za FN/V server - koreni cvor 3. Node parent = ...; // uvek je iz drugog klastera 4. Set<Node> children = ...; 5. Set<Node> interclusterChildren = ...; 7. upon receive(T transaction, Node sender) { 8. **if** (sender == C) { 9. bool valid = checkTx(transaction); 10. **if** (valid == **false**) { 11. **return**; 12. } 13. } 15. addMempool(transaction); 17. **if** (sender == C || children.contains(sender) 18. || interclusterChildren.contains(sender)) { 19. **if** (parent != nil) { 20. parent.receive(transaction, self); 21. } 22. } 24. **for** (Node child : children) { 25. **if** (child == sender) { 26. **continue**; 27. } 29. child.send(transaction, self); 30. } 32. **for** (Node interclusterChild : interclusterChildren) { 33. **if** (interclusterChild == sender) { 34. **continue**; 35. } 37. interclusterChild.send(transaction, self); 38. } 39. } 41. // Pseudokod za FN/V server - unutrasnji cvor 43. Node parent = ...; // uvek je iz istog klastera 44. Set<Node> children = ...; 46. upon receive(T transaction, Node sender) { 47. **if** (sender == C) { 48. bool valid = checkTx(transaction); 49. **if** (valid == **false**) { 50. **return**; 51. } 52. } 54. addMempool(transaction); 56. **if** (sender == C || children.contains(sender)) { 57. // roditelj nikada nije nil za unutrasnji cvor 58. parent.send(transaction, self); 59. } 61. **for** (Node child : children) { 62. **if** (child == sender) { 63. **continue**; 64. } 66. child.send(transaction, self); 67. } 68. } |

У овом случају, позив функције ***checkMempool(T)*** такође је непотребан. Сваки FN сервер примиће трансакцију само једном.

* + 1. Анализа перформанси

Размотримо најпре предности класификације сервера у мрежи у кластере. Претпоставимо да је мрежа моделирана повезаним графом (који је у коначном решењу заправо стабло) који има ***e*** чворова и ***v*** грана. У тренутном решењу, број порука који се размени за једну клијентску трансакцију једнак је ***2v***. Како је у [11] математички доказано да кластеровањем смањујемо број грана, самим тим смо смањили број порука који се размени за једну клијентску трансакцију.

Даље, груписањем сервера у кластеру по неком критериујуму „близине“, постижемо то да су сервери мање „удаљени“ једни од других, и да стога могу да комуницирају једни са другима брже. Кластеровање смањује кашњење (енг. *latency*) у мрежи. Формирањем стабла унутар кластера, а и унутар целе мреже, број порука који се размени за једну клијентску трансакцију је оптималан и износи ***e-1***. Број редундантних порука које сваки сервер у мрежи прими једнак је 0.

* + 1. Дискусија решења

Алгоритам објашњен у овом потпоглављу не производи никакву редундантност. Међутим, у случају отказа овакво решење неће функционисати исправно. Уколико било који сервер у стаблу постане неисправан, клијентска трансакција неће доспети до свих сервера у мрежи, чиме потенцијално неће доћи до задовољавања услова <1> и <2>. Да би се ово решило, SMR сервери унутар сваког кластера би могли да формирају већи број стабала – шуму стабала у којој чвор који је лист у једном стаблу, није лист ни у једном од осталих стабала (разлог овога је оптимално искоришћење пропусног опсега сервера). Оваква идеја предложена је у [12], и заснована је на *SplitStream* алгоритму. Јасно је да решавање проблема отказа у мрежи захтева одређени ниво редундантности.

Што се тиче мана предложеног решења, прва се тиче кластера. Није лако уочити јасне критеријуме за класификацију сервера у кластере. Ово је предмет многих истраживања, како је наведено у [11]. Даље, имплементирање SMR скупа сервера који ће одржавати стабло динамички је изазован задатак. Одређивање консензус алгоритма који ће се користити у том скупу сервера неопходно је за исправно функционисање мреже.

С друге стране, уколико би се овакав алгоритам имплементирао, не би произвео никакву редундантност. Коришћењем кластеровања стабала у нашем моделу система, порука ће сигурно стићи до свих сервера у мрежи, и самим тим бити предложена од стране неког предлагача из скупа валидатора, чиме су задовољени услови <1> и <2>.

* 1. PPP HEAP PSS

Алгоритам приказан у овом потпоглављу базиран је на трећем приступу наведеном у уводном делу – динамичком повезивању сервера, чиме се формира насумични граф мреже. Алгоритам комбинује протокол за дисеминацију који се састоји из три фазе и који је описан у [13] (***PPP – Push-pull-push gossip protocol***), протокол који узима у обзир неједнака својства сервера у мрежи описан у [14] (***HEAP*** – ***HEterogeneity Aware Protocol***) и протокол за дисеминацију описан у [15] чија је улога да сваком серверу обезбеди информације о његовом подскупу суседа (***PSS – Peer Sampling Service***).

* + 1. Објашњење алгоритма

Дакле, предложени алгоритам је комбинација три различита протокола:

1. *Push-pull-push*
2. *HEAP*
3. *PSS*

Сваки од наведених протокола је додатно оптимизован ради бољих перформанси, и биће објашњен у засебним пододељцима. Уопштено, алгоритам приложен овде је базиран на динамичком креирању графа мреже. Сваки сервер може да мења свој подскуп суседа потпуно динамички. Клијентска трансакција биће дисеминирана серверима у подскупу суседа. Нека је величина подскупа суседа означена са ***f***. У даљем тексту, величина подскупа суседа називаће се ***распон чвора*** (енг. *node’s fanout*). У литератури се овај појам за величину подскупа суседа често среће ([13], [14], [15]).

* + - 1. **Push-pull-push**

Претпоставимо да постоји мрежа сервера, еквивалентна оној описаној у потпоглављу о моделу система. Нека свака трансакција има јединствени идентификатор који је цео број. Увешћемо додатну претпоставку – сваки сервер у мрежи мора да зна за све друге сервере у мрежи. Евидентно је да усвајањем ове претпоставке претпостављамо да постоји потпуна мрежна повезаност, што није случај у Тендерминту. Међутим, ова претпоставка је усвојена само зато да би објашњавање прва два протокола (*PPP*, *HEAP*) било једноставније. У трећем пододељку (*PSS*) биће објашњено како предложени алгоритам може да се примени на WAN мрежу где сваки сервер зна само за подскуп осталих.

Дакле, постоји мрежа сервера где сваки сервер зна IP адресу сваког другог сервера у мрежи. Основна идеја дисеминације је да сервер, периодично, шаље поруку униформно одабраном подскупу скупа свих сервера. Као што је напоменуто раније, величина тог подскупа назива се ***распон***. Теоретска [11] и експериментална [13] анализа су доказале да уколико желимо да граф мреже буде повезан са великом вероватноћом (односно уколико желимо велику вероватноћу да сви сервери добију поруку), оптимална вредност за ***f*** једнака је ***ln(n)***, где ***n*** представља број свих сервера у мрежи.

Протокол где се користи дисеминација која се састоји из три фазе је неопходан када постоји велико оптерећење мреже. Један разлог је тај што гарантује да ће порука бити достављена (додата у RAM меморију сервера) само једном. Други је тај што гарантује да сервер никада неће примити редундантну поруку у којој се налази читава трансакција. У овом протоколу, могуће је примити редундантну поруку у којој се налази једино идентификатор трансакције. Како је идентификатор трансакције далеко мањи него сама трансакција, овим се количина сувишног саобраћаја коју сервер прими умногоме смањује. Трећи разлог је тај **што се дисеминира једино идентификатор трансакције**. Потреба за слањем читаве трансакције јавља се само онда када је та трансакција била експлицитно захтевана (трансакција се захтева слањем њеног идентификатора).

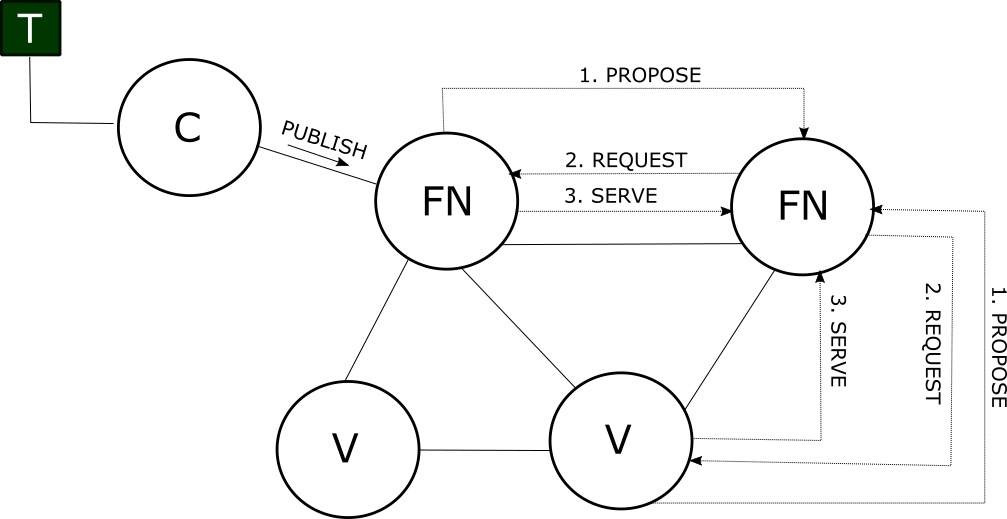
Три фазе које се користе у PPP протоколу су:

1. ***PUSH*** – представљена је слањем ***PROPOSE*** поруке
2. ***PULL*** – представљена је слањем ***REQUEST*** поруке
3. ***PUSH*** – представљена је слањем ***SERVE*** поруке

У наставку је дат псеудокод PPP протокола. Затим следи објашњење протокола.

Табела 3.4.1 – PPP протокол – псеудокод

|  |
| --- |
| 1. // Ovo se izvrsava na svakom FN serveru 3. // Inicijalizacija 4. **int** f = ln(n); 5. Set<**int**> toPropose = EMPTYSET; 6. Set<T> delivered = EMPTYSET; 7. Set<**int**> requested = EMPTYSET; 8. start(GossipTimer(gossipPeriod)); 10. // Faza 1 - PUSH T ids 11. upon (receive(PUBLISH, Transaction t))) { 12. // kada klijent posalje transakciju FN serveru 13. bool valid = checkTx(transaction); 14. **if** (valid == **false**) { 15. **return**; 16. } 18. deliverEvent(t); 19. requested.add(t.id); 20. toPropose.add(t.id); 21. } 23. upon (GossipTimer % gossipPeriod) == 0 { 24. gossip(toPropose); 25. toPropose = EMPTYSET; // Infect and die model 26. } 28. // Phase 2 - PULL wanted T ids 29. upon (receive(PROPOSE, proposed)) { 30. Set<**int**> wanted = EMPTYSET; 31. **for** (**int** id : proposed) { 32. **if** (!requested.contains(id)) { 33. wanted.add(id); 34. } 35. } 37. requested.add(wanted.getAll()); 38. reply(REQUEST, wanted); 39. } 41. // Phase 3 - PUSH requested T 42. upon (receive(REQUEST, wanted)) { 43. Set<T> asked = EMPTYSET; 44. **for** (**int** id : wanted) { 45. asked.add(getEvent(id)); 46. } 48. reply(SERVE, asked); 49. } 51. // poruka ce biti primljena samo jednom 52. upon (receive(SERVE, events)) { 53. **for** (T t : events) { 54. toPropose.add(t.id); 55. deliverEvent(t); 56. } 57. } 59. // Ostalo 60. function selectNodes(**int** f) Set<Node> { 61. **return** f uniformly random nodes from the set of all nodes; 62. } 64. function gossip(Set<**int**> eventIds) { 65. Set<Node> peerSubset = selectNodes(f); 66. **for** (Node node : peerSubset) { 67. node.send(PROPOSE, eventIds); 68. } 69. } 71. function getEvent(**int** id) T { 72. **return** T corresponding to the id; 73. } 75. function deliverEvent(T t) { 76. delivered.add(t); 77. addMempool(t); 78. } |



Слика 3.4.1 – Поруке које се размењују у току PPP протокола

* ***C*** - Клијент
* ***FN*** – Стандардни сервер (енг. *Full node*)
* ***V*** – Валидатор сервер (енг. *Validator node*)
* ***T*** – Клијентска трансакција

Протокол је заснован на размени три поруке између два сервера, као што је приказано на слици 3.4.1. Кад год сервер жели да пошаље информације о некомитованим трансакцијама које се налазе унутар његове RAM меморије, он шаље ***PROPOSE*** поруку (**ова порука садржи само идентификаторе трансакција**). Након тога, сви сервери који су примили ту поруку могу да одговоре ***REQUEST*** поруком (**ова порука садржи само идентификаторе трансакција**), да би добили трансакције чије су идентификаторе навели у *REQUEST* поруци. По пријему ***SERVE*** поруке (**ово је једина порука која заиста садржи трансакције, а не само њихове идентификаторе**), сервер може да дода примљене трансакције у своју RAM меморију.

