Universitatea “Politehnica” din Bucureşti
Facultatea de Electronică, Telecomunicații şi Tehnologia Informației

Algoritmi TCP – AQM de combatere a congestiei în rețelele de calculatoare

Proiect de diplomă

prezentat ca cerință parțială pentru obținerea titlului de
Inginer în domeniul Calculatoare şi Tehnologia Informației
programul de studii de licență Ingineria Informației

Conducător ştiințific
Conf. dr. ing Ștefan STĂNCESCU

Absolvent
George-Alexandru CARAIVAN

2014

Universitatea "Politehnica" din Bucureşti
Facultatea de Electronică, Telecomunicaţii şi Tehnologia Informaţiei
Departamentul de Electronica Aplicată şi Ingineria Informaţiei

Aprobat Director de Departament:
Prof. dr. ing. Pasca Sever

TEMA PROIECTULUI DE DIPLOMĂ
a studentului Caraivan M. George-Alexandru

1. Titlul temei:
Algoritmi TCP - AQM de combatere a congestiei in retele de calculatoare

2. Contribuţia practică, originală a studentului va consta în (*exclusiv* partea de documentare):
Studiu comparativ al algoritmilor AQM pentru TCP, optimizare a algoritmilor de tratare a congestiei prin tehnici AQM, simulare în Network Simulator a algoritmilor AQM, implementare cu recompilare NS cu algoritm modificat, comparatie de performante, reprezentare grafică performante, concluzii.

3. Proiectul se bazează pe cunoştinţe dobîndite în principal la următoarele 3-4 discipline:
Sisteme de operare, Reţele de calculatoare, Teoria transmisiunii informaţiei, Programarea calculatoarelor, Structuri de date si algoritmi.

4. Realizarea practică/proiectul rămâne în proprietatea:
Departamentului de Electronică Aplicată şi Ingineria Informaţiei

5. Proprietatea intelectuală asupra proiectului aparţine:
Departamentului de Electronica Aplicată şi Ingineria Informaţiei

6. Locul de desfăşurare a activităţii:
Departamentului de Electronică Aplicată şi Ingineria Informaţiei, Camera B227

7. Data eliberării temei: 03.12.2013

CONDUCĂTOR LUCRARE:
Conf. dr. ing. Ştefan Stăncescu

STUDENT:
George-Alexandru Caraivan

DECLARAŢIE DE ORIGINALITATE A PROIECTULUI DE DIPLOMĂ

Subsemnatul CARAIVAN M. GEORGE-ALEXANDRU , posesor al C.I. seria XR, nr. 296022, având CNP: 1900412090043 am întocmit proiectul de diplomă cu tema: „ ALGORITMI TCP - AQM DE COMBATERE A CONGESTIEI ÎN REŢELE DE CALCULATOARE " , în vederea susţinerii examenului de finalizare a studiilor universitare de licenţă, organizat de către Facultatea de Electronică, Telecomunicaţii şi Tehnologia Informaţiei, Departamentul Electronică Aplicată şi Ingineria informaţiei, din cadrul Universităţii POLITEHNICA din Bucureşti, sesiunea vară, anul universitar 2013-2014.

Luând în considerare conţinutul art. 143 din Legea Educaţiei Naţionale nr. 1/2011, al. (4) „*Îndrumătorii proiectelor de diplomă răspund în solidar cu autorii acestora de asigurarea originalităţii conţinutului acestora*" şi al. (5) „*Este interzisă comercializarea de lucrări ştiinţifice în vederea facilitării falsificării de către cumpărător a calităţii de autor al unui proiect de diplomă*", declar pe proprie răspundere, că această lucrare este originală, fiind rezultatul propriei activităţi intelectuale, nu conţine porţiuni plagiate, iar sursele bibliografice au fost folosite cu respectarea legislaţiei în vigoare.

Cunosc faptul că plagiatul sau prezentarea unui proiect elaborat de alt absolvent sau preluată de pe internet, din manuale şi cărţi, fără precizarea sursei constituie infracţiune (furt intelectual şi nerespectarea dreptului de autor şi a proprietăţii intelectuale) şi atrage după sine anularea examenului de diplomă, precum şi răspunderea penală.

Data: 27.06.2014 Semnătura: _____
(în original)

Cuprins

Lista acronimelor

LAN - Local Area Network
WAN - Wide Area Network
NSFNET - National Science Foundation Network
MSS – Maximum segment size
ACK- Acknowledgement
NTG - Normalized Throughput Gradient
BAU - Basic Adjustment Unit
RED - Random Early Drop
ARED - Adaptive Random Early Drop
AIMD - Additive increase multiplicative decrease
REDM - Random Early Drop modificat
RRED - Robust random early drop
FRED - Flow Random Early Drop
NS3 - Network Simulator 3
IDE - Integrated Development Environment

Introducere

Congestia este un fenomen din ce în ce mai întâlnit în natură. Am devenit conştient de acest fenomen când am citit o carte, recomandată de un prieten de la medicină, scrisă de Arnold Ehred. Titlul cărţii este "Sistemul de vindecare prin dietă fără mucus" şi în timp ce o citeam am înţeles că vorbea despre congestie din punctul de vedere al organismului o perspectivă pe care până atunci nu o luasem în considerare.

Întâmplător, citeam această carte în perioada în care trebuia să îmi aleg un subiect de licenţă şi m-am decis pe loc. Fenomenul congestiei este un fenomen care ne urmăreşte de mult timp dar care abia acum în timpurile moderne devine din ce în ce mai evident şi din toate domeniile în care poate fii găsit am ales să îl studiez în domeniul reţelelor de calculatoare.

Pentru a studia acest fenomen am avut nevoie de 2 lucruri: informaţii şi un mediu de lucru. Informaţiile despre acest fenomen le-am luat din articolele de specialitate publicate de-a lungul a 25 de ani de când a fost prima oară identificat fenomenul de congestie ca o posibilă problemă care poate apărea în reţelele de calculatoare. Mediu de lucru ales este o librărie care se cheamă NS3 şi care este folosită pentru simularea reţelelor de calculatoare atât la nivel de topologie cât şi la nivel de structură prin algoritmii care stau la baza transferului de date.

Din multitudinea de algoritmi care controlează congestia mi-am ales 3 algoritmi mai populari iar la cei trei am adăugat o versiune modificată a unuia dintre ei. Odată aleşi aceşti algoritmi mi-am imaginat un mediu congestiv în care să le pot observa performanţele iar apoi am trecut la realizarea lui prin modificarea şi recompilarea librăriei NS3 pentru a integra aceşti algoritmi şi a crea simularea propriu-zisă.

După ce mediu de lucru a fost pregătit am pornit simulările şi am analizat anumiţi parametri caracteristici precum procentul de pierderi, lungimea medie a cozii care sunt strâns legaţi de parametrii de performanţă rata de transfer şi întârzieri prin reţea.

În urma experimentului am observat că versiunea modificată are un control mai bun al congestiei decât versiunea originală reducând pierderile de pachete cu un ordin de mărime şi lungimea medie a cozii cu 16% adică întârzierile prin reţea devin mult mai mici.

Capitolul 1

1.1 Definiția congestiei

Congestia în rețelele de calculatoare se referă la starea rețelei atunci când o legătură sau un nod este nevoit să transfere atât de multe date încât acest lucru scade calitatea serviciului ducând la întârzieri, pierderi de pachete și respingerea conexiunilor noi.

De obicei congestia apare atunci când pentru a ajunge la destinație datele, care până acum au traversat sau provin din mai multe legături, sunt nevoie să circule printr-o singură legătură ca în cazul unor LAN-uri conectate prin legături WAN. Congestia apare și în cazul ruterelor care sunt asaltate de un trafic, de date, care depășește capacitatea lor de procesare.

Prima congestie a fost identificată în 1984 [1] atunci când pe rutele principale din rețeaua NSFNET, care se dorea a fi o rețea care să le ofere cercetătorilor acces la capacitățile de procesare ale supercalculatoarelor oferite de Fundația Națională de Știință, s-a observat o scădere a ratei de transfer de la 32 kbit/s la 40 bit/s. Atunci a apărut pentru prima oară nevoia de a controla congestia iar 3 ani mai târziu datorită lui Van Jacobson a fost implementat primul mecanism de control al congestiei.

Utilizatori rețelelor de calculatoare doresc să trimită date cât mai repede iar asta implică o rată de transfer foarte mare. Satisfacerea acestei nevoi poate duce la apariția următoarelor 2 probleme: prima este în cazul în care nevoia este satisfăcută în cazul ideal iar rețeaua nu va putea duce cantitatea de date și va apărea congestia, iar a 2-a în cazul în care se încearcă evitarea agresivă a primei probleme și ratele lor de transfer vor fi atât de mult reduse încât resursele rețelei nu mai sunt eficient utilizate. Scopul mecanismelor de control al congestiei este chiar rezolvarea acestor 2 probleme de mai sus astfel încât resursele să fie bine utilizate și utilizatorul să aibă o rată de transfer cât mai bună.

1.2 Metode de control a congestiei

Pentru a gestiona congestia următoarele tehnici sunt folosite:

- Tehnici de control al fluxului
- Tehnici de controlul al congestiei
- Tehnici de evitare a congestiei
- Alocarea resurselor
- Depozitarea datelor

Tehnicile de control al fluxului nu sunt o propriu zis o modalitate de gestiune a congestiei ci sunt folosite ca metode de a împiedica supraîncărcărea bufferelor de la destinație de către surse. Cu alte cuvinte aceste tehnici reduc congestia la destinație.

Tehnicile de control al congestiei sunt similare technicilor de control al fluxului dar scopul nu mai este reducerea congestiei la desținație ci reducerea congestiei în rețea.

Tehnicile de evitare a congestiei aduc în ecuație ruterele pe care pot fi implementate mecanisme cu ajutorul cărora se decide dacă congestia poate avea loc caz în care sursele vor fi informate să își reducă rata de transfer înainte ca cozile de la bufferele rutelor să se umple și să apară propriu-zis fenomenul de congestie.

Alocarea resurselor presupune programarea resurselor pentru anumite perioade de timp. Aceasta tehnică poate elimina congestia din rețea blocând traficul în exces.

Depozitarea datelor presupune mutarea datelor mai aproape de utilizatori astfel încât majoritatea traficului devine local evitându-se traversarea unor legături posibil congestionate.

Capitolul 2

2.1 TCP

De la primul mecanism de control al congestiei propus de Van Jacobson şi implementat în anii 1988 metodele de evitare a congestiei au devenit din ce în ce mai sofisticate.

Pentru realizarea transferului de date în cel mai simplist mod sunt folosite 2 tipuri de ferestre, fereastra de transmisie care aparţine sursei şi fereastra de recepţie care bineînteles aparţine destinaţiei.

Fereastra de transmisie reprezintă dimensiunea datelor pe care TCP le poate transmite fără a primi mesaj de confirmare a recepţiei datelor de la destinaţie. Fereastra de recepţie reprezintă cât de multe date poate receptorul primi atunci când datele nu ajung în ordinea lor naturală.

Din punctul de vedere al transferului de date datelor trebuie să ne asigurăm că niciodată fereastra de transmisie nu este mai mare decât cea de recepţie.

Rata de transfer depinde în mod indirect de dimensiunea ferestrei de transmisie. Presupunem că utilizatorul doreşte să transmită date cu o fereastra de transmisie este egală cu dimensiunea maximă a unui segment. Transferul se va realiza ca în figura de mai jos:

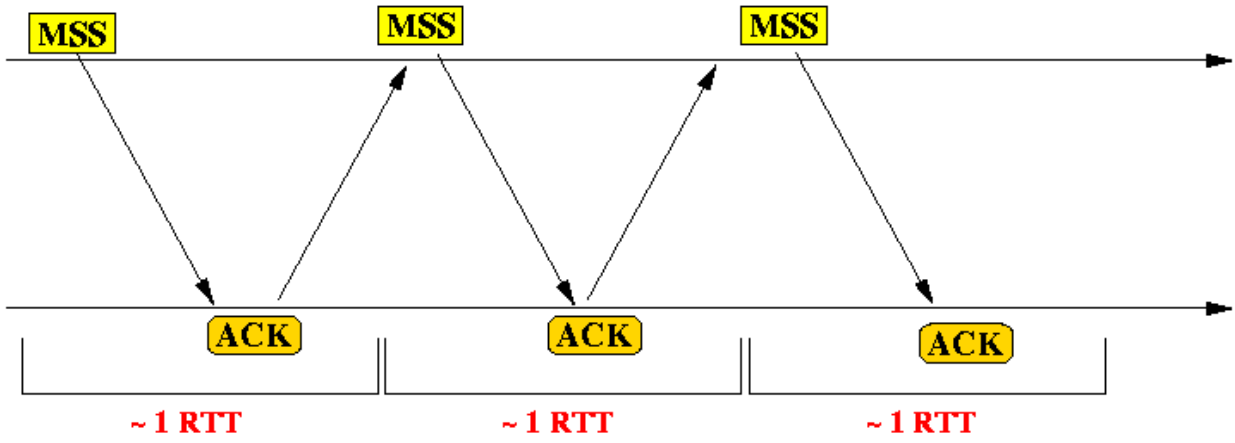

Figura 2.1 “Rata de transfer” [2]

Sursa poate trimite doar un pachet iar apoi este nevoită să aştepte primirea unui mesaj de confirmare. Mesajul de confirmare va veni în aproximativ RTT secunde iar atunci sursa poate trimite următorul pachet. Rata de transfer poate fi aproximată ca fiind MSS/RTT bps.

Dacă fereastra ar fi fost dublă adică egală cu 2MSS atunci sursa va putea trimite mai rapid:

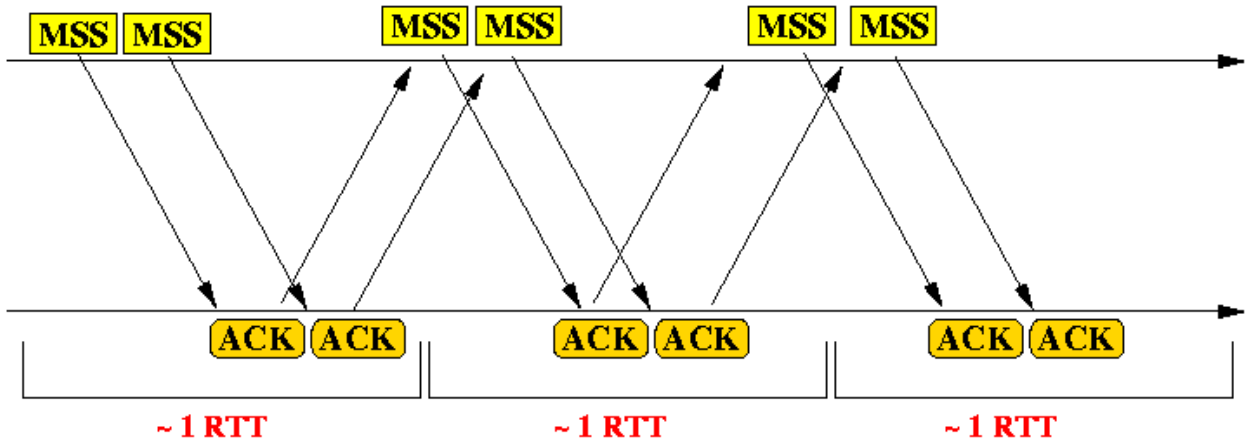

Figura 2.2 “Rată de transfer dublă” [2]

Cu toate aceastea rata de transfer nu este proporţională cu dimensiunea ferestrei pentru că în realitate trebuiesc luati în considerare şi alţi factori precum întârzierile, schimbările de rută care fac ca relaţia dintre dimensiunea ferestrei şi rata de transfer să devină impredictibilă.

Din nevoie de a controla congestia a mai fost implementată o nouă fereastră numită fereastra de congestie. Fereastra de congestie este cantitatea de date pe care sursa o poate trimite fără să existe riscul de a cauza congestie. Acest parametru variază în funcţie de statusul curent al reţelei care este impredictibil.

Ca şi în cazul legăturii dintre fereastra de transmisie şi cea de recepţie, fereastra de transmisie nu trebuie să fie mai mare decât fereastra de congestie.

Din cele 2 restricţii pentru fereastra de transmisie rezultă ca ea va lua valoarea minimului dintre fereastra de congestie şi cea anunţată de destinaţie astfel încât algoritmul de control al congestiei va ajusta fereastra de congestie în funcţie de evenimentele din reţea influenţând în mod direct fereastra de transmisie care va rezulta într-o modificare a ratei de transfer.

2.2 Algoritmi TCP de control al congestiei

2.2.1 Start-încet

Primul mecanism de control al congestiei a fost dezvoltat de Jacobson şi Karels. Acest mecanism are la bază principul de conservare al pachetelor [3] iar conexiunea care respectă acest principiu se consideră că este la echilibru. Dacă toate conexiunile sunt la echilibru congestia nu poate să apară.

Principiul conservării pachetelor poate încălcat în 3 cazuri [4] :

1. conexiunea nu ajunge la echilibru
2. sursa introduce un pachet nou în reţea înainte ca un pachet vechi să părăsească reţeaua
3. congestia din reţea împiedică ca o conexiune să ajungă la echilibru.

Pentru ca facilita atingerea echilibrului a fost dezvolvat mecanismul start-încet. Fereastra de congestie a apărut pentru prima oară datorită acestui algoritm.