Пошто смо претпоставили да један сервер зна за све друге сервере у мрежи, његов распон, ***f***, иницијализује се на вредност ***ln(n)***. Ово представља величину подскупа суседа, или прецизније, број сервера који ће примити *PROPOSE* поруку. Сервер води евиденцију о три скупа:

* + ***toPropose*** – скуп целих бројева који садржи идентификаторе трансакција; овај скуп периодично се дисеминира у *PROPOSE* поруци.
  + ***delivered*** – скуп трансакција који садржи све трансакције које су додате у RAM меморију сервера.
  + ***requested*** – скуп целих бројева који садржи идентификаторе трансакција које су захтеване *REQUEST* поруком.

Из разлога што се дисеминација обавља периодично, у фази иницијализације стартује се тајмер. Кад год прође одређени период времена (***gossipPeriod***), сервер шаље информације о свакој трансакцији коју до тада није предложио. Након тога, скуп *toPropose* бива испражњен. Овакав модел се у литератури о епидемичним протоколима назива ***Infect and die***, зато што сервер информацију шаље само једном, а потом је „уништава“.

Кад год клијент пошаље трансакцију FN серверу, долази до пријема ***PUBLISH*** поруке. По пријему ове поруке, прво се проверава валидност трансакције. Ако је трансакција валидна, она се додаје у RAM меморију сервера, а самим тим и у *delivered* скуп. Поред тога, додаје се у преостала два скупа. Разлог зашто се додаје у *toPropose* скуп јесте тај да би информација о њој била прослеђена у *PROPOSE* поруци при наредној дисеминацији. **Додавање трансакције у *requested* скуп је кључно, јер управо оно спречава да сервер прими редундантну поруку у којој се налази трансакција.**

Наиме, по пријему *PROPOSE* поруке, сервер одговара *REQUEST* поруком која садржи само оне трансакције које су му предложене, а које никад пре није тражио. Битно је уочити да сервер сваку трансакцију тражи само једном. Ово је могуће јер је претпоставка да је мрежа поуздана. Уколико сервер не би додао клијентску трансакцију у *requested* скуп када је прими, она би могла да му буде предложена *PROPOSE* поруком неког другог сервера, узевши у обзир да се трансакција шири кроз мрежу, а да граф мреже свакако може да садржи циклусе. По пријему *PROPOSE* поруке, сервер који је иницијално примио клијентску трансакцију не би је имао у свом *requested* скупу, тако да би додао и њен идентификатор у *REQUEST* поруку.

По пријему *SERVE* поруке, добио би, између осталих, и трансакцију коју већ има у својој RAM меморији. Из тог разлога, неопходно је да по пријему *PUBLISH* поруке, сервер дода трансакцију и у *requested* скуп. Ова модификација уведена је при оптимизацији PPP протокола, чија је оригинална верзија дата у [13].

По пријему *REQUEST* поруке, сервер одговара *SERVE* поруком која садржи захтеване трансакције. Трансакције се дохватају из RAM меморије позивом функције ***getEvent(int)***. По пријему *SERVE* поруке, сервер додаје трансакције у своју RAM меморију. Такође, додаје их у *toPropose* скуп. То значи да ће идентификатори додатих трансакција бити дисеминирани при следећем слању *PROPOSE* поруке.

PPP протокол смањује количину саобраћаја која пролази кроз мрежу, с обзиром на чињеницу да се дисеминира једино идентификатор трансакције. Трансакција се шаље тек када је експлицитно захтевана. Међутим, овај протокол има једну ману. Претпоставља да сваки сервер у мрежи има распон ***f*** = ***ln(n)***. То значи да је, при дисеминацији, сваки сервер једнако оптерећен. Нажалост, сви сервери у мрежи немају иста својства (нпр. пропусни опсег). Стога, наметање великог оптерећења на сервер који је слабији може да доведе до отказа, мањег протока и већег кашњења. HEAP, други протокол који је део предложеног алгоритма, има циљ да реши овај проблем.

* + - 1. **HEAP**

HEAP (*HEterogeneity Aware Protocol*) не претпоставља да је мрежа хомогене структуре. Штавише, овај протокол се ослања на то да сервери у мрежи имају различита својства, таква да су неки сервери бржи и продуктивнији од других. HEAP прилагођава распон чвора узевши у обзир његов пропусни опсег, просечан пропусни опсег и просечан распон чворова у мрежи. У случају HEAP, просечан распон чворова у мрежи је ***ln(n)***, где је ***n*** број свих сервера у мрежи.

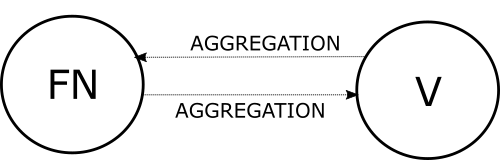
Једначина која се користи у HEAP [14] је:

*f = b* / *\_b\_* \* *ln(n)*,

где је ***f*** распон чвора, ***b*** је пропусни опсег сервера, а ***\_b\_*** је просечан пропусни опсег у мрежи. Просечан пропусни опсег ажурира се локално на сваком серверу, зависно од информација о својствима других сервера које се у HEAP протоколу такође дисеминирају кроз мрежу. Ажуриран псеудокод, који укључује PPP и HEAP, дат је у даљем тексту, праћен објашњењем.

Табела 3.4.2 – HEAP протокол – псеудокод

|  |
| --- |
| 1. // Ovo se izvrsava na svakom FN serveru 3. // Inicijalizacija 4. Set<Capability> capabilities = EMPTYSET; 5. Bandwidth b = MY\_BANDWIDTH; 6. Bandwith \_b\_ = ...; // average bandwidth 8. **int** f = ln(n); // average fanout 9. Set<**int**> toPropose = EMPTYSET; 10. Set<T> delivered = EMPTYSET; 11. Set<**int**> requested = EMPTYSET; 12. start(GossipTimer(gossipPeriod)); 14. start(AggregationTimer(aggregationPeriod)); 16. // Phase 1 - PUSH T ids 17. upon (receive(PUBLISH, Transaction t))) { 18. // kada klijent posalje transakciju FN serveru 19. bool valid = checkTx(transaction); 20. **if** (valid == **false**) { 21. **return**; 22. } 24. deliverEvent(t); 25. requested.add(t.id); 26. toPropose.add(t.id); 27. } 29. upon (GossipTimer % gossipPeriod) == 0 { 30. gossip(toPropose); 31. toPropose = EMPTYSET; // Infect and die model 32. } 34. // Phase 2 - PULL wanted T ids 35. upon (receive(PROPOSE, proposed)) { 36. Set<**int**> wanted = EMPTYSET; 37. **for** (**int** id : proposed) { 38. **if** (!requested.contains(id)) { 39. wanted.add(id); 40. } 41. } 43. requested.add(wanted.getAll()); 44. reply(REQUEST, wanted); 45. } 47. // Phase 3 - PUSH requested T 48. upon (receive(REQUEST, wanted)) { 49. Set<T> asked = EMPTYSET; 50. **for** (**int** id : wanted) { 51. asked.add(getEvent(id)); 52. } 54. reply(SERVE, asked); 55. } 57. // poruka ce biti primljena samo jednom 58. upon (receive(SERVE, events)) { 59. **for** (T t : events) { 60. toPropose.add(t.id); 61. deliverEvent(t); 62. } 63. } 65. // Aggregation protocol 66. upon (AggregationTimer % aggregationPeriod) == 0 { 67. Set<Node> peerSubset = selectNodes(getFanout()); 68. **for** (Node node : peerSubset) { 69. Set<Capability> fresh = capabilities.get(K); 70. node.send(AGGREGATION, fresh); 71. } 72. } 74. upon (receive(AGGREGATION, newCapabilities)) { 75. capabilities.merge(newCapabilities); 76. update(\_b\_, capabilities); 77. } 79. // Fanout adaptation 80. function getFanout() **int** { 81. **return** b / \_b\_ \* f; 82. } 84. // Ostalo 85. function selectNodes(**int** f) Set<Node> { 86. **return** f uniformly random nodes from the set of all nodes; 87. } 89. function gossip(Set<**int**> eventIds) { 90. Set<Node> peerSubset = selectNodes(getFanout()); 91. **for** (Node node : peerSubset) { 92. node.send(PROPOSE, eventIds); 93. } 94. } 96. function getEvent(**int** id) T { 97. **return** T corresponding to the id; 98. } 100. function deliverEvent(T t) { 101. delivered.add(t); 102. addMempool(t); 103. } |



Слика 3.4.2 – Додатне поруке које се размењују у HEAP протоколу

* ***FN*** – Стандардни сервер (енг. *Full node*)
* ***V*** – Валидатор сервер (енг. *Validator node*)

Сваки FN сервер сада има информацију о скупу својстава – ***Set<Capability>***. Овај скуп се користи да би се ажурирала вредност просечног пропусног опсега – ***update(Bandwidth, Set<Capability>)***. Принцип ажурирања просечног пропусног опсега зависи од имплементације протокола. Такође, подаци које садржи ***Set<Capability>*** зависе од имплементације протокола. То могу бити било који подаци који валидно осликавају својства сервера у мрежи (нпр. њихови пропусни опсези).

У HEAP протоколу, осим размене *PROPOSE, REQUEST* и *SERVE* порука, сервери у мрежи периодично размењују и ***AGGREGATION*** поруке, као што је илустровано на слици 3.4.2. Стога, у току фазе иницијализације, стартује се посебан тајмер – ***Aggregation timer***. Кад год истекне период времена ***aggregationPeriod***, FN сервер дисеминираће информације из свог ***Set<Capability>*** униформно одабраном подскупу скупа свих сервера. Сада се распон чвора рачуна по малопре поменутој једначини, позивом функције ***getFanout()***. Када се дисеминира информација о својствима из локалног скупа својстава, сервер ће одабрати ***K*** вредности из тог скупа. Вредност за ***K*** зависи од имплементације протокола. По пријему *AGGREGATION* поруке, FN сервер спојиће добијене вредности својстава са оним које већ има код себе, а затим ће ажурирати просечан пропусни опсег по неком принципу.

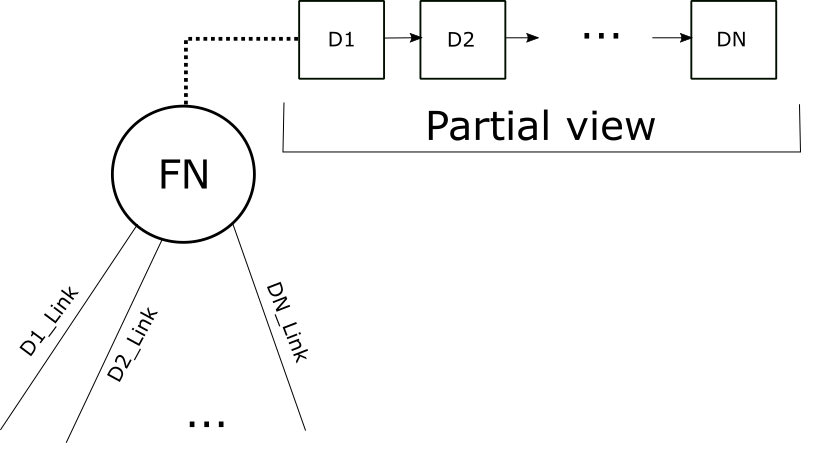
Главна улога HEAP протокола јесте да оптимизује искоришћење својстава сервера (нпр. пропусног опсега), тако што ће прилагодити распон тог сервера у складу са поменутим параметрима (*b, \_b\_, ln(n)*). Стога, сервери у мрежи периодично дисеминирају информације о својствима других сервера преко *AGGREGATION* порука. Ово доводи до тога да сви сервери у мрежи „уче“ о потенцијалу осталих, и да зависно од тога рачунају просечан пропусни опсег мреже. Вредност просечног пропусног опсега користи се када се рачуна распон чвора, при дисеминацији информација његовом подскупу суседа.

* + - 1. **PSS**

На самом крају, описаћемо трећи протокол који се користи у предложеном алгоритму. Овај протокол омогућава примену алгоритма у WAN мрежи. До сада, претпоставка је била да сваки FN сервер мора да зна за све друге сервере у мрежи. Да би овај проблем био превазиђен, у даљем тексту биће описан PSS (*Peer Sampling Service*) [15].

Суштинска сврха PSS јесте да сваком серверу обезбеди информације о његовом *подскупу суседа.* До овог дела, FN сервер је при дисеминацији униформно бирао онолико сервера из скупа свих сервера колика је била вредност његовог распона, притом имплицитно стварајући свој подскуп суседа. Међутим, вођење евиденције о свим серверима у мрежи је непрактично када се ради о великим дистрибуираним системима.

Идеја PSS јесте да се, као што се подаци дисеминирају кроз мрежу, дисеминирају и информације о серверима у мрежи. Стога, PSS је врста протокола за дисеминацију. Он је заслужан за динамичко креирање насумичног графа мреже. Експериментална анализа у [15] даје обећавајуће резултате када је у питању PSS.



Слика 3.4.3 – Структура сервера у PSS протоколу

* ***FN*** – Стандардни сервер (енг. *Full node*)
* ***D*** – Дескриптор чвора

Као што је приказано на слици 3.4.3 и описано у [15], сваки FN сервер садржи информацију о свом подскупу суседа, који је у [15] означен као ***парцијални поглед*** ***чвора*** (енг. *node’s partial view*). Парцијални поглед представљен је листом ***дескриптора чворова***. Сваки дескриптор садржи *IP адресу суседа* и *број мрежних хопова до тог суседа*. Парцијални поглед садржи највише један дескриптор за једну IP адресу, и сортиран је неопадајуће по броју хопова. Максимална величина парцијалног погледа је ограничена. Нека је та величина означена са ***c***.