La realizarea unei noi conexiuni sau atunci când o conexiune este reinițializată după o pierdere de pachete fereastra de congestie este setată la un pachet iar la recepționarea unui ACK va crește cu un pachet. Respectând acestă metoda se estimează că fereastra de congestie va ajunge la dimeansiunea ferestrei de recepție în RTT log2 W secunde [5], unde W este dimensiunea ferestrei de recepție măsurată în pachete.

Cazul 2 de încălcare a principiului de conservare a pachetelor apare atunci când timpul de retransmitere este prea scurt și face ca sursa să retransmită un pachet despre care nu se știe nici dacă a fost recepționat nici dacă a fost pierdut. Pentru a evita acest lucru este necesă o metodă eficientă de estimare a RTT [6]:

$$\text{Eroare} = \text{Ultimul\_RTT} - \text{RTT\_Estimat}$$
$$\text{RTT\_Estimat} = \text{RTT\_Estimat} + g * \text{Eroare}$$

Paramentrul g reprezintă o creștere legată de varianță. Aceasta este o îmbunătățire a metodei anterioare care folosea o constantă pentru varianță.

Cazul 3 este rezolvat de mechanismele de evitare a congestiei, pierderile de pachete fiind strâns legate de congestie deoarece evenimentele în care un pachet este pierdut datorită transitului prin rețea sunt extrem de rare.

Atunci când apare congestia fereastra de congestie a ruterului este setată la jumătate din ferestra curentă iar apoi pentru fiecare ACK primit fereastra crește cu inversul dimensiunii ferestrei de congestie.

2.2.2 Start-încet cu căutare

Algoritmul start-încet suferă oscilații atunci când rețeaua este supraîncărcată. Ferestra oscilează între dimensiunea maximă posibilă a ferestrei pentru o legătură gâtuită și jumătate din acea valoare, fapt care provoacă întârzieri.

O altă problemă a algoritmului este că favorizează conexiunile cu mai puține hopuri. Creșterea aditivă și scăderea multiplicativă forțează ca la un anumit moment dat dimensiunile tutoror ferestrelor de transmisie să fie egale. Cu toate acestea până se va realiza acest lucru conexiunile scurte cu RTT mai mici vor crește rapid.

Pentru a rezolva probleme de mai sus Wang a propus un nou mecanism numit start-încet cu căutare, parte care foloseste orice schimbare a ratei de transfer ca indicație a congestiei. Algoritmul calculează gradientul normalizat al ratei de transfer și îl compară cu niște praguri pentru a menține conexiunea la un punct de operare optim care maximeaza rata de transfer și evită congestia.

Pe măsură ce solicitările cresc rata de transfer crește liniar până când rețeaua atinge capacitatea limită. Gradientul graficului ratei de transfer este folosit pentru semnalarea congestiei. Odată cu apariția de noi conexiuni graficul ratei de transfer se nivelează iar atunci când ele dispar graficul

devine liniar. Gradientul ratei de transfer este exprimat în formula următoare:

$$TG(W_n) = (T(W_n) - T(W_{n-1})) / (W_n – W_{n-1})$$

unde W_n este dimensiunea ferestrei n iar $T(W_n)$ este rata de transfer atunci când se foloseste fereastra de dimensiune W_n. Pentru a ţine cont de RTT-ul diferilor conexiuni gradientul este normalizat după formula de mai jos:

$$NTG(W_n) = TG(W_n) / TG(W_1)$$

Gradientul normalizat al ratei de transfer varianză între 1 şi 0 odată cu schimbarea traficului. Pentru un trafic lejer NTG va fi aproape de 1 deoarece a creşterea ratei de transfer este proporţinală cu o creştere a solicitărilor. Atunci când solicitările fac ca reţeaua să fie folosită la capacitate maximă, NTG atinge 0.

Rata de transfer medie a unei conexiuni atunci când packetul n este în transit este : $T_n = W_n / RTT_n$ unde W_n este dimensiunea ferestrei de transmisie atunci când packetul n a fost trimis şi RTT_n este RTT pachetului n.

Atunci când se realizează o nouă conexiune, fereastrei de transmisie este setată la 1 BAU şi creşte cu un pachet pentru fiecare ACK primit până când devine egală cu dimensiunea ferestrei de recepţie.

Atunci când timerul unui anumit pachet expiră, dimensiunea ferestrei de transmisie este setată la 1 BAU şi este crescută cu câte un BAU pentru fiecare ACK atât timp cât NTG este deasupra unei anumit prag NTG_d. Când fereastra creşte cu mai mult de un pachet NTG este din nou calculat. Dacă NTG este sub un anumit prag NTG_i dimensiunea ferestrei este scazută cu un pachet.

Setarea corectă a pragurilor NTG_d şi NTG_i este foarte importantă deoarece pragul NTG_d determină punctul de operare al conexiunilor iar NTG_i determină sensibilitatea cu care se detectează eliberarea legăturii.

Acest algoritm foloseste RTT primului pachet pentru estimarea întârzierii de propagare iar dacă conexiunea este deja încărcată estimarea este incorectă.

2.2.3 DUAL

Pentru a corecta oscilaţiile asociate algoritmului start-încet Wang a propus şi o altă metodă numită DUAL care este similară algoritmului start-încet cu căutare.

RTT unei pachet este suma dintre întârzierile de propagare şi întârzierile suferite la trecerea prin ruter. Valoarea minima a RTT va fi egală cu întârzierile de propagare, D_p, fapt care se poate întâmpla atunci când întârzierile prin ruter sunt nule:

$$RTT_{min} = D_p$$

Valoarea maximă a RTT presupune că întârzierile prin ruter sunt maxime şi anume atunci când coada ruterului unei legăturii gâtuite este plină.

$$RTT_{max} = D_p + B / R$$

unde B este dimensiunea maximă a bufferului ruterului şi R este rata de procesare a pachetelor.

Algoritmul start-încet detectează congestia atunci când un pachet este pierdut din cauza unei cozi pline. Soluţia propusă de această metodă este folosirea unui prag pentru a evita supraîncărcarea cozii ruterului unei legături gâtuite. Pragul folosit depinde de minimul şi maximul RTT - ului şi este definit de următoarea relaţie:

$$RTT_i = (1- a) * RTT_{min} + a * RTT_{max}$$

unde a < 1 [5].

Metoda DUAL foloseşte algoritmul start-încet pentru iniţializarea şi repornirea conexiunilor dar diferă prin faptul că la fiecare 2 * RTT secunde verifică dacă RTT curent este mai mare decât RTT_i şi reduce fereastra de congestie cu 7/8. Pe lângă acest lucru minimul şi maximul RTT sunt recalculate la fiecare estimare a RTT iar atunci când timerul pentru un anumit packet expiră, pe lângă reducerea ferestrei de congestie la un pachet şi repornirea cu algoritmul start-încet, setează minimul şi maximul RTT la 0, respectiv infinit.

2.2.4 TCP Reno

Algoritmul Reno este primul algoritm care a implementat mecanismele de retransmitere rapidă şi recuperare rapidă prezentate mai jos.

Înainte de apariţia algoritmului de retransmitere rapidă TCP folosea un mecanism de timeout prin care aştepta ACK. După trimiterea unui pachet se seta un timer pentru acel pachet iar dacă ACK pentru acel pachet venea înainte de expirarea timerului, timerul pentru acel pachet era resetat şi se aşteptau ACK pentru celelalte pachete. În schimb dacă ACK nu ajungea în timp util pachetul era retransmis şi conexiunea repornea cu start-încet.

Cu toate acestea s-a observat că acest mecanism reacţiona la evenimente din cadrul reţelei la care nu ar fi trebui să reactioneze ducând la o gestiune ineficientă a timpului. Din acest motiv a fost creat retransmiterea rapidă, un mecanism care folosit în combinaţie cu ferestre de dimensiuni mari îmbunătăţeşte performanţa. Mecanismul de timeout încă mai există dar este folosit doar pentru atunci când ferestrele sunt mici.

Atunci când mecanismul de retransmitere rapidă este activ nu se mai aşteaptă expirarea timerului şi când sursa primeşte 3 ACK duplicat retransmite imediat pachetul iar apoi reporneşte cu start-încet. Numărul de 3 a fost luat în considerare deoarece nu se poate ştii dacă un ACK duplicat a fost primit din cauză că s-a pierdut un segment sau este necesară o reordonare a pachetelor la destinaţie. Presupunând că atunci când pachetele nu sunt primite în ordinea în care au fost trimise reordonarea lor la destinatar ar putea dura 2 ACK astfel încât doar cel de-al 3-lea va putea indica că un segment a fost pierdut cu adevărat [7].

Mecanismul de recuperare rapidă previne intrarea în modul start-încet deoarece primirea unui ACK duplicat indică faptul că încă mai există trafic între sursă şi destinaţie. Acest mecanism permite o

rata de transfer mare într-un mediu cu congestie moderată. Spre deosebire de retransmiterea rapidă după ce este retrimis pachetul lipsă, pragul start-încet este ca de obicei setat la jumatatea ferestrei de congestie dar fereastra de congestie nu mai este setată la un pachet ci este setată la valoarea pragului start-încet plus încă trei pachete.

2.2.5 TCP Vegas

Vegas foloseşte un mecanism de retransmisie mai bun decât mecanismul de retransmitere rapidă. Pe când mecanismul de retransmitere rapidă foloseşte 3 ACK duplicat pentru a stabili dacă pachetul a fost pierdut, mecanismul de retransmisie folosit de Vegas înregistrează timpul trimiterii pachetelor pentru a calcula RTT pentru fiecare ACK primit. Atunci când un ACK duplicat este primit Vegas verifică dacă diferenţa dintre timpul înregistrat pentru acel pachet şi timpul curent este mai mare decât durata timpului de aşteptare. Dacă este, Vegat retransmite pachetul fără a aştepta cel de-la treilea ACK duplicat. Acest mecanism este mai bun deoarece în multe cazuri fereastra sursei este atât de mică încât este posibil să nu aibă loc să primească 3 ACK duplicat sau unul dintre ACK să fie pierdut în reţea.

Atunci când este primit un ACK neduplicat şi este primul sau al doilea ACK primit de la retransmisie, Vegas verifică daca intervalul de timp de când packetul a fost trimis este mai mare decât durata timpului de aşteptare şi dacă este retransmite packetul. Dacă exista pachete care au fost pierdute de la începerea retransmisie ele vor fi retransmise fără aşteptarea unor ACK duplicat.

Pentru a evita congestia Vegas compară rata de transfer actuală cu rata de transfer estimată. Rata de transfer estimată este minimul dintre toate ratele de transfer iar rata de transfer actuală este numărul de octeţi transmişi în timpul scurs între transmiterea pachetului şi recepţionarea ACK supra RTT acelui pachet. Vegas calculează diferenţa dintre valoarea ratei de transfer estimate şi valoarea ratei de transfer actuale şi o compară cu 2 praguri a şi b. Daca diferenţa este mai mică decât pragul a atunci fereastra creşte liniar iar dacă diferenţa este mai mare mare decât pragul b fereastra este scade liniar.

Capitolul 3

3.1 AQM

Deşi TCP dispune de mecanisme sofisticate de control al congestiei el nu poate funcţiona corect într-un mediu congestiv dacă nu există o modalitate prin care transferul de date să fie încetit pentru ca reţeaua să nu ajungă să fie suprasolicitată.

Scopul AQM este să ofere o metodă prin care se evită încărcarea totală a bufferelor fenomen care duce la reducerea ratei de transfer. În mod ideal bufferele ar trebui să fie aproape goale astfel încât întârzierea pachetelor prin router să fie cât mai mică.

În lipsa unui algoritm AQM ruterele adaugă în buffer cât de multe pachete pot iar atunci când nu mai au spaţiu le aruncă pe cele pe care nu le mai pot stoca. Acest lucru duce la fenomenul de sincronizare globală TCP. Atunci când bufferul ruterelor este plin şi încep să arunce pachetele pe care nu le mai pot încărca, conexiunile TCP încep simultan să funcţioneze în modul start-încet. Testează capacitatea reţelei şi să îşi mărească ferestrele de transfer simultan fapt care la un moment dat va duce din nou la supraîncărcarea reţelei. Când acest lucru se va întâmpla rutele vor arunca din nou pachetele pe care nu le mai pot stoca şi din nou conexiunile TCP vor reporni simultan cu start-încet. Pentru a evita ca acest fenomen să se întâmple la infinit a trebuit să fie creat un mecanism prin care conexiunile TCP să fie anunţate să îşi reducă rata de transfer la momente de timp diferite.

3.2 Algoritmi AQM de control al congestiei

Domeniul AQM este un domeniu deosebit de larg. Din acest domeniu larg am ales pentru studiu şi pentru comparaţie 3 algoritmi: cel mai vechi (RED), varianta lui modernă (ARED) şi încă un algoritm modern numit BLUE.

3.2.1 RED

Primul algoritm care a soluţionat problema sincronizării totale TCP a fost RED care este un acronim pentru aruncarea aleatoare timpurie, în engleză “Random Early Drop”. Modul de funcţionare RED este acela de a monitoriza dimenziunea medie a cozii de pachete de la bufferul unui ruter şi să arunce pachetele pe baza unor probabilităţi calculate în funcţie de starea cozii. Atunci când bufferul este aproape gol este normal ca toate pachetele să fie aceptate de către ruter iar pe măsura ce spaţiul se umple prin încărcarea de pachete probabilitatea de aruncare creşte până când atinge unitatea respectiv cazul în care toate pachetele sunt aruncate.

Pseudocodul algoritmului RED este prezentat mai jos:

```
avg = 0
count = -1
```

```
pentru fiecare pachet
     calculăm noua valoare a cozii medii
          daca lungimea cozii diferită de zero
               avg = (1 - wq) * avg + wq * q
          altfel
               m = f (time – q_time)
               avg = (1 – wq) mavg
     daca min_prag <= avg < max_prag
          count = count + 1
          calculeaza probabilitatea pa:
               pb = maxp(avg – min_prag) / (max_prag – min_prag)
               pa = pb / (1 – count * pa)
          cu probabilitatea pa:
               marchează pachetul curent
               count = 0
     altfel daca max_prag <= avg
          marcheaza pachetul curent
          count = 0
     altfel count = -1
când lungimea cozii este nulă:
q_time = time
```

Variabile folosite:

avg : lungimea medie a cozii
q_time: startul timpului de idle al cozii
count : numărul de pachete de la ultima marcare.
w_q : ponderea cozii actuale
min_prag : pragul minim al cozii
max_prag : pragul maxim al cozii
max_p : valoarea maximă pentru probabilitatea p_b
p_a : probabilitatea de marcare a pachetului curent
q : lungimea curentă a cozii
time : timpul curent
f(t) : funcție liniară pentru parametrul timp.

3.2.1.1 Calcularea cozii medii

Pentru a calcula lungimea medie a cozii se folosește o filtrare trece-jos. Această filtrare are rolul de a evita ca perioadele de timp reduse în care lungimea cozii crește brusc să conteze pentru calculul cozii medii. Aceste creșteri bruște a cozii se pot datora unei mici explozii a traficului sau unei congestii tranzitorii care poate apărea atunci când o conexiune TCP își descoperă fereastra maximă de transfer aducând pentru un interval redus de timp un surplus de date în rețea pentru care ruterul va trebui să o informeze ca a atins limita și să-și reducă transferul. Acest filtru trece-jos este o medie mobilă exponențială și are formula:

$$avg = (1-w_q) * avg + w_q * q$$

Dacă ponderea w_q este prea mare media nu va filtra congestia de scurtă durată care poate apărea la ruter. Iar dacă este prea mică media va răspunde prea încet la schimbările cozii curente q. Conform [8] este bine ca ponderea să satisfacă următoarea equație :

$$L + 1 + \frac{(1-w_q)^{L+1}-1}{w_q} < min_{th}$$

unde L este lungimea maximă a cozii măsurată în pachete.

3.2.1.2 Setarea parametrilor min_{th} și max_{th}

Valorile pentru cei doi parametrii depind de valoarea dorită a cozii medii. Dacă traficul are loc în rafale este de dorit ca pragul minim să fie cât mai mare pentru a avea o eficiență ridicată. Pragul maxim se setează în funcție de cât de mare este permisă să fie întârzierea prin router. În mod normal este bine ca max_{th} să fie de 2 ori mai mare decât min_{th}.

3.2.1.3 Calculul probabilității de marcare

Probabilitea de marcare este calculată în 2 etape. În prima etapă se calculează probabilitatea inițială de marcare a pachetului p_b care este o funcție liniară care depinde de media cozii:

$$p_b = max_p * (avg - min_{th}) / (max_{th} - min_{th})$$

Parametrul max_p reprezintă valoarea maximă a probabilității de aruncare și se atinge atunci când media este egala cu pragul maxim. De obicei max_p este ales mai mic decât 0.1.

Probabilitatea finală crește odată cu creșterea numărul de pachete de la ultimul pachet aruncat, count :

$$p_a = p_b / (1 - count * p_b)$$

Probabilitatea de marcare a pachetelor în funcție de valoarea medie a cozii este prezentată în graficul următor:

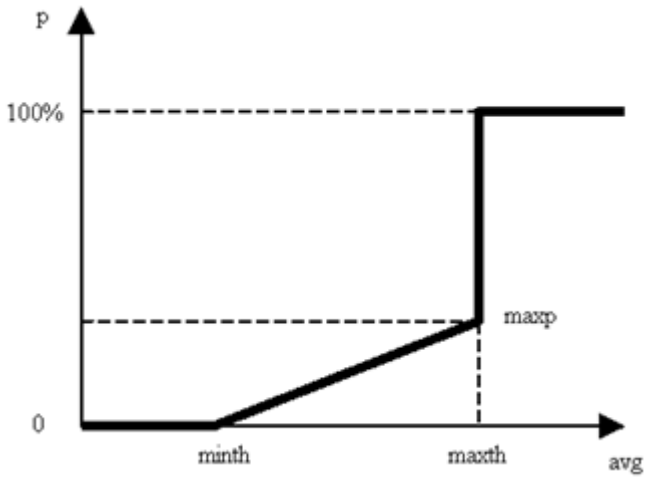

Figura 3.1 "Probabilitatea de marcare"

Atunci când lungimea medie a cozii se află de între 0 şi pragul minim probabilitatea de aruncare este nulă şi toate pachetele sunt acceptate de ruter. Între pragul minim şi pragul maxim probabilitatea de aruncare variază liniar între 0 şi valoarea prag max_p. După ce media depăşeşte pragul setat ca maxim, probabilitatea de aruncare atinge unitatea şi în acest caz toate pachetele vor fi aruncate.