PSS протокол састоји се из две нити:

* **Активна нит** – иницира комуникацију са осталим серверима
* **Пасивна нит** – прима долазеће поруке

Псеудокод PSS протокола дат је у наставку. Псеудокод је модификован у односу на онај из [15]. Псеудокод биће праћен објашњењем.

Табела 3.4.3 – PSS протокол – псеудокод

|  |
| --- |
| 1. // Peer Sampling Service - pseudokod 2. bool push = ...; // najmanje jedan uvek ima vrednost true 3. bool pull = ...; 4. PartialView myPartialView; 6. // Aktivna nit - kod koji se izvrsava 7. **while** (**true**) { 8. wait(SOMETIME); 9. Node p = selectPeer(); 11. **if** (push) { 12. // 0 je inicijalni broj hopova 13. Descriptor myDescriptor = **new** Descriptor(myIPAddress, 0); 14. PartialView buffer = merge(myPartialView, myDescriptor); 16. send(p, buffer); 17. } 19. **if** (pull) { 20. // pravi se pull zahtev 21. send(p, EMPTYSET); 23. PartialView receivedView = receiveViewFrom(p); 25. increaseHopCount(receivedView); 26. PartialView buffer = merge(receivedView, myPartialView); 27. myPartialView = selectView(buffer); 28. } 29. } 31. // Passivna nit - kod koji se izvrsava 33. **while** (**true**) { 34. {Node p, PartialView receivedView} = receiveMessage(); 36. // aktivna nit nekog drugog servera napravila je pull zahtev 37. **if** (receivedView == EMPTYSET) { 38. // 0 je inicijalni broj hopova 39. Descriptor myDescriptor = **new** Descriptor(myIPAddress, 0); 40. PartialView buffer = merge(myPartialView, myDescriptor); 42. send(p, buffer); 43. } **else** { 44. increaseHopCount(receivedView); 45. PartialView buffer = merge(receivedView, myPartialView); 46. myPartialView = selectView(buffer); 47. } 48. } |

PSS протокол параметризован је са две логичке променљиве – ***push*** и ***pull***. Такође, параметризован је и двема функцијама – ***selectPeer()***  и ***selectView(PartialView)***. Различите стратегије за одређивање ових параметара дате су у [15] – стр. 5. Функција ***selectPeer()*** бира једног суседа са којим ће сервер комуницирати, из његовог парцијалног погледа. Функција ***selectView(PartialView)*** ограничава бафер ***buffer***, тако да буде максималне дужине ***c***. Функција ***merge(PartialView, PartialView)*** спаја два погледа које прима као параметре. Резултујући поглед је поново сортиран по броју хопова. Уколико у унији два погледа који су параметри функције постоје два дескриптора са истом IP адресом, узима се онај са мањим бројем хопова. Функција ***increaseHopCount(PartialView)*** инкрементира вредност за број хопова у сваком дескриптору погледа који је параметар функције.

Активна нит периодично иницира комуникацију са одабраним суседом. Комуникација се иницира у зависности од стратегије – *push, pull* или *pushpull* [15]. У случају да логичка променљива *push* има вредност *true*, активна нит шаље свој парцијални поглед и сопствени дескриптор пасивној нити одабраног суседа. У случају да логичка променљива *pull* има вредност *true*, активна нит направиће ***pull захтев*** (енг. ***pull request***) тако што ће да пошаље посебну вредност (нпр. празан скуп) пасивној нити одабраног суседа. Након тога чекаће одговор од суседа.

Пасивна нит служи да прима информације и одговара на *pull захтеве* активних нити других сервера. У случају да није послата посебна вредност од стране активне нити неког сервера, пасивна нит ажурира свој парцијални поглед у складу са одабраном стратегијом. У случају да је послата посебна вредност од стране активне нити неког сервера, пасивна нит одговориће поруком у којој се налази парцијални поглед њеног сервера и његов дескриптор.

Може се закључити да коришћењем PSS протокола избегавамо потребу за тим да сваки сервер зна за све друге сервере у мрежи. Сервери ће периодично размењивати информације о њиховим парцијалним погледима и ажурираће сопствене парцијалне погледе, у складу са одабраним стратегијама.

* + - 1. **Коначно решење**

Предложени алгоритам комбинује сва три претходно описана протокола. Користи PPP тако да се дисеминира једино идентификатор трансакције. Користи HEAP да оптимизује распон сваког чвора, а самим тим максимално искористи његове способности. Користи PSS да сваком чвору достави информације о његовом подскупу суседа, чиме динамички мења граф мреже. Псеудокод коначног решења дат је у даљем тексту.

Табела 3.4.4 – Псеудокод коначног решења

|  |
| --- |
| 1. // Ovo se izvrsava na svakom FN serveru 3. // Inicijalizacija 4. Set<Capability> capabilities = EMPTYSET; 5. Bandwidth b = MY\_BANDWIDTH; 6. Bandwith \_b\_ = ...; // average bandwidth 8. PartialView myPartialView; 9. initPSS(); 11. **int** f = ln(n); // average fanout 12. Set<**int**> toPropose = EMPTYSET; 13. Set<T> delivered = EMPTYSET; 14. Set<**int**> requested = EMPTYSET; 15. start(GossipTimer(gossipPeriod)); 17. start(AggregationTimer(aggregationPeriod)); 19. // Phase 1 - PUSH T ids 20. upon (receive(PUBLISH, Transaction t))) { 21. // kada klijent posalje transakciju FN serveru 22. bool valid = checkTx(transaction); 23. **if** (valid == **false**) { 24. **return**; 25. } 27. deliverEvent(t); 28. requested.add(t.id); 29. toPropose.add(t.id); 30. } 32. upon (GossipTimer % gossipPeriod) == 0 { 33. gossip(toPropose); 34. toPropose = EMPTYSET; // Infect and die model 35. } 37. // Phase 2 - PULL wanted T ids 38. upon (receive(PROPOSE, proposed)) { 39. Set<**int**> wanted = EMPTYSET; 40. **for** (**int** id : proposed) { 41. **if** (!requested.contains(id)) { 42. wanted.add(id); 43. } 44. } 46. requested.add(wanted.getAll()); 47. reply(REQUEST, wanted); 48. } 50. // Phase 3 - PUSH requested T 51. upon (receive(REQUEST, wanted)) { 52. Set<T> asked = EMPTYSET; 53. **for** (**int** id : wanted) { 54. asked.add(getEvent(id)); 55. } 57. reply(SERVE, asked); 58. } 60. // poruka ce biti primljena samo jednom 61. upon (receive(SERVE, events)) { 62. **for** (T t : events) { 63. toPropose.add(t.id); 64. deliverEvent(t); 65. } 66. } 68. // Aggregation protocol 69. upon (AggregationTimer % aggregationPeriod) == 0 { 70. Set<Node> peerSubset = selectNodes(getFanout()); 71. **for** (Node node : peerSubset) { 72. Set<Capability> fresh = capabilities.get(K); 73. node.send(AGGREGATION, fresh); 74. } 75. } 77. upon (receive(AGGREGATION, newCapabilities)) { 78. capabilities.merge(newCapabilities); 79. update(\_b\_, capabilities); 80. } 82. // Fanout adaptation 83. function getFanout() **int** { 84. **return** b / \_b\_ \* f; 85. } 87. // Ostalo 88. function selectNodes(**int** f) Set<Node> { 89. **if** (myPartialView.size() > f) { 90. **return** myPartialView.getNodes(f); 91. } **else** { 92. **return** myPartialView.getAllNodes(); 93. } 94. } 96. function gossip(Set<**int**> eventIds) { 97. Set<Node> peerSubset = selectNodes(getFanout()); 98. **for** (Node node : peerSubset) { 99. node.send(PROPOSE, eventIds); 100. } 101. } 103. function getEvent(**int** id) T { 104. **return** T corresponding to the id; 105. } 107. function deliverEvent(T t) { 108. delivered.add(t); 109. addMempool(t); 110. } |

FN сервер сада садржи информације о свом подскупу суседа унутар локалне променљиве ***myPartialView***. Ову променљиву ажурира PSS протокол који такође ради у позадини, на сваком FN серверу. Функција ***initPSS()*** иницијализује PSS протокол на FN серверу. Различите технике иницијализације овог протокола наведене су у [15].

**Важно је приметити да се мења једино имплементација функције *selectNodes(int)***. Ова функција враћа подскуп суседа FN сервера. Том подскупу ће бити дисеминирана одговарајућа порука. Подскуп суседа се у овом случају бира у складу са оптималним распоном који је израчунат захваљујући HEAP протоколу. Ако оптимални број суседа израчунат од стране HEAP протокола означимо са ***f***, функција ***selectNodes(int)*** ће увек одабрати највише ***f*** суседа из парцијалног погледа сервера. Циљ овога је да се оптимизује искоришћење сервера и да се он не преоптерети. Алтернатива би била да се порука дисеминира свима у парцијалном погледу, чак и у случају да их има више од ***f***. Требало би уочити да сада нема потребе да један FN сервер зна за IP адресе свих других у мрежи. Довољно је да зна само број сервера у мрежи, ***n***, да би могао да израчуна ***ln(n)***.

* + 1. Анализа перформанси

Размотрићемо број порука које се размене за сваку клијентску трансакцију код PPP протокола. Нека је мрежа моделирана графом који има ***e*** чворова. Уколико узмемо у обзир само оне поруке у којима се налази трансакција, тада ће за једну клијентску трансакцију бити размењено укупно ***e-1*** порука, што је оптимално. Разлог овоме је што сваки сервер шаље *SERVE* поруку у којој се налазе само оне трансакције које су експлицитно захтеване. Трансакције се захтевају само једном. Из овога следи да је и број редундантних порука у којима се налази трансакција, а које сваки сервер прими, једнак 0, што је такође оптимално.

Међутим, редундантност постоји у пријему *PROPOSE* порука. Сервер може да прими поруку од једног суседа у којој се налази идентификатор трансакције коју је он претходно захтевао од неког другог суседа. С друге стране, редундантност је у овим случајевима далеко је мања, како показују резултати симулације.

* + 1. Дискусија решења

Главни недостатак овог приступа је тај што је пробабилистички. Ослања се на теоретску и експерименталну анализу која тврди да коришћењем вредности ***ln(n)*** за просечни распон чворова у мрежи, граф остаје повезан са високом вероватноћом. Ова вероватноћа асимптотски тежи 1. У нашем моделу система минимална величина подскупа суседа је 1 и граф је увек повезан, тако да, теоретски гледано, и овај алгоритам ће увек дисеминирати трансакцију до свих сервера у мрежи, чиме ће задовољити услове <1> и <2>.

У случају отказа, била би потребна нека врста ретрансмисије. Идеја за ово постоји у [14]. Предложено је коришћење посебног тајмера, ***RetTimer***. Принцип почива на томе да FN сервер изнова и изнова шаље *REQUEST* поруке за одређену групу трансакција, све док их не добије у *SERVE* поруци.

Постоји много предности алгоритма предложеног у овом потпоглављу. Алгоритам је потпуно динамички, и сваки његов концепт базира се на епидемичним протоколима. Мрежа не бива преплављена трансакцијама, већ само њиховим идентификаторима. Трансакција се шаље само онда када је експлицитно тражена. Овај алгоритам је могуће имплементирати у реалним система, јер у њему не постоје концепти које би требало додатно истраживати, као што је то био случај у прва два предложена алгоритма.

1. Симулациона анализа алгоритама

У овом поглављу биће детаљно објашњена симулациона анализа два алгоритма за дисеминацију из поглавља 3. – тренутног решења које се користи у Тендерминту и PPP протокола. Циљ ове анализе јесте да упореди та два алгоритма на основу три различите метрике:

* + Процента редундантних порука (енг. *overhead*) које сваки сервер у мрежи прими.
  + Просечног кашњења при дисеминацији трансакције.
  + Просечног максималног броја хопова при дисеминацији трансакције.

Симулациона анализа врши се баш за ова два алгоритма из разлога што PPP протокол може бити реално имплементиран у Тендерминту.

* 1. Поставка симулатора

Симулатор је представљен конкурентном апликацијом чији се програмски код, написан у језику JAVA, може наћи у [21]. Како се ради о конкурентној апликацији, била је потребна синхронизација различитих нити. За сву потребну синхронизацију коришћени су семафори из пакета ***java.util.concurrent***. Симулатор симулира мрежу која се састоји од једног клијента и више сервера. Битно је приметити да број клијената у симулацији није битан, јер је акценат на дисеминацији трансакција које сервери добију, а не на томе ко је те трансакције послао.

Клијент шаље фиксан број трансакција у мрежу. Те трансакције бивају дисеминиране кроз мрежу користећи неки од два алгоритма (PPP протокол или тренутно решење). Симулација се завршава онда када сви сервери у мрежи приме све трансакције у своју RAM меморију.

* + 1. Сервери у симулацији

Најпре, објаснићемо принцип функционисања сервера у симулацији. Сервер је представљен апстрактном класом ***Server***. Сваки сервер састоји се од две нити (ове нити представљене су двема унутрашњим класама класе ***Server***):

* + **Активна нит –** периодично дисеминира информације осталим серверима у подскупу суседа.
  + **Пасивна нит** – обрађује приспеле захтеве све док сви сервери у мрежи не додају све клијентске трансакције у своје RAM меморије.

Активна нит периодично, са периодом (која је у конкретним симулацијама имала вредност од 10*ms*), позива апстрактну методу ***gossip()***. Ова метода биће имплементирана у поткласама класе ***Server*** – ***CurrentSolutionServer*** и ***PushPullPushServer***. Имплементација ове методе одређује начин на који сервер дисеминира клијентске трансакције. Активна нит престаје са радом онда када сервер прими све клијентске трансакције у своју RAM меморију – у том тренутку се логичка променљива ***activeThreadFinished*** поставља на вредност *true*. Програмски код активне нити дат је у наставку текста.

Табела 4.1.1 – Активна нит сервера

|  |
| --- |
| 1. **private** **class** ActiveThread **extends** Thread { 3. @Override 4. **public** **void** run() { // Aktivna nit periodicno diseminira informacije 5. **while** (!activeThreadFinished) { 6. **try** { 7. Thread.sleep(Constants.GOSSIP\_PERIOD); 8. } **catch** (Exception e) { 9. e.printStackTrace(); 10. } 12. **if** (activeThreadFinished) { 13. **return**; 14. } 16. gossip(); 17. } 18. } 20. } |

Пасивна нит се врти у бесконачној петљи. На почетку петље, она чека на ***requestsEmpty*** семафору. Овај семафор има иницијалну вредност 0. Када год клијент или неки сусед жели да пошаље поруку одређеном серверу, позваће се метода ***send(Object)*** тог сервера. Ова метода прима захтев који је енкапсулиран унутар JAVA класе ***Object.*** У оквиру ове методе, захтев се додаје у посебан бафер, ***requestBuffer***, и сигнализира се пасивној нити тог сервера на ***requestsEmpty*** семафору, како би она могла да преузме управо додат захтев. Дакле, пасивна нит, након што је успела да прође семафор, узима захтев из бафера и обрађује га позивом методе ***processRequest(Object)***. Ова метода биће имплементирана у поткласама класе ***Server*** – ***CurrentSolutionServer*** и ***PushPullPushServer***. Имплементација ове методе одређује начин на који сервер обрађује приспеле трансакције. Након обраде трансакције, проверава се да ли је примљена и последња клијентска трансакција. Уколико јесте, приспела информација дисеминира се по последњи пут (*Infect and die* модел), зауставља се активна нит тог сервера и сигнализира на семафору ***toSignal***. На овом семафору чека главна нит програма. Иницијална вредност овог семафора је таква да ће главна нит моћи да га прође тек када све пасивне нити свих сервера сигнализирају на њему. Практично, пролазак главне нити на овом семафору је знак да је симулација завршена. Након што главна нит прође овај семафор, поставиће логичку променљиву ***passiveThreadFinished*** свих пасивних нити на вредност *true*, и још једном сигнализирати на ***requestsEmpty*** семафору свих пасивних нити. На овај начин завршиће се и све пасивне нити у симулацији. Програмски код пасивне нити сервера, као и методе ***send(Object)***, дат је у наставку текста.

Табела 4.1.2 – пасивна нит сервера

|  |
| --- |
| 1. **private** **class** PassiveThread **extends** Thread { 3. @Override 4. **public** **void** run() { 5. **while** (**true**) { 6. requestsEmpty.acquireUninterruptibly(); 8. **if** (passiveThreadFinished) { 9. **return**; 10. } 12. requestBufferSemaphore.acquireUninterruptibly(); 13. Object request = requestBuffer.remove(0); 14. requestBufferSemaphore.release(); 16. processRequest(request); 18. **if** (mempool.size() == Constants.NUM\_OF\_TRANSACTIONS && !activeThreadFinished) { 19. // diseminacija po poslednji put, da bi se prosledile novodobijene informacije 20. // sve sto se primi od sad na dalje bice redundantno 21. gossip(); 23. activeThreadFinished = **true**; 24. toSignal.release(); 25. } 26. } 27. } 28. } 30. //... 32. **protected** **void** send(Object request) { 33. requestBufferSemaphore.acquireUninterruptibly(); 34. requestBuffer.add(request); 35. requestBufferSemaphore.release(); 37. requestsEmpty.release(); 38. } |

У даљем тексту размотриће се имплементација две поткласе класе ***Server*** – ***CurrentSolutionServer*** и ***PushPullPushServer***. ***CurrentSolutionServer*** је класа која представља сервер чији алгоритам за дисеминацију функционише као тренутни алгоритам за дисеминацију у Тендерминту. Имплементацијa ***gossip()*** методе обухвата слање свих трансакција из RAM меморије сервера, које још увек нису дисеминиране, свима у подскупу суседа, позивом њихове методе ***send(Object)***. Пошто се сервер састоји из две нити, активне и пасивне, и како пасивна нит прима захтеве а активна их дисеминира, у имплементацији овог типа сервера додат је још један бафер – ***toGossip***. У овом баферу налазе се трансакције које се дисеминирају свима у подскупу суседа, по *Infect and die*принципу. Пасивна нит додаје трансакције у њега, а активна нит дисеминира транскције из њега. Имплементација ***processRequest(Object)*** методе практично је еквивалентна псеудокоду тренутног решења који је дат у потпоглављу 3.1. ***PushPullPushServer*** је класа која представља сервер чији је алгоритам за дисеминацију PPP протокол. Имплементација ове класе је заснована на псеудокоду PPP протокола који је дат у потпоглављу 3.4. Имплементација ***gossip()*** методе обухвата дисеминацију ***toPropose*** скупа. Имплементација ***processRequest(Object)*** методе је еквивалентна делу псеудокода PPP протокола који одговара пријему *PUBLISH*, *PROPOSE*, *REQUEST* и *SERVE* порука.

* + 1. Клијент у симулацији

Да би се извршила једна симулација, позива се метода ***simulate(int, int, SimulationStrategy)***. ***SimulationStrategy*** је тип набрајања који може имати две вредности – ***CURRENT\_SOLUTION*** и ***PUSH\_PULL\_PUSH.*** Вредност трећег параметра методе ***simulate(int, int, SimulationStrategy)*** одређује да ли се врши симулација мреже у којој се за дисеминацију користи PPP протокол, или тренутно решење Тендерминта. Први параметар методе ***simulate(int, int, SimulationStrategy)*** представља број сервера за који се покреће симулација. Други параметар ове методе представља број трансакција које ће кружити мрежом.

Унутар ове методе најпре се креира семафор на коме ће, по пријему свих трансакција у своју RAM меморију, сигнализирати пасивна нит неког сервера. Затим се рачуна просечна величина подскупа суседа. Наиме, симулације су покретане за три различите просечне величине подскупа суседа – ***ln(n), 2ln(n), 3ln(n)***, где је ***n*** број свих сервера у мрежи у посматраној симулацији. Разлог за то је да би се уочила промена метрика при повећању просечног распона чвора, односно просечне величине подскупа суседа сервера.

Након овога, позива се метода ***initServers(int, SimulationStrategy, Semaphore, int)***. Циљ ове методе јесте да креира све сервере у мрежи, као и њихове подскупове суседа. Подскуп суседа се креира тако да је граф мреже увек повезан, односно да му је минимална величина 1. Просечна величина подскупа суседа одређена је четвртим параметром ове функције.

Клијент у симулацији представљен је главном нити програма. Пре покретања свих сервера (или прецизније, позива ***start()*** метода њихових активних и пасивних нити), главна нит програма позваће методу ***sendTransactions(ArrayList<Server>, int)***. У овој методи ће се послати фиксан број трансакција (овај број прослеђује се као други параметар функције) насумично одабраним серверима у мрежи.

Након што је проследио трансакције у мрежу, клијент чека на семафору ***toSignal*** да се симулација заврши. Када се то догоди, заустављају се пасивне нити свих сервера. Повратна вредност методе ***simulate(int, int, SimulationStrategy)*** је листа свих сервера у посматраној симулацији. Разлог је тај што се, након завршене симулације, од сваког сервера могу добити информације о одговарајућој метрици израчунатој у току симулације. Програмски код функције ***simulate(int, int, SimulationStrategy)*** дат је у наставку.

Табела 4.1.3 – Програмски код клијента

|  |
| --- |
| 1. // funkcija izvrsava jednu simulaciju 2. **private** ArrayList<Server> simulate(**int** numberOfServers, **int** numberOfTransactions, SimulationStrategy simulationStrategy) { 3. Semaphore toSignal = **new** Semaphore(-numberOfServers + 1, **true**); 4. **int** averagePeerSubsetSize = Constants.PEER\_SUBSET\_MULTIPLIER \* ((**int**) Math.log(numberOfServers)); 6. **if** (averagePeerSubsetSize <= 0) { 7. averagePeerSubsetSize = 1; 8. } 10. ArrayList<Server> servers = initServers(numberOfServers, simulationStrategy, toSignal, averagePeerSubsetSize); 12. sendTransactions(servers, numberOfTransactions); 14. startServers(servers); 16. toSignal.acquireUninterruptibly(); // ceka se da svi serveri prime sve transakcije u svoju RAM memoriju 18. **for** (Server server : servers) { 19. server.finishPassiveThread(); 20. } 22. **return** servers; 23. } |

* + 1. Одређивање метрика у симулацији

Пре него што резултати симулације буду приказани, биће објашњено на који начин су рачунате три метрике поменуте на почетку овог поглавља. У симулацији, свака трансакција описана је објектом класе ***Transaction***. Свака трансакција има јединствен целобројни идентификатор. Кад год се трансакција додаје у RAM меморију неког сервера, прави се копија одговарајућег објекта.

* + - 1. **Одређивање процента редундантности**

Сваки сервер има локалну променљиву, ***numOfRedundantTransactionsReceived***, чија је иницијална вредност 0. Уколико се ради о серверу чији је алгоритам за дисеминацију PPP протокол, онда ова променљива бива инкрементирана за сваки редундантни идентификатор примљен у *PROPOSE* поруци. Уколико се ради о серверу чији је алгоритам за дисеминацију онај који се тренутно користи у Тендерминту, тада ова променљива бива инкрементирана када сервер добије трансакцију која је већ унутар његове RAM меморије. Делови кода који ажурирају ову променљиву, као и начин рачунања процента редундантности, дати су у наставку.

Табела 4.1.4 – Одређивање процента редундантности

|  |
| --- |
| 1. // Trenutno resenje 2. **if** (isInMyMempol) { 3. numOfRedundantTransactionsReceived++; 5. **return**; 6. } 8. // PPP protokol 9. **for** (Integer id : proposed) { 10. **if** (!requested.contains(id)) { 11. wanted.add(id); 12. requested.add(id); 13. } **else** { 14. numOfRedundantTransactionsReceived++; 15. } 16. } 18. // Racunanje procenta redundantnosti 19. **public** **double** getOverhead() { 20. **if** (**this** **instanceof** CurrentSolutionServer) { 21. **return** (numOfRedundantTransactionsReceived \* 100f) / (numOfRedundantTransactionsReceived + mempool.size()); 22. } **else** { 23. **int** divisor = Constants.AVERAGE\_TRANSACTION\_SIZE\_IN\_BYTES / Constants.IDENTIFIER\_SIZE\_IN\_BYTES; 24. **return** (((**double**) numOfRedundantTransactionsReceived / divisor) \* 100f) / (((**double**) numOfRedundantTransactionsReceived / divisor) + mempool.size()); 25. } 26. } |

Проценат редундантности рачуна се на нивоу броја редундантних **трансакција** које су примљене, а не на нивоу броја редундантних **идентификатора** који су примљени. Просечна величина трансакције дефинисана је одговарајућом константом (и износи око 550 бајтова). Информација о просечној величини трансакције у *Bitcoin* мрежи може се наћи у [20]. Просечна величина трансакције у *Bitcoin* мрежи одговара просечној величини трансакције у Тендерминту. Величина идентификатора јесте 32 бајта (идентификатор се добија рачунањем *SHA256* хеш вредности трансакције). Дакле, отприлике на сваких 17 идентификатора долази једна трансакција. Сразмерно томе, рачуна се проценат редундантности, у зависности од тога да ли се ради о тренутном решењу или PPP протоколу.

* + - 1. **Одређивање просечног кашњења**

Да би се валидно осликала реална мрежа, у симулацију се уводе два типа линкова између сервера – LAN и WAN линкови. Ови линкови представљени су класом ***Link*.** Стога, подскуп суседа неког сервера дефинисан је као ***ArrayList<Link>***. При генерисању подскупа суседа, у оквиру функције ***initServers(int, SimulationStrategy, Semaphore, int)***, генеришу се и одговарајући линкови. У 80% случајева, линк између два сервера је WAN линк, у осталих 20% случајева линк између два сервера је LAN линк. Линк између клијента и сервера је увек LAN.