3.2.2 ARED

Cum întârzierile sunt o componentă majoră a calităţii serviciilor este de dorit ca întârzierile prin rutere să poată fi estimate. Deşi RED reuşeşte să atingă o rata de transfer mare cu întârzieri cât mai mici, lungimea medie a cozii depinde de nivelul de congestie şi de parametrii RED prin urmare nu este deloc predictibilă şi întârzierele prin rutere nu pot fi estimate. Pentru a realiza acest lucru RED ar trebui să poată să îşi adapteze parametrii la condiţiile traficului.

În [9] este propusă adaptarea parametrului max_p pentru a păstra lungimea medie a cozii între pragurile min_{th} şi max_{th} după algoritmul următor:

```
pentru fiecare interval măsurat în secunde:
     if (avg > target şi max_p <= 0.5)
          creşte maxp: max_p +=  α
     elseif (avg < target şi max_p >= 0.01)
          scade max_p: max_p *= β
```

Variabile:

interval : 0.5 secunde
target : valoare dorită pentru avg, [min_{th} + 0.4 * (max_{th} - min_{th}), min_{th} + 0.6 * (max_{th} - min_{th})]

α : factor de creştere, min(0.01, max_p/4)
β : factor de scădere, 0.9

Parametrul max_p nu este adaptat doar pentru a ţine lungimea medie a cozii între pragurile min_{th} şi

max_{th} pentru a o ține într-un interval egal departat de cele 2 de praguri: min_{th} și max_{th}. Adaptarea parametrului max_p se realizează încet și în intervale de timp mai mari decât RTT și folosește politica AIMD.

3.2.2.1 Setarea parametrilor

Valoarea maximă a lui max_p este setată la valoarea de 0.5 pe motivul că nu are mai are rost să optimizezi RED atunci când ai pierderi mai mari de 50% iar pragul minim pentru max_p este 0.01 și a fost setat având în vedere scenariile în care pierderile de pachete sunt reduse iar performanța algoritmului a fost bună în [9].

Pentru setarea parametrilor α și β se dorește ca o singură modificare a parametrului max_p să nu modifice lungimea medie a cozii de la o valoare mai mare decât intervalul țintă la o valoare sub intervalul țintă, respectiv de la o valoare mai mică decât intervalul țintă la o valoare deasupra intervalului țintă.

Dacă presupunem că atunci când parametrul max_p este schimbat probabilitatea p de aruncare a pachetelor nu se schimbă iar lungimea medie a cozii ține cont de noul max_p pentru $p < max_p$ atunci când max_p crește cu α, lungimea medie a cozii scade cu :

$$\frac{\alpha}{(max_p + \alpha)} \quad \frac{p}{max_p} \quad (\quad max_{th} - min_{th} \quad)$$

Atât timp cât această valoare este mai mică decât 0.2 ($max_{th} - min_{th}$), lungimea media a cozii nu ar trebui să mute de la o valoare superioară intervalului țintă la o valoare inferioară pentru o singură adaptare [9]. Astfel valoarea pentru α trebuie să fie mai mică decât un sfert din valoare max_p. O analiză similară arată că β trebuie ales mai mare decât 0.83.

3.3.3 BLUE

Algoritmul BLUE funcţionează la fel ca și RED, aruncă pachete în mod aleator înainte ca bufferul ruterului să devină plin. Spre deosebire de RED care folosește lungimea medie a cozii pentru a calcula probabilitatea de aruncare, BLUE foloseste istoria utilizării legăturii și a pierderilor de pachete.

Algoritmul foloseste o singură probabilitate p_m pentru aruncarea pachetelor. Evenimentele în care pachetele sunt aruncate cresc probabilitatea p_m iar atunci când bufferul este gol sau legătura este în repaus probabilitatea p_m scade. În acest mod algoritmul învață rata corectă cu care trebuie să arunce pachetele.

Pseudocodul algoritmului este:

```
dacă se pierde un pachet:
     dacă ( ( now – last_update ) > freeze_time )
          pm += pm + δ1
```

```
          last_update = now
dacă legătura este în repaos:
     dacă ( ( now – last_update ) > freeze_time )
          pm += pm - δ2
          last_update = now
```

Din pseudocod se observă că pe lângă probabilitate BLUE foloseste încă 4 parametrii. Primul parametru, last_update, păstrează timpul ultimei modificări. Cel de-al doilea parametru, freeze_time, determină intervalul de timp minim dintre 2 modificări succesive ale probabilității p_m. Acest parametru este folosit pentru a se asigura ca noua valoare a probabilității este în vigoare înainte ca ea să fie modificată din nou.

Parametrii δ_1, δ_2 sunt folosiți pentru a seta cantitatea cu care probabilitatea crește/ scade. În [11] este sugerat că cantitatea cu care probabilitatea crește să fie mult mai mare decât cantitatea cu care probabilitea scade deoarece folosirea ineficientă a legături poate apare atunci când gestiunea congestiei este ori prea conservativă, ori prea agresivă dar pierderile de pachete au loc doar atunci când gestiunea congestiei este prea conservativă.

Capitolul 4

4.1 Modificare RED

În graficul probabilității de marcare de pachete în funcție de lungimea medie a cozii pentru algoritm RED se poate observa 2 zone în care probabilitatea de aruncare a pachetelor este diferită de zero. Prima zonă este între min_{th} și max_{th} unde probabilitatea crește liniar de la 0 la max_p iar cea de-a doua zonă este între max_{th} și limita bufferului unde probabilitatea de marcare este egală cu unitatea și toate pachetele sunt aruncate.

Ținând cont de zonele de aruncare de pachete de mai sus am propus modificarea probabilității de marcare cu o funcție de tip sigmoid :

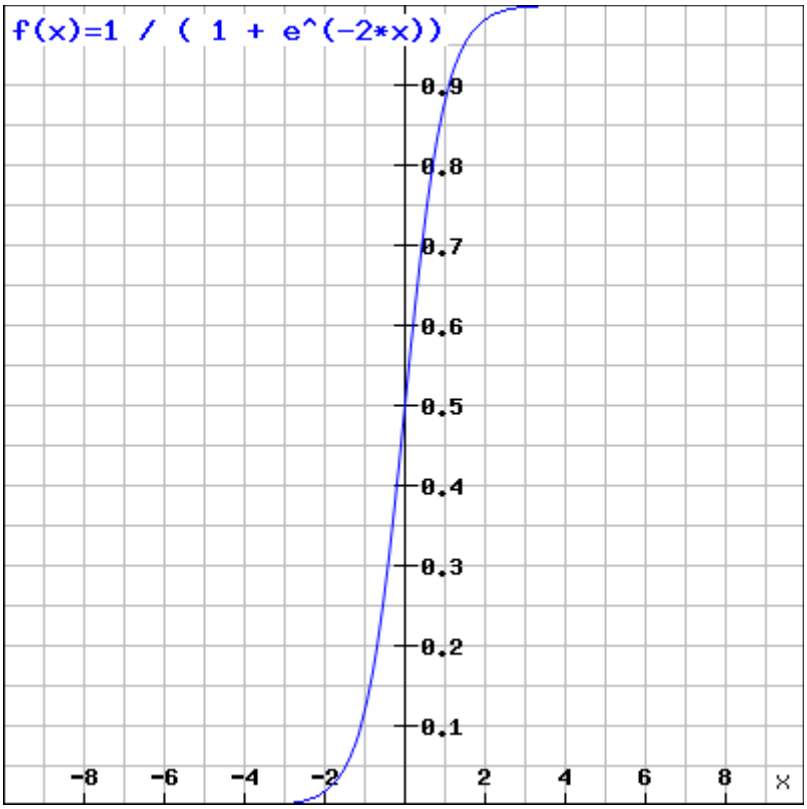

Figura 4.1 "Funcție sigmoid"

Motivul pentru care am ales această funcție este că graficul ei poate fi împărțit în 2 lucru care ne poate oferi două modalități de marcare a pachetelor. Dacă luam partea din stânga și o privim din punctul de vedere al unei probabilități de aruncare putem observa că ea este puțin agresivă pe intervalul [-3, 0] iar în rest este egală cu zero. Partea din dreapta a graficului este foarte agresivă probabilitatea de aruncare fiind între 0.5 și 1.