У вези са различитим типовима линкова који могу да постоје у мрежи, уводе се и различита кашњења. Кашњење представља време у *ms* које је потребно да трансакција стигне са једног на други крај линка. На WAN линковима, кашњење се генерише као насумичан број у опсегу [100*ms*, 200*ms*]. На LAN линковима, кашњење се генерише као насумичан број у опсегу [10*ms*, 30*ms*]. Кашњење се генерише при дисеминацији трансакције, и представљено је атрибутом који је додат уз трансакцију – ***delay***. Када се ради о PPP протоколу, при генерисању кашњења, претпоставља се да је за поруке у којима је идентификатор, потребно мање времена за пренос (*PROPOSE, REQUEST*) у односу на оне поруке у којима се налази цела трансакција (*SERVE*). Начин генерисања кашњења дат је у наставку.

Табела 4.1.5 – Генерисање и одређивање просечног кашњења

|  |
| --- |
| 1. // Generisanje kasnjenja u trenutnom resenju 2. **int** delay = peer.getLinkType().equals(Link.LinkType.WAN) ? ThreadLocalRandom.current().nextInt(Constants.MIN\_WAN\_DELAY, Constants.MAX\_WAN\_DELAY) : ThreadLocalRandom.current().nextInt(Constants.MIN\_LAN\_DELAY, Constants.MAX\_LAN\_DELAY); 3. t.setDelay(t.getDelay() + delay); 5. // Generisanje kasnjenja u PPP protokolu 6. **int** divisor = Constants.AVERAGE\_TRANSACTION\_SIZE\_IN\_BYTES / Constants.IDENTIFIER\_SIZE\_IN\_BYTES; 8. **int** proposeDelay = linkType.equals(Link.LinkType.WAN) ? ThreadLocalRandom.current().nextInt(Constants.MIN\_WAN\_DELAY, Constants.MAX\_WAN\_DELAY) : ThreadLocalRandom.current().nextInt(Constants.MIN\_LAN\_DELAY, Constants.MAX\_LAN\_DELAY); 9. proposeDelay /= divisor; 11. **int** requestDelay = linkType.equals(Link.LinkType.WAN) ? ThreadLocalRandom.current().nextInt(Constants.MIN\_WAN\_DELAY, Constants.MAX\_WAN\_DELAY) : ThreadLocalRandom.current().nextInt(Constants.MIN\_LAN\_DELAY, Constants.MAX\_LAN\_DELAY); 12. requestDelay /= divisor; 14. **int** serveDelay = linkType.equals(Link.LinkType.WAN) ? ThreadLocalRandom.current().nextInt(Constants.MIN\_WAN\_DELAY, Constants.MAX\_WAN\_DELAY) : ThreadLocalRandom.current().nextInt(Constants.MIN\_LAN\_DELAY, Constants.MAX\_LAN\_DELAY); 15. t.setDelay(t.getDelay() + proposeDelay + requestDelay + serveDelay); 16. // Racunanje prosecnog kasnjenja 17. **public** **double** getAverageDelay() { 18. **return** ((**float**) sumDelay) / mempool.size(); 19. } |

Променљива ***sumDelay*** представља збир кашњења свих трансакција у RAM меморији посматраног сервера. Она се ажурира при сваком додавању нове трансакције у RAM меморију сервера. Генерисање кашњења на овакав начин је усвојено првенствено из разлога да би се показало какав је однос између просечног кашњења код алгоритма који се користи у тренутном решењу и просечног кашњења код PPP протокола.

* + - 1. **Одређивање просечног максималног броја хопова**

Број хопова представља пут који трансакција пређе од клијента до одредишног сервера, изражен у броју сервера на том путу. Број хопова представљен је додатним атрибутом који је додат уз трансакцију – ***hopCount.*** Ова променљива инкрементира се за сваку трансакцију у тренутку када се врши пријем те трансакције (односно додавање те трансакције у RAM меморију неког сервера). Просечан максимални број хопова осликава дубину графа мреже, јер показује колико је највише сервера у просеку морала да пређе клијентска трансакција, да би дошла до циљног сервера. У наставку је дат код који осликава ажурирање овог атрибута трансакције.

Табела 4.1.6 – Одређивање максималног броја хопова

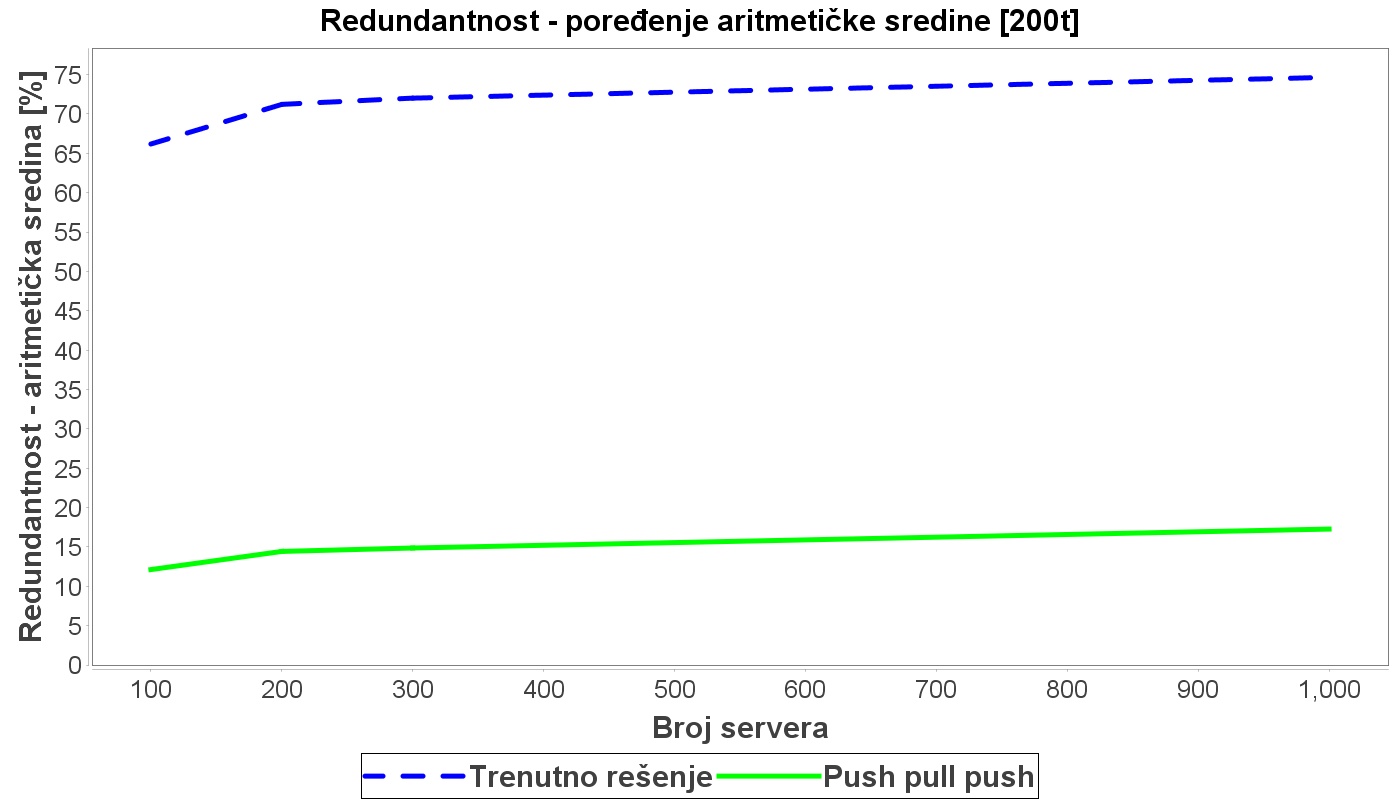
|  |
| --- |
| 1. // azuriranje lokalnog atributa transakcije 2. transaction.setHopCount(transaction.getHopCount() + 1); 3. mempool.add(transaction); 5. // odredjivanje transakcije sa maksimalnim brojem hopova 6. **public** **int** getMaxHopCount() { 7. **int** maxHopCount = 0; 9. **for** (Transaction t : mempool) { 10. **if** (t.getHopCount() > maxHopCount) { 11. maxHopCount = t.getHopCount(); 12. } 13. } 15. **return** maxHopCount; 16. } |

* 1. Резултати симулације и анализа добијених резултата

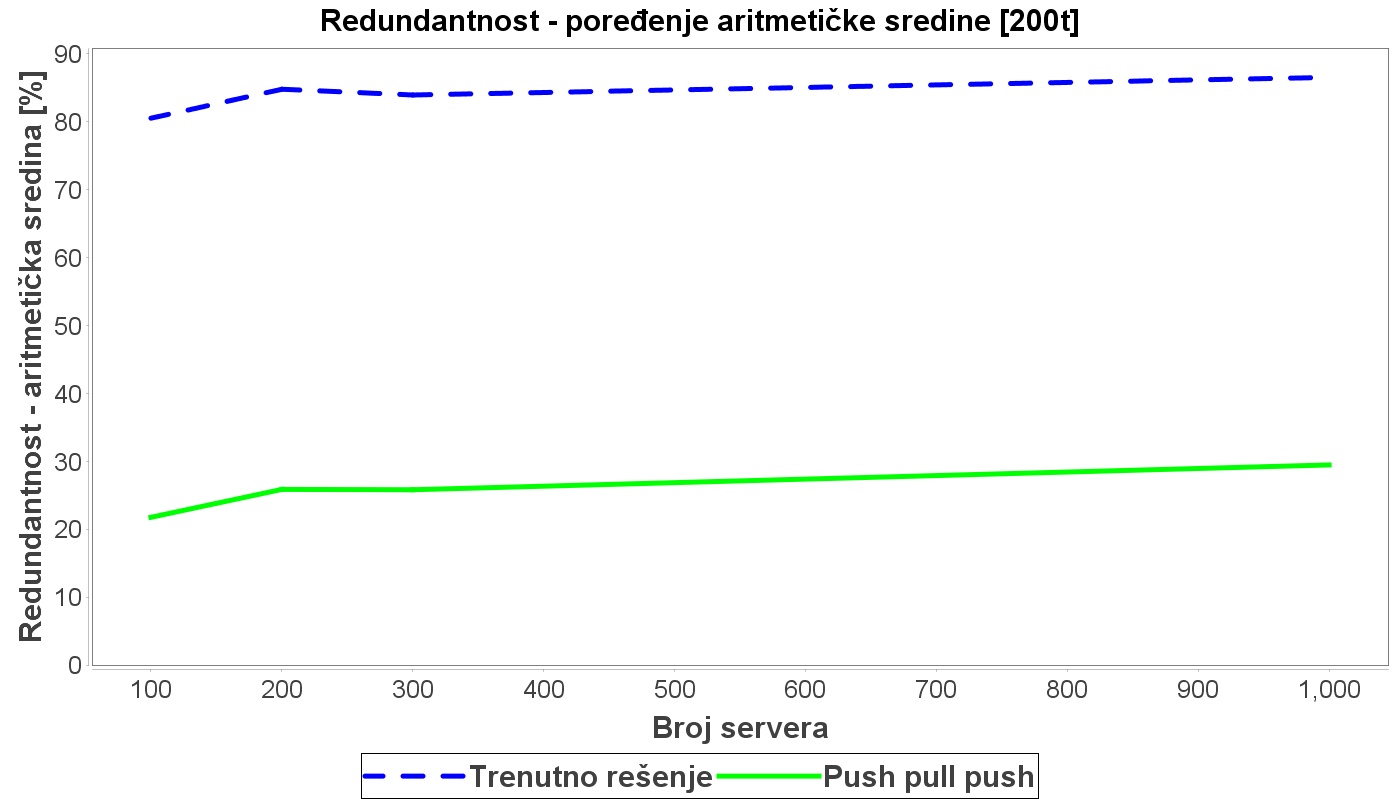
Симулације за различите параметре, помоћу којих су креирани графици који осликавају резултате симулација, покретане су позивом методе ***createMetrics()*** класе ***Simulator***. За креирање графика, коришћене су библиотеке *jfreechart* и *jcommon*. Симулације су покретане за {100, 200, 300, 1000} сервера, са увек фиксним бројем од 200 трансакција.

Како резултати симулације донекле зависе и од генератора насумичних бројева у програмском језику JAVA, и од распоређивања нити на JVM (*Java Virtual Machine*), за сваку комбинацију параметара {100s, 200t}, {200s, 200t}, {300s, 200t}, {1000s, 200t} покретане су по три симулације за сваки алгоритам (тренутно решење и PPP). Затим је као коначна вредност узета просечна вредност одговарајућих резултата добијених у те три симулације. Разлог је тај да би се што више елиминисала зависност симулације од програмског језика JAVA.

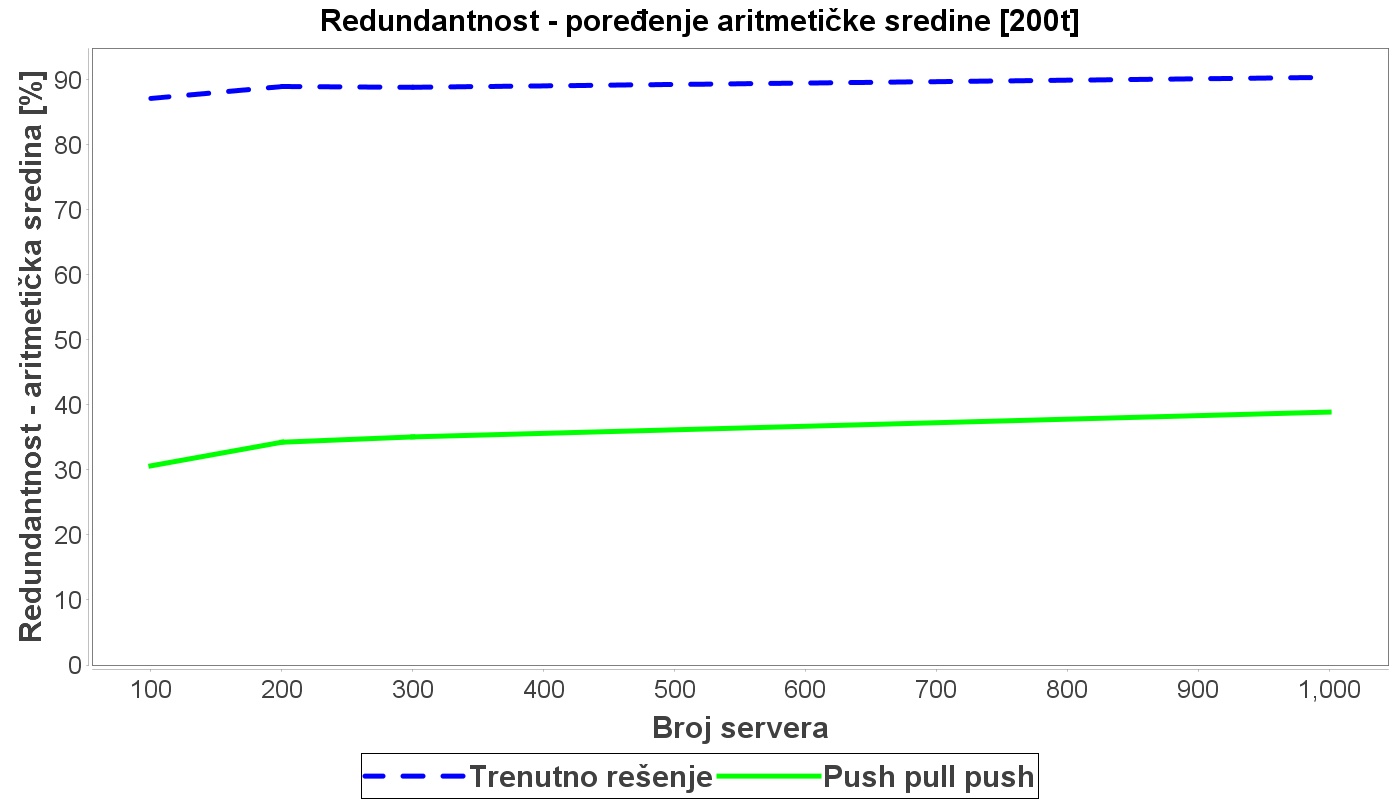
Као што је напоменуто раније, метода ***simulate(int, int, SimulationStrategy)*** као повратну вредност враћа листу сервера који су чинили симулирану мрежу. Управо ова листа се користи да би се израчунале одговарајуће метрике. Просечан проценат редундантности рачуна се тако што се збир процената редундантности свих сервера у листи подели бројем сервера. Просечно кашњење рачуна се тако што се збир просечних кашњења трансакција у свим серверима подели бројем сервера. Просечан максимални број хопова рачуна се тако што се збир максималног броја хопова (за неку трансакцију) свих сервера подели бројем сервера. На основу израчунатих метрика креирају се графици. У даљем делу документа, биће приложени ови графици, праћени дискусијом добијених резултата. На *x* оси приказан је број сервера. На *y* оси приказане су одговарајуће метрике – проценат редундантности, просечно кашњење и просечан максимални број хопова. Испрекидана линија на графику представља тренутни алгоритам за дисеминацију, док пуна линија представља PPP протокол.



Слика 4.2.1 – Проценат редундантности – просечан распон *f* = *ln(n)*

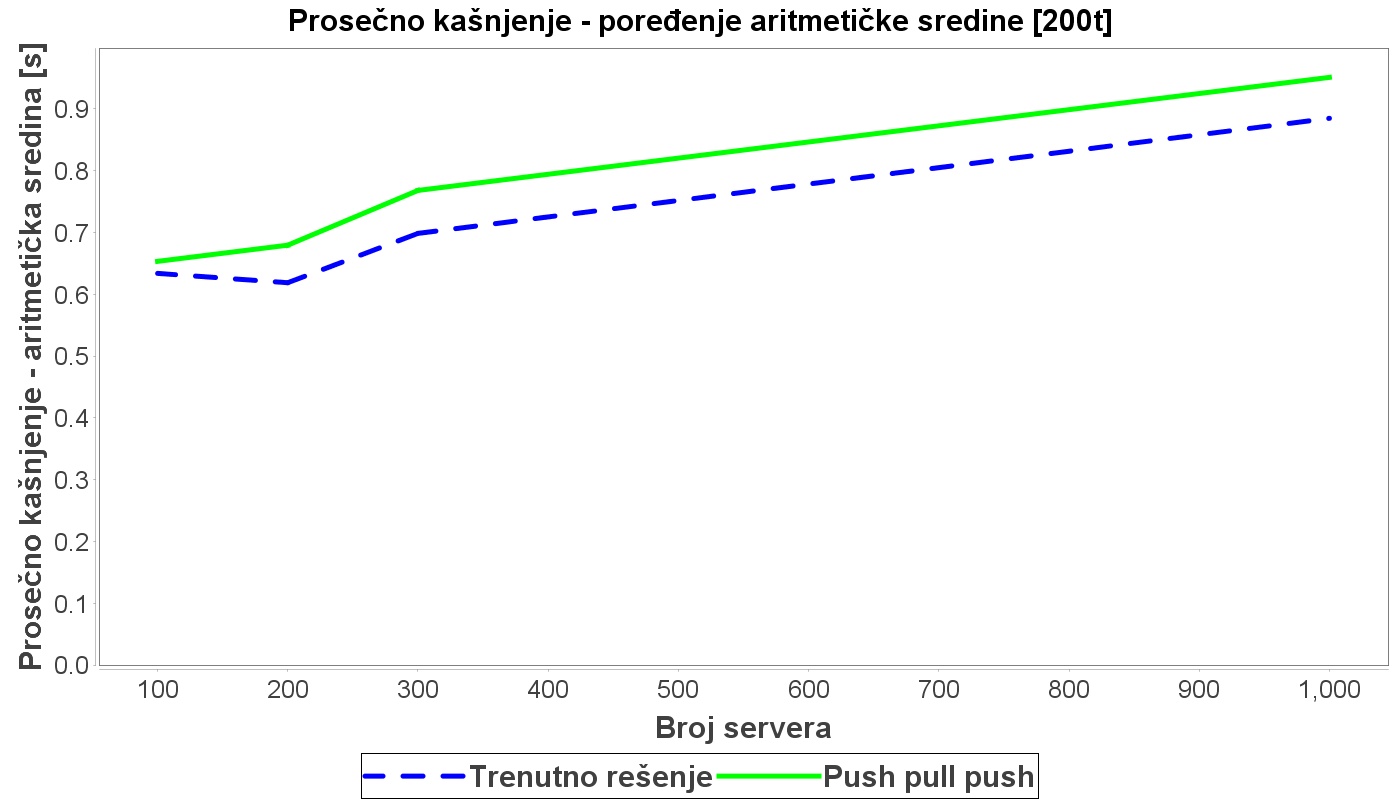


Слика 4.2.2 – Проценат редундантности – просечан распон *f* = *2ln(n)*

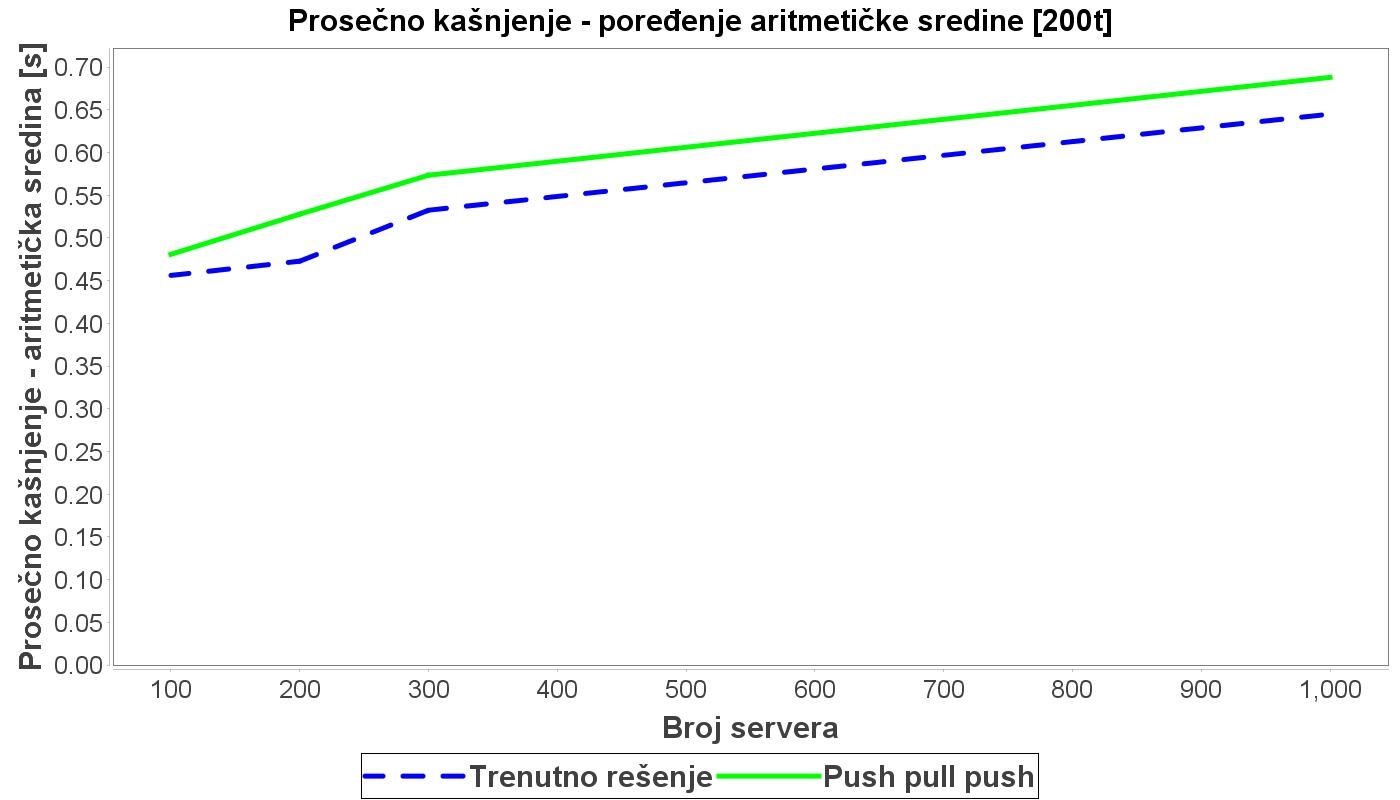


Слика 4.2.3 – Проценат редундантности – просечан распон *f* = *3ln(n)*

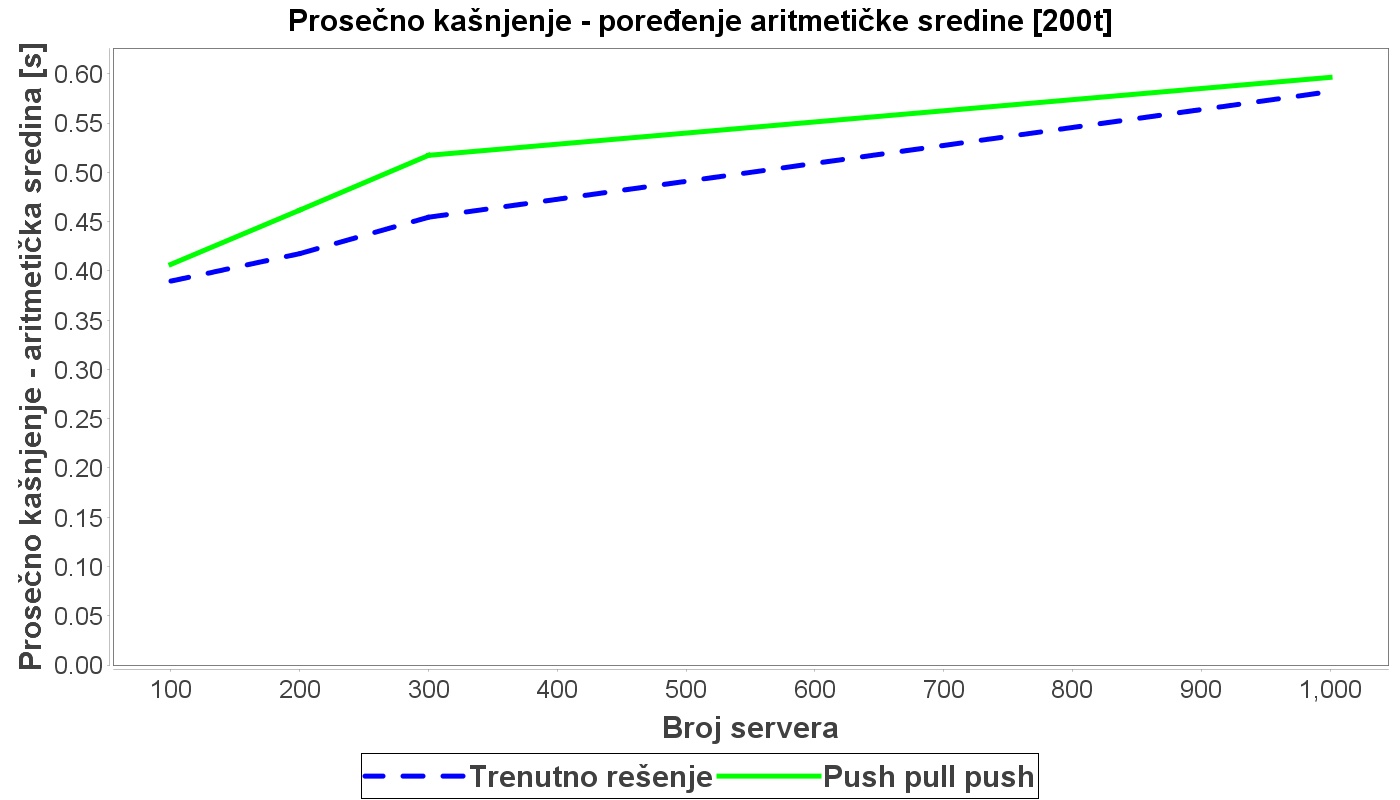
Очигледно је да проценат редундантности расте са повећањем распона чвора (односно величине подскупа суседа), које год решење да користи. Разлог за то је тај да, што више линкова имамо у графу мреже, порука се више дисеминира кроз мрежу. Међутим, графици показују да је редундантност која постоји код тренутног решења далеко већа у односу на редундантност која постоји код PPP протокола, поготово за оптималну величину просечног распона чвора ***ln(n)***.