Dorința mea a fost de a crea un mecanism de aruncare care să evite cât mai mult aruncarea pachetelor atunci când bufferul este departe de a fi plin și doar atunci când bufferul tinde să fie plin să mecanismul să acționeze agresiv prin aruncarea de pachete. Pentru a implementa acest mecanism am împărțit dimensiunea bufferului în 2 și am atribuit-o unei părți a funcției.

Pentru partea bufferului de la 0 la 0.9 din dimensiunea bufferului am folosit funcţia pentru valorile lui x < 0, iar pentru ultima zecime a bufferului am folosit funcţia pentru valori ale lui x >= 0.

Prin acest lucru am reuşit să realizez un mecanism de aruncare care să evite aruncarea pachetelor pentru o mare parte nivelurilor de încărcare ale bufferului iar pentru cazurile în care încărcarea bufferului se apropie de limită să acţioneze foarte agresiv prin aruncarea de pachete pentru a informa conexiunile că ruterul este la limită şi ar trebui să îşi reducă ratele de transfer.

Pentru a mapa funcţia sigmoid pe intervalul dat de dimensiune bufferului am presupus că intervalul pe care ia valori este între -10 şi 10 iar fiecare nivel de încărcare al routerului este transpus pe acest interval pentru a calcula probabilitatea de marcare.

Capitolul 5

5.1 Mediul de lucru

Pentru compararea celor patru algoritmi : RED, ARED, BLUE şi REDM am folosit NS3 care este un simulator de reţele folosit atât pentru cercetare cât şi în scop educaţional.

Urmărind paşii din [11] am reuşit configurez Eclipse Kepler pentru dezvoltatori C/C++ să lucreze cu librăria NS3 pentru a putea recompila şi rula programul într-un mediu IDE pe Ubuntu Linux 12.04.

Pentru realizarea graficelor am folosit Gnuplot care este un program de generare a graficelor în 2D sau 3D din linie de comandă.

5.2 Modificarea librăriei NS3

Pentru a putea compara cei patru algoritmi am fost nevoit să modific libraria NS3 care nu avea împlementat decât algoritmul RED.

Astfel conform cu pseudocodul ARED am introdus noi variable necesare pentru a rula ARED. Variabila *interval* care spune cât de des se face adaptarea am setat-o la valoarea 0.5 secunde, constantele de creştere şi scădere a pragului probabilităţii de aruncare max_p, notate cu α şi β le-am setat la valorile 0.01 şi 0.9, iar valorile de prag pentru max_p au fost setate la valorile 0.01 şi 0.5 exact ca în pseudocod. Totodată pentru a putea alege între RED şi ARED am introdus o constantă de configurare booleană pentru adaptare care atunci când este setată va acţiona algoritmul ARED.

Pe lângă introducerea de noi variabile am adăugat o funcţie care să poată să fie apelată odată la *interval* secunde. Funcţia este prezentată în Anexa 3 şi este apelată prin intermediul unei variabile de tip eveniment care planifică apelarea ei la fiecare *interval* secunde, acestă variabilă eveniment fiind setată doar dacă variabila de adaptare este setată.

De asemenea pentru apelarea variantei de RED modificat am realizat o constantă de configurare booleană ca şi pentru varianta adaptivă. Dacă această constantă este setată adevărat în calculul probabilităţii va apela funcţia descrisă în Anexa 4 care mapează funcţia sigmoid prezentată în capitolul precendent pe dimensiunea bufferului folosit.

În cazul algoritmului BLUE am fost nevoit să implementez de la zero algoritmul prin adăugare unei clase header şi unei clase sursă urmărind articolele [10], [12], [13], [14]. Codul este prezentat în Anexa 1.

5.3 Crearea Topologiei

După implementarea algoritmilor mi-am imaginat o topologie dinamică ca în figura 5.1 în care să pot varia numărul de conexiuni și să pot trece ușor de la un algoritm la altul. Și acest lucru a posibil în NS3 iar codul de generare a simulării este prezentat în Anexa 2.

Topologia folosită pentru a compara cei patru algoritmi a fost următoarea:

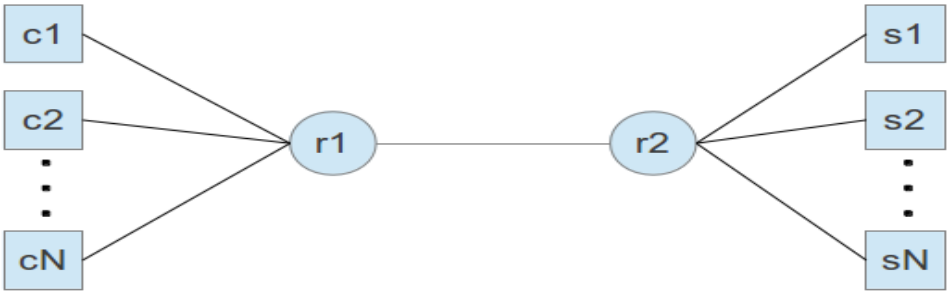

Figura 5.1 “Topologia” [15]

Pentru legăturile c1-r1, c2-r1, …, cN – r1, r2-s1, r2-s2, …, r2-sN am setat capacitatea de transfer la 10Mbps cu întârzieri de propagare de 5ms iar pentru legătura r1- r2 am setat capacitatea de transfer la 5Mbps cu întârzieri de propagare de 10ms.

Dimensiunea bufferului a fost setata la 60 de pachete iar pragul minim și maxim la 12, respectiv 48.

Pentru a observa cum reacționează algoritmii în cadrul congestiei am variat N de la 2 la 10. Pe măsură ce N crește congestia devine din ce mai puternică iar pierderile de pachete sunt din ce în ce mai frecvente.

Pentru N = 2 lungimea medie a cozii pentru cei 4 algoritmi are graficul următor:

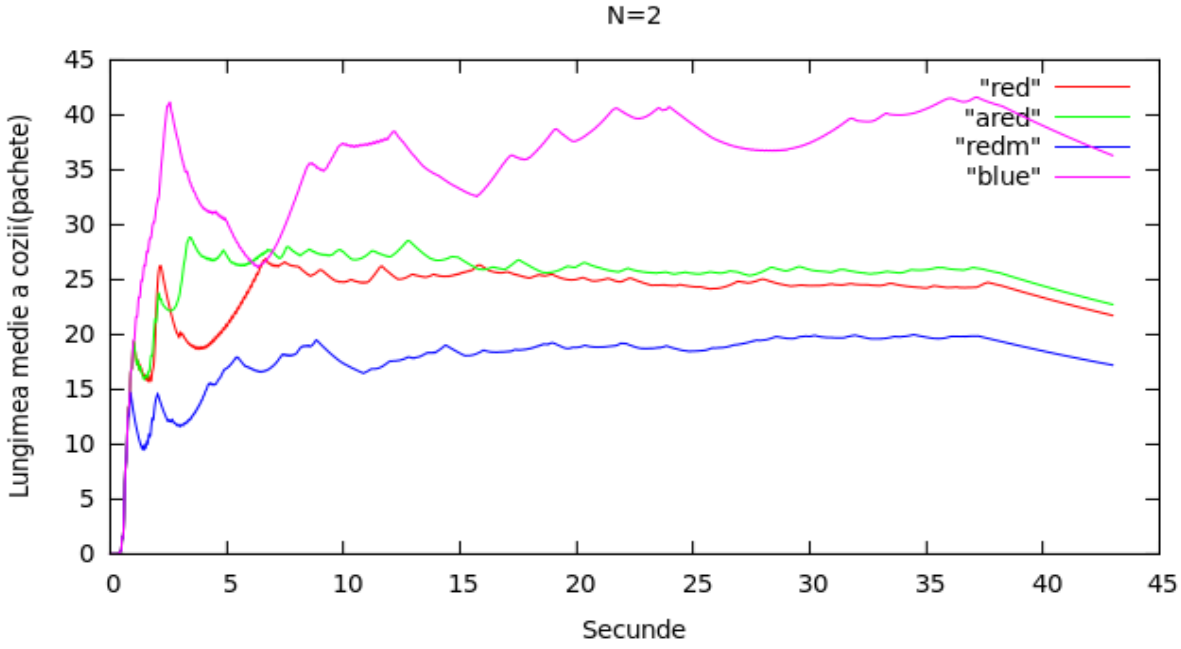

Figura 5.2 “Lungimea medie a cozii pentru N = 2”

Algoritm	Procentaj pierderi pentru r1	Procentaj pierderi pentru r2
RED	3.4 %	1.33 %
ARED	2.5 %	1.82 %
REDM	0.59 %	0.49 %
BLUE	0.31 %	0.16 %

Tabel 5.1 “Procentaj pierderi pachete pentru N = 2”