Слика 4.2.4 – Просечно кашњење – просечан распон *f* = *ln(n)*

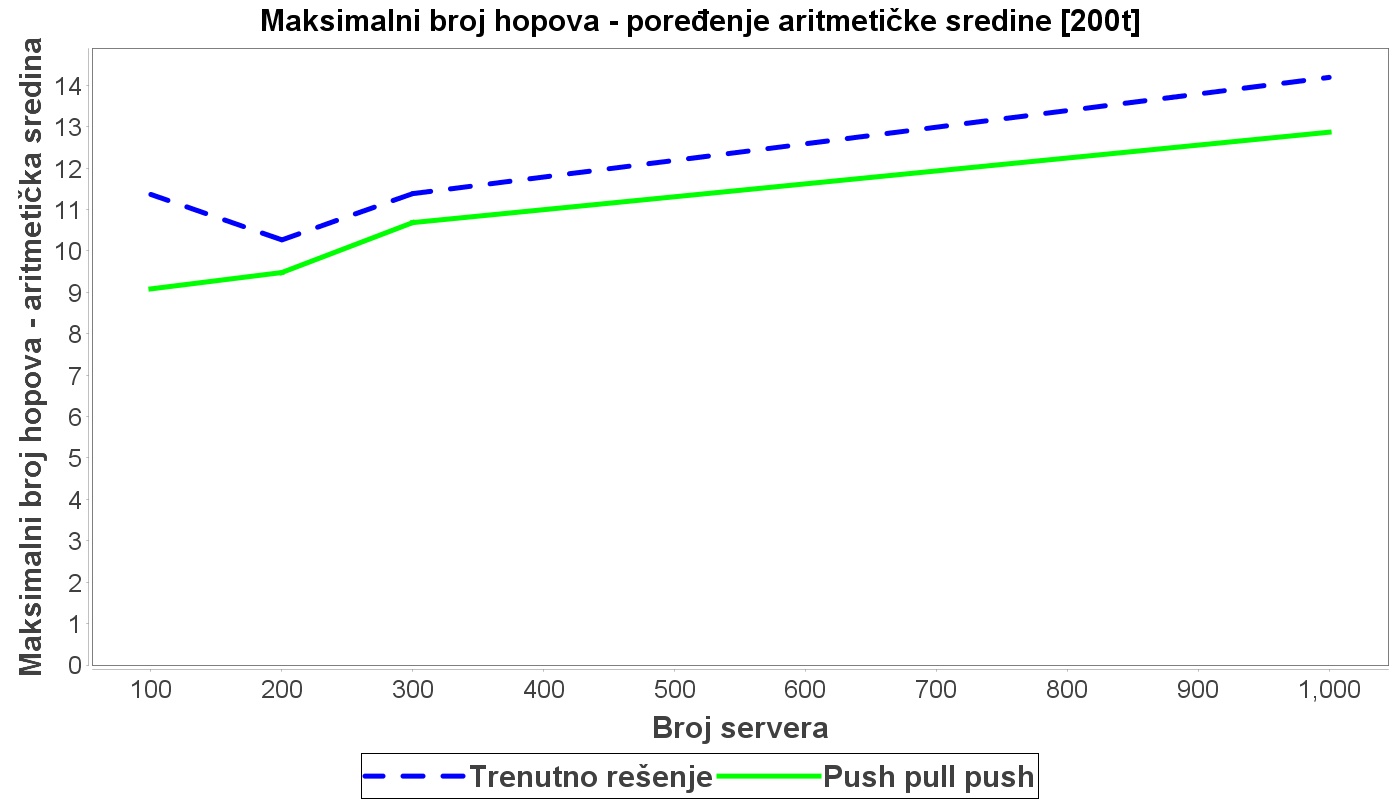


Слика 4.2.5 – Просечно кашњење – просечан распон *f* = *2ln(n)*

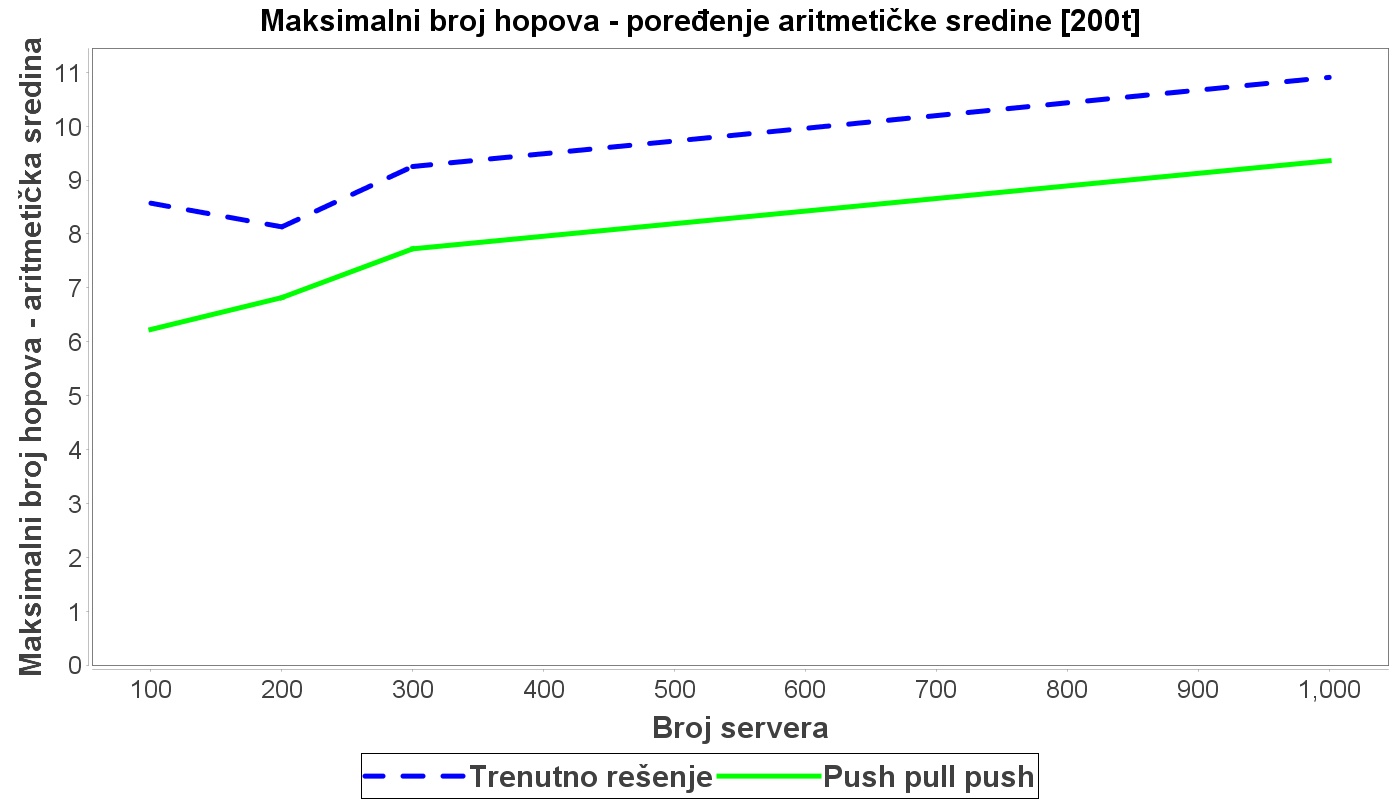


Слика 4.2.6 – Просечно кашњење – просечан распон *f* = 3*ln(n)*

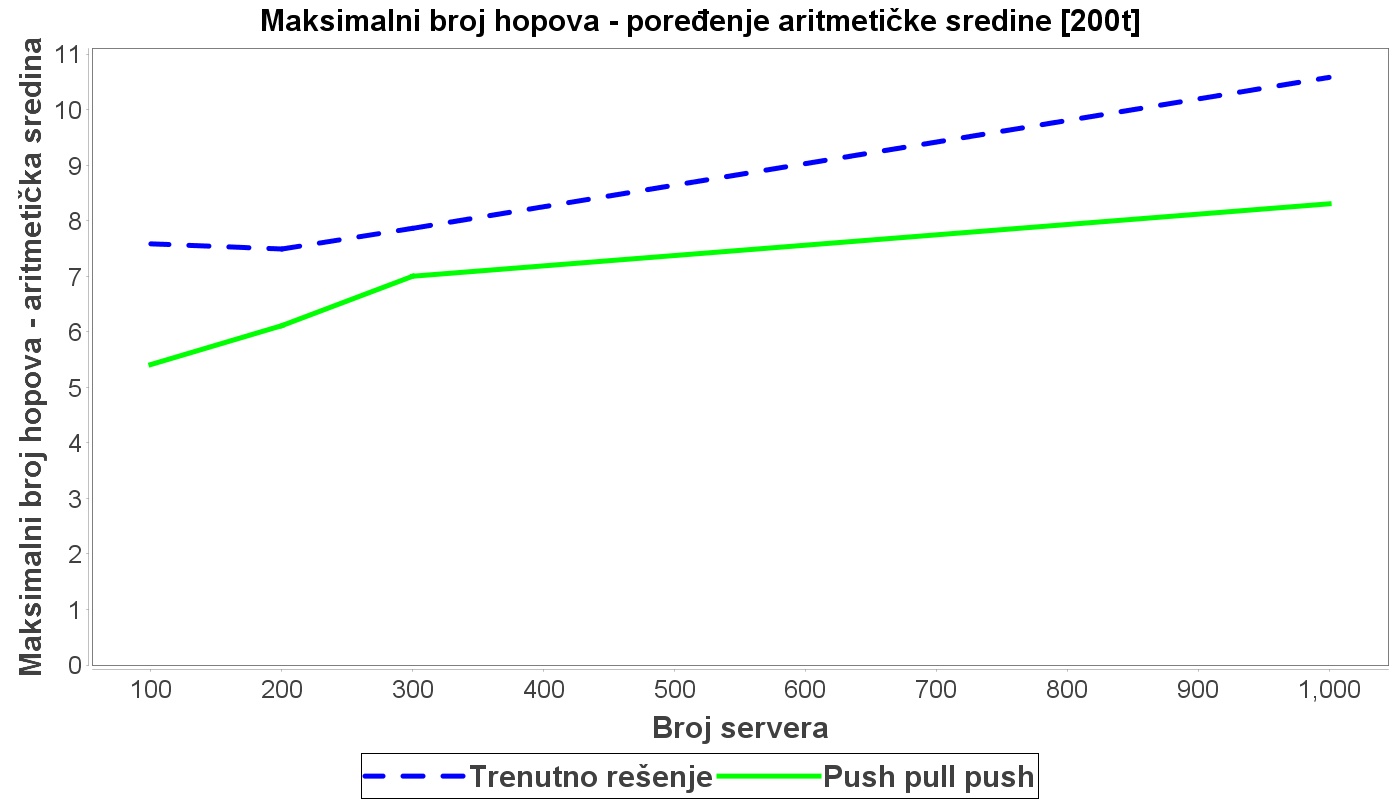
PPP протокол има веће просечно кашњење него тренутни алгоритам за дисеминацију. Ово је и очекивано, с обзиром на то да је PPP протокол који се састоји из три корака. Са порастом величине подскупа суседа, смањује се кашњење, зато што се порука брже шири кроз мрежу и брже долази до свих сервера.