Pentru N = 4 am obținut următorul grafic pentru lungimea medii a cozii:

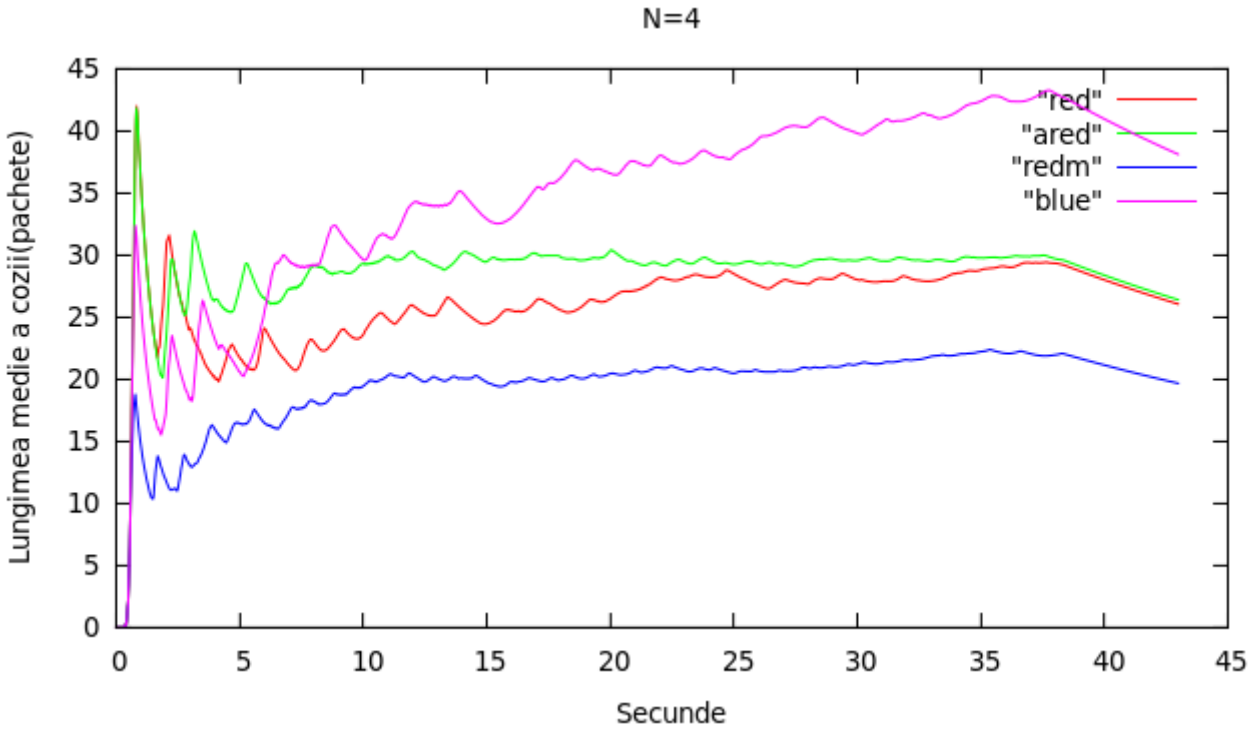

Figura 5.3 “Lungimea medie a cozii pentru N = 4”

Algoritm	Procentaj pierderi pentru r1	Procentaj pierderi pentru r2
RED	11.9 %	9.3 %
ARED	7.99 %	5.86 %
REDM	1.48 %	1.18 %
BLUE	0.89 %	0.9 %

Tabel 5.2 “Procentaj pierderi pachete pentru N = 4”

Pentru N = 6 am obținut:

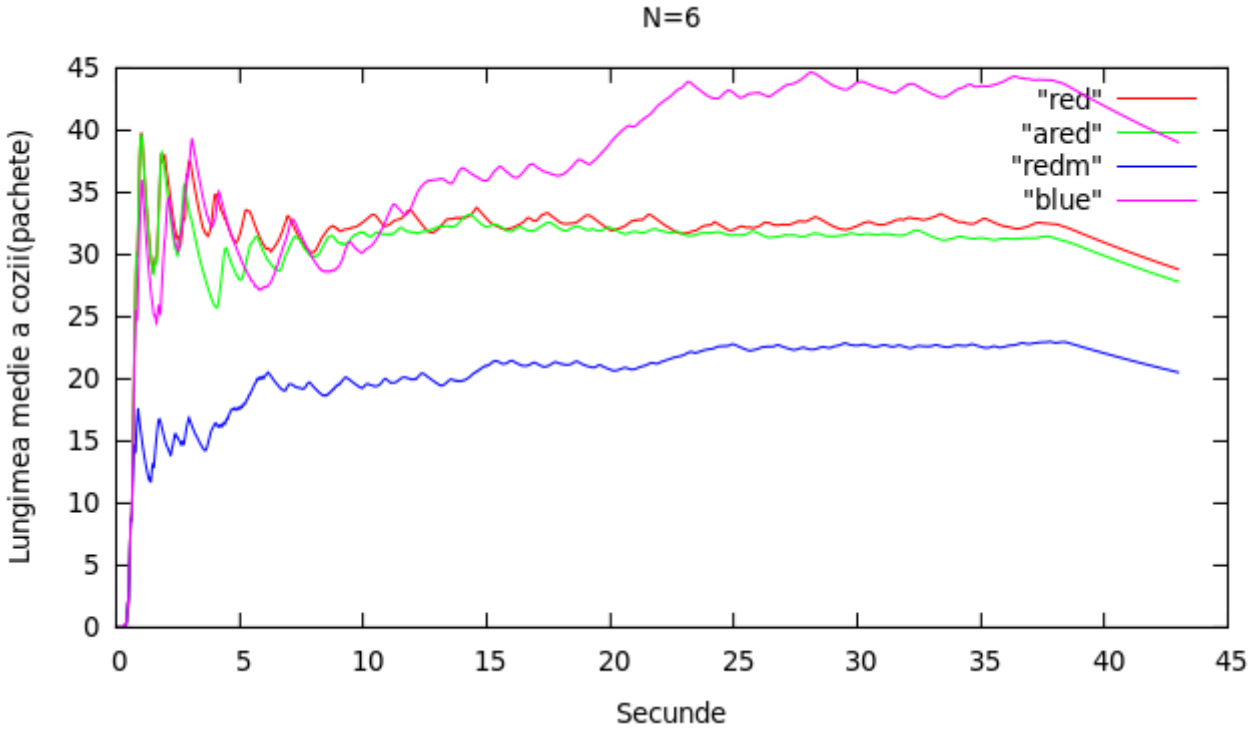

Figura 5.4 “Lungimea medie a cozii pentru N = 6”

Algoritm	Procentaj pierderi pentru r1	Procentaj pierderi pentru r2
RED	18.66 %	18.55 %
ARED	10.2 %	8.17 %
REDM	2.36 %	2.11 %
BLUE	1.38 %	1.35 %

Tabel 5.3 “Procentaj pierderi pachete pentru N = 6”

Pentru N = 8 am obținut:

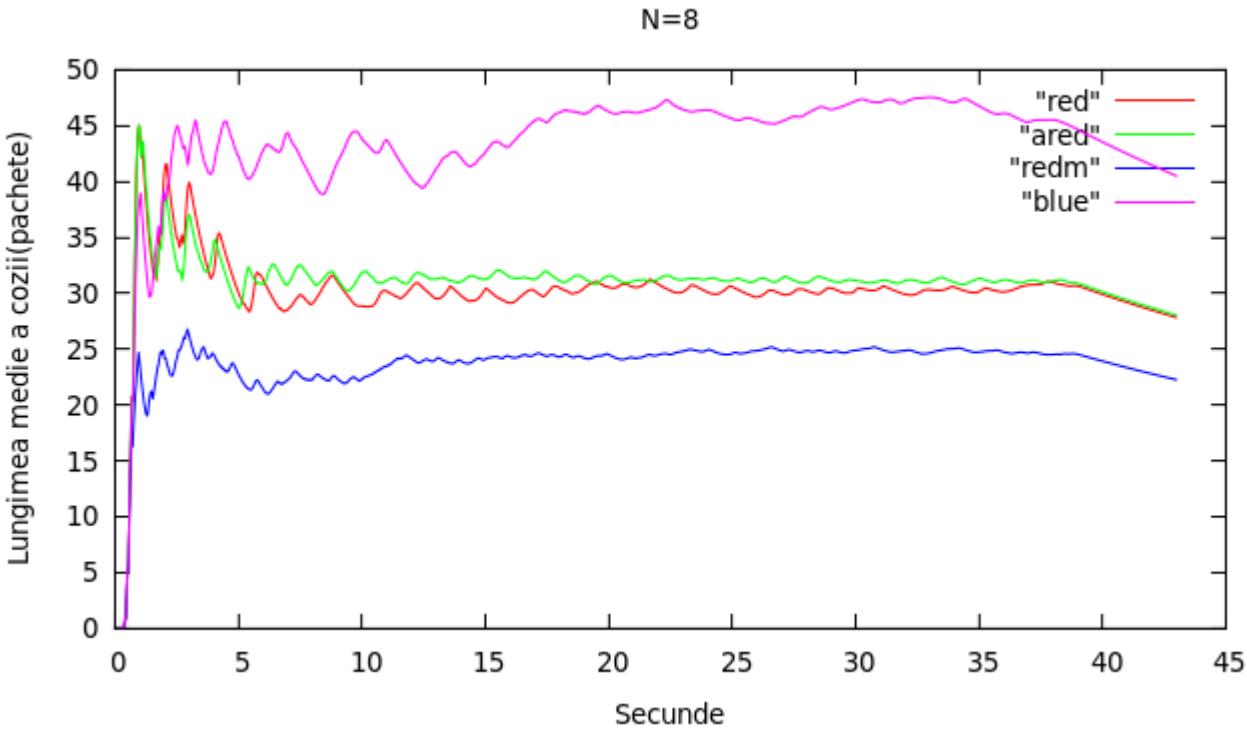

Figura 5.5 “Lungimea medie a cozii pentru N = 8”