Слика 4.2.7 – Просечан максимални број хопова – просечан распон *f* = *ln(n)*



Слика 4.2.8 – Просечан максимални број хопова – просечан распон *f* = *2ln(n)*



Слика 4.2.9 – Просечан максимални број хопова – просечан распон *f* = 3*ln(n)*

Са порастом максималне величине подскупа суседа, смањује се просечан максимални број хопова. Разлог је тај што, при већој повезаности сервера у мрежи, порука брже стиже до свих сервера. Просечан максимални број хопова је мањи код PPP протокола из следећег разлога – У овом протоколу, сервер увек тражи *SERVE* поруку од онога ко му је послао *PROPOSE*. То значи да, на самом почетку, када клијент пошаље своју трансакцију и неки сервер је прими, тај сервер послаће *PROPOSE* поруку свима у свом подскупу суседа. Они ће баш од њега захтевати ту трансакцију, чиме је број хопова минималан. Ова особина се даље рекурзивно одражава на читаву мрежу. Кад год неки сервер добије трансакцију, он шаље *PROPOSE* свима у подскупу суседа, а они ту трансакцију траже баш од њега.

Код тренутног решења, порука може брже да стигне до неког сервера по пет LAN линкова него што ће по једном WAN линку. Код PPP протокола, на сваком од пет LAN линкова биле би инициране све три фазе алгоритма, што би повећало укупно кашњење по сваком линку. Због тога, вероватније је да ће *PROPOSE* порука путовати до неког сервера по мањем броју линкова.

Из приложених резултата симулације може се закључити да је практичније да се користи PPP протокол за дисеминацију. Прво, просечна редундантност је доста мања код овог протокола, поготово за оптималну просечну величину подскупа суседа. Што се величине подскупа суседа тиче, њено повећавање смањује кашњење и максимални број хопова, а повећава редундантност. Можда је могуће наћи такву максималну вредност величине подскупа суседа у PPP протоколу, тако да проценат редундантности никад не превазилази 50%.

1. Закључак

Циљ ове тезе био је да реши проблем редундантности у алгоритму за дисеминацију трансакција. Тај алгоритам користи се у блокчејн мрежи по имену Тендерминт.

Први приступ решавању овог проблема била је формална анализа три различита алгоритма, од којих је сваки заснован на неком од основних приступа који се могу наћи у литератури. У два од три алгоритма, показано је да теоретски постоји идеално решење, у коме редундантности уопште нема. С друге стране, поједини концепти иза та два алгоритма су изазовни за имплементацију. Штавише, они су предмет многих научних истраживања. Трећи предложени алгоритам је могуће имплементирати у реалном систему. Он има веома добре особине узимајући у обзир да може да функционише у потпуно динамичком окружењу, и да је у потпуности заснован на принципима који важе за протоколе дисеминације.

Други приступ решавању проблема редундантности била је симулациона упоредна анализа тренутног алгоритма и протокола за дисеминацију који се састоји из три фазе. Симулациона анализа показала је да тај протокол има доста боље карактеристике наспрам тренутног решења, у истим условима симулације. Редундантност је значајно смањена, као и просечан максимални пут који трансакција пређе при дисеминацији кроз мрежу. Кашњење јесте повећано, али је то прихватљив недостатак, с обзиром на све предности које предложено решење доноси.

У даљим истраживањима, било би занимљиво фокусирати се на имплементацију неког од прва два предложена алгоритма. Такође, било би корисно размотрити имплементацију трећег предложеног алгоритма у мрежи која је непоуздана. Задатак би био учинити алгоритам таквим да буде отпоран на отказе у мрежи. Додатно, како је тренутна симулација конкурентна апликација, било би интересантно видети какви би били резултати симулације у стварном дистрибуираном окружењу, коришћењем дистрибуиране апликације за симулацију.

Литература

1. E. Buchman, J. Kwon, Z. Milosevic, "The latest gossip on BFT consensus", 2018. [Online]. Available: <https://github.com/tendermint/spec/releases/download/v0.5/paper.pdf> (07.09.2018.)
2. S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system" 2009. [Online]. Available: <http://www.bitcoin.org/bitcoin.pdf> (07.09.2018.)
3. M. Crosby, Nachiappan, P. Pattanayak, S. Verma, V. Kalyanaraman, "Blockchain technology - beyond Bitcoin" 2017. [Online]. Available: <https://j2-capital.com/wp-content/uploads/2017/11/AIR-2016-Blockchain.pdf> (07.09.2018.)
4. G. Zyskind, O. Nathan, A. Pentland, "Decentralizing privacy: using Blockchain to protect personal data" 2015. [Online]. Available: <https://ieeexplore.ieee.org/document/7163223/> (07.09.2018.)
5. J. Kwon, "Tendermint - consensus without mining" 2014. [Online]. Available: <https://cdn.relayto.com/media/files/LPgoWO18TCeMIggJVakt_tendermint.pdf> (07.09.2018.)
6. L. Lamport, R. Shostak, M. Pease, "The Byzantine Generals problem" 1982. [Online]. Available: <https://dl.acm.org/citation.cfm?id=357176> (07.09.2018.)
7. F. B. Schneider, "Implementing fault-tolerant services using the state machine approach - a tutorial" 1990. [Online]. Available: <https://www.cs.cornell.edu/fbs/publications/SMSurvey.pdf> (07.09.2018.)
8. A. Montresor, "Gossip and Epidemic Protocols" 2017. [Online]. Available: [http://disi.unitn.it/~montreso/ds/papers/montresor17.pdf](http://disi.unitn.it/%7Emontreso/ds/papers/montresor17.pdf) (08.09.2018.)
9. M. Castro, B. Liskov, "Practical Byzantine fault tolerance and proactive recovery" 2002. [Online]. Available: <http://www.pmg.csail.mit.edu/papers/bft-tocs.pdf> (09.09.2018.)
10. M. Jelasity, "Gossip-based protocols for large-scale distributed systems" 2013. [Online]. Available: <http://www.inf.u-szeged.hu/~jelasity/dr/doktori-mu.pdf> (09.09.2018.)
11. A.M. Kermarrec, L. Massoulie, A.J. Ganesh, "Probabilistic Reliable Dissemination in Large-Scale systems" 2002. [Online]. Available: <https://www.researchgate.net/profile/Ayalvadi_Ganesh/publication/2832321_Probabilistic_Reliable_Dissemination_in_Large-Scale_Systems/links/00b7d535646945981f000000.pdf> (09.09.2018.)
12. M. Castro, P. Druschel, A.M. Kermarrec, A. Nandi, A. Rowstron, A. Singh, "SplitStream: High-Bandwidth Multicast in Cooperative Environments" 2003. [Online]. Available: <http://rowstron.azurewebsites.net/PAST/SplitStream-sosp.pdf> (09.09.2018.)
13. D. Frey, R. Guerraoui, A.M. Kermarrec, M. Monod, V. Quema, "Stretching Gossip with Live Streaming" 2009. [Online]. Available: <https://hal.inria.fr/inria-00436130/document> (09.09.2018.)
14. D. Frey, R. Guerraoui, A.M. Kermarrec, B. Koldehofe, M. Mogensen, M. Monod, V. Quema, "Heterogeneous gossip" 2009. [Online]. Available: <https://infoscience.epfl.ch/record/140640/files/middleware-monod.pdf> (09.09.2018.)
15. M. Jelasity, R. Guerraoui, A.M. Kermarrec and M.V. Steen, "The Peer Sampling Service - Experimental Evaluation of Gossip-Based Implementations" 2004. [Online]. Available: <http://lpdwww.epfl.ch/upload/documents/publications/neg--1184036295all.pdf> (09.09.2018.)
16. Спецификација Тендерминта (Tendermint specification) - <https://github.com/tendermint/tendermint/tree/master/docs/spec> (11.09.2018.)
17. Јавни блокчејн Тендерминта (Cosmos SDK*)* **-** <https://cosmos.network/> (11.09.2018.)
18. Документација о валидаторима у Тендерминту (Validators) - <https://github.com/tendermint/tendermint/wiki/Validators> (11.09.2018.)
19. Имплементација функције ***checkTx(T)*** (Mempool.go) - <https://github.com/tendermint/tendermint/blob/master/mempool/mempool.go> (11.09.2018.)
20. Просечна величина трансакције у Bitcoin мрежи - <https://tradeblock.com/bitcoin/historical/1w-f-tsize_per_avg-01101> (11.09.2018.)
21. Програмски код симулатора - <https://github.com/lukamiletic95/simulator> (11.09.2018.)

Списак скраћеница

HEAP – *Heterogeneity aware protocol*

IP – *Internet protocol*

JVM – *Java Virtual Machine*

LAN – *Local area network*

P2P – *Peer to peer*

PPP – *Push-pull-push*

PSS – *Peer sampling service*

RAM – *Random access memory*

SMR – *State Machine Replication*

SPOF – *Single point of failure*

WAN – *Wide area network*

Списак слика

[Слика 2.2.1 – Стандардна клијент-сервер архитектура 6](#_Toc524604441)

[Слика 2.2.2 – Реплицирање сервера 7](#_Toc524604442)

[Слика 2.2.3 – Детерминистички коначни аутомат унутар реплике сервера 8](#_Toc524604443)

[Слика 2.2.4 – клијент-сервер архитектура заједно са Мемпул компонентом 9](#_Toc524604444)

[Слика 2.2.5 – Модел система 11](#_Toc524604445)

[Слика 2.2.6 – Блокчејн на сваком серверу 12](#_Toc524604446)

[Слика 2.2.7 – Блокчејн на сваком серверу 14](#_Toc524604447)

[Слика 3.1.1 – Тренутни алгоритам за дисеминацију трансакција у Тендерминту 17](#_Toc524604448)

[Слика 3.2.1 – Илустрација алгоритма за дисеминацију заснованог на клијенту 20](#_Toc524604449)

[Слика 3.3.1 – Подела мреже у кластере – *Hierarchical membership protocol* 23](#_Toc524604450)

[Слика 3.3.2 – Решавање проблема редундантности унутар кластера – *Flat membership server based protocol* 25](#_Toc524604451)

[Слика 3.3.3 – SMR сервери који се користе за формирање стабала 26](#_Toc524604452)

[Слика 3.3.4 – Пропагација клијентске трансакције при коришћењу стабала 27](#_Toc524604453)

[Слика 3.3.5 – Решавање проблема редундантности у целој мрежи – коначно решење 30](#_Toc524604454)

[Слика 3.4.1 – Поруке које се размењују у току PPP протокола 37](#_Toc524604455)

[Слика 3.4.2 – Додатне поруке које се размењују у HEAP протоколу 41](#_Toc524604456)

[Слика 3.4.3 – Структура сервера у PSS протоколу 43](#_Toc524604457)

[Слика 4.2.1 – Проценат редундантности – просечан распон *f* = *ln(n)* 60](#_Toc524604458)

[Слика 4.2.2 – Проценат редундантности – просечан распон *f* = *2ln(n)* 60](#_Toc524604459)

[Слика 4.2.3 – Проценат редундантности – просечан распон *f* = *3ln(n)* 61](#_Toc524604460)

[Слика 4.2.4 – Просечно кашњење – просечан распон *f* = *ln(n)* 62](#_Toc524604461)

[Слика 4.2.5 – Просечно кашњење – просечан распон *f* = *2ln(n)* 62](#_Toc524604462)

[Слика 4.2.6 – Просечно кашњење – просечан распон *f* = 3*ln(n)* 63](#_Toc524604463)

[Слика 4.2.7 – Просечан максимални број хопова – просечан распон *f* = *ln(n)* 64](#_Toc524604464)

[Слика 4.2.8 – Просечан максимални број хопова – просечан распон *f* = *2ln(n)* 64](#_Toc524604465)

[Слика 4.2.9 – Просечан максимални број хопова – просечан распон *f* = 3*ln(n)* 65](#_Toc524604466)

Списак табела

[Табела 3.1.1 – Тренутно решење – псеудокод 18](#_Toc524604467)

[Табела 3.1.2 – Једноставно побољшање тренутног решења 20](#_Toc524604468)

[Табела 3.2.1 – Алгоритам за дисеминацију заснован на клијенту – псеудокод 21](#_Toc524604469)

[Табела 3.3.1 - Пропагација клијентске трансакције при коришћењу стабала – псеудокод 28](#_Toc524604470)

[Табела 3.3.2 – Решавање проблема редундантности у целој мрежи – псеудокод 29](#_Toc524604471)

[Табела 3.3.3 – Кластеровање стабала – псеудокод 31](#_Toc524604472)

[Табела 3.4.1 – PPP протокол – псеудокод 35](#_Toc524604473)

[Табела 3.4.2 – HEAP протокол – псеудокод 40](#_Toc524604474)

[Табела 3.4.3 – PSS протокол – псеудокод 44](#_Toc524604475)

[Табела 3.4.4 – Псеудокод коначног решења 46](#_Toc524604476)

[Табела 4.1.1 – Активна нит сервера 51](#_Toc524604477)

[Табела 4.1.2 – пасивна нит сервера 52](#_Toc524604478)

[Табела 4.1.3 – Програмски код клијента 55](#_Toc524604479)

[Табела 4.1.4 – Одређивање процента редундантности 56](#_Toc524604480)

[Табела 4.1.5 – Генерисање и одређивање просечног кашњења 57](#_Toc524604481)

[Табела 4.1.6 – Одређивање максималног броја хопова 58](#_Toc524604482)