Algoritm	Procentaj pierderi pentru r1	Procentaj pierderi pentru r2
RED	24.87 %	24.18 %
ARED	18.21 %	15.65 %
REDM	3.3 %	3.53 %
BLUE	2.08 %	2.29 %

Tabel 5.4 “Procentaj pierderi pachete pentru N = 6”

Pentru N = 10 am obţinut:

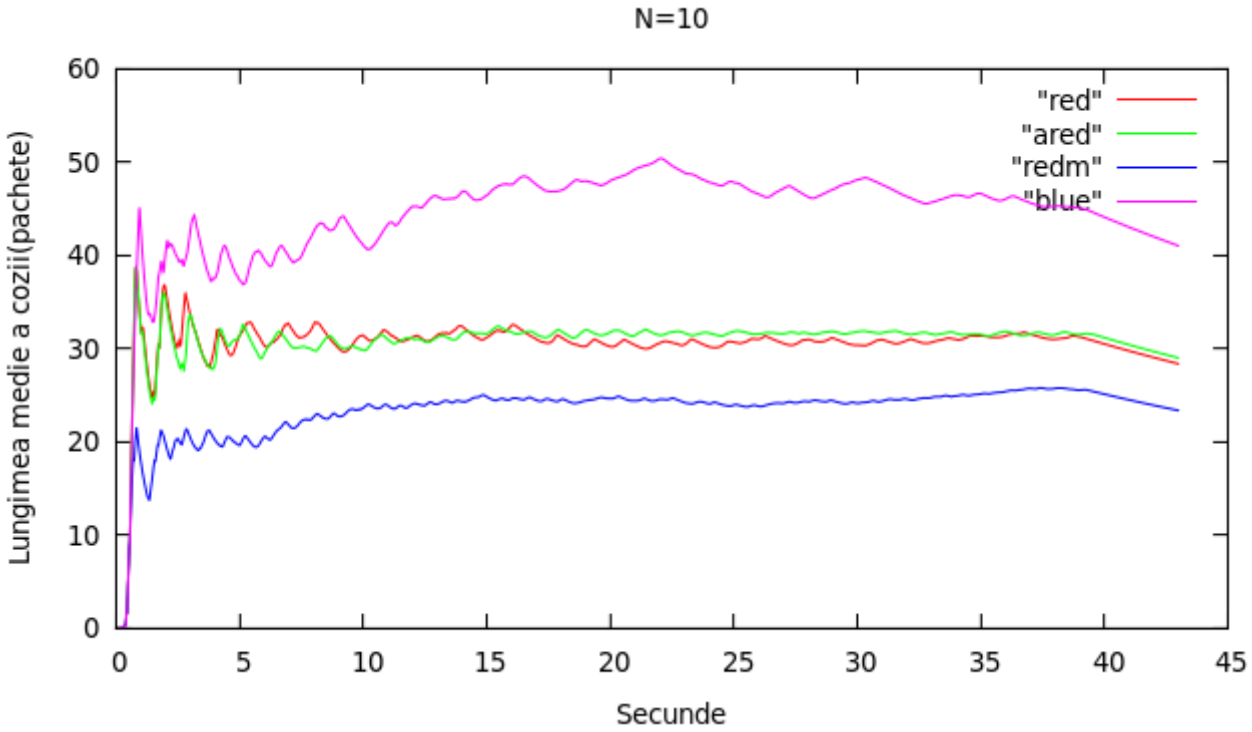

Figura 5.6 “Lungimea medie a cozii pentru N = 10”

Algoritm	Procentaj pierderi pentru r1	Procentaj pierderi pentru r2
RED	25.11 %	27.27 %
ARED	15.79 %	18.48 %
REDM	3.94 %	3.57 %
BLUE	2.08 %	2.57 %

Tabel 5.5 “Procentaj pierderi pachete pentru N = 10”

Din cele 5 grafice de mai sus putem observa că din punctul de vedere al lungimii cozii medii cel mai bun este algoritmul RED modificat reusind să păstreze lungimea cozii cu 16% mai mică decât varianta RED nemodifică rezultând în întârzieri mai mici. Algoritmul BLUE deşi are în toate cazurile o lungime a cozii mai mare deci întârzieri mai mari prin ruter, procentajul de pierderi este cel mai mic urmat îndeaproape de varianta RED modificată care spre deosebire de varianta RED nemodicată are un procent de pierderi mai mic cu aproape un ordin de magnitudine pentru un mediu puternic congestionat.

Pentru a observa comportamentul celor 4 algoritmi în funcţie de numărul de conexiuni am ales momentele de timp 4, 16 si 32 secunde ale simulării. Am obţinut următoarele grafice:

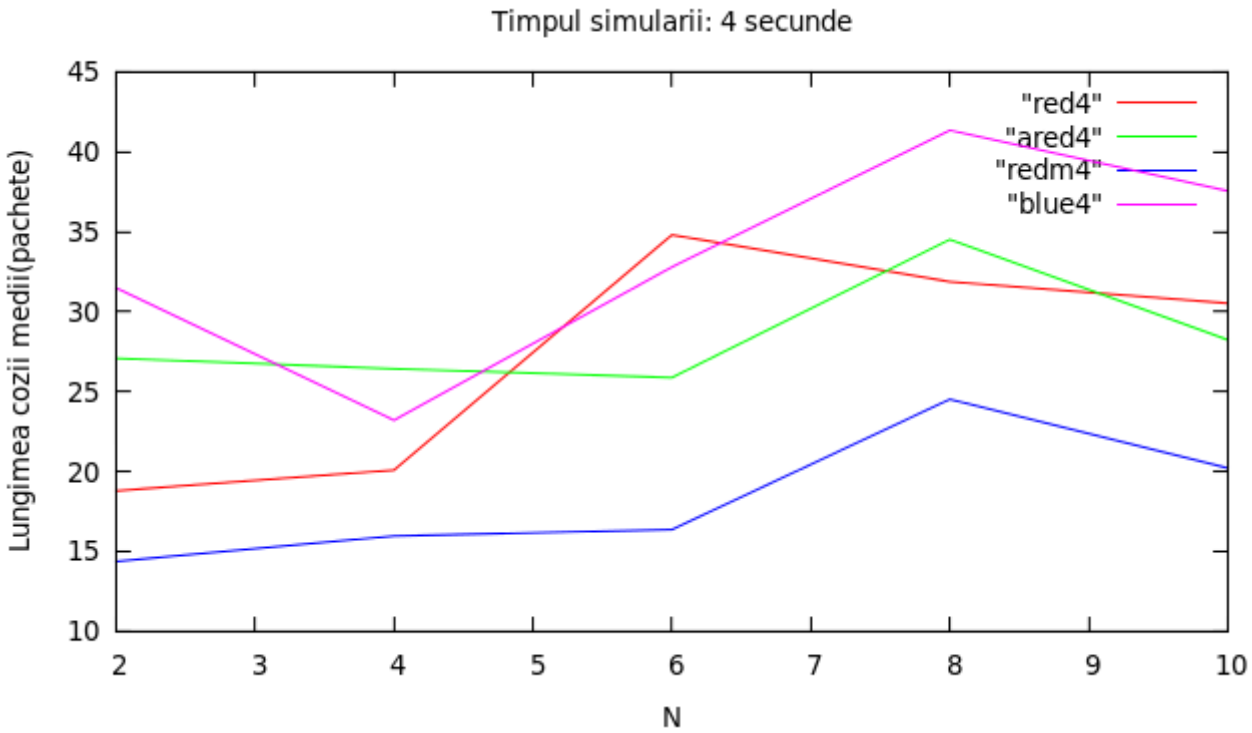

Figura 5.7 “Lungimea medie a cozii la momentul de timp 4 secunde”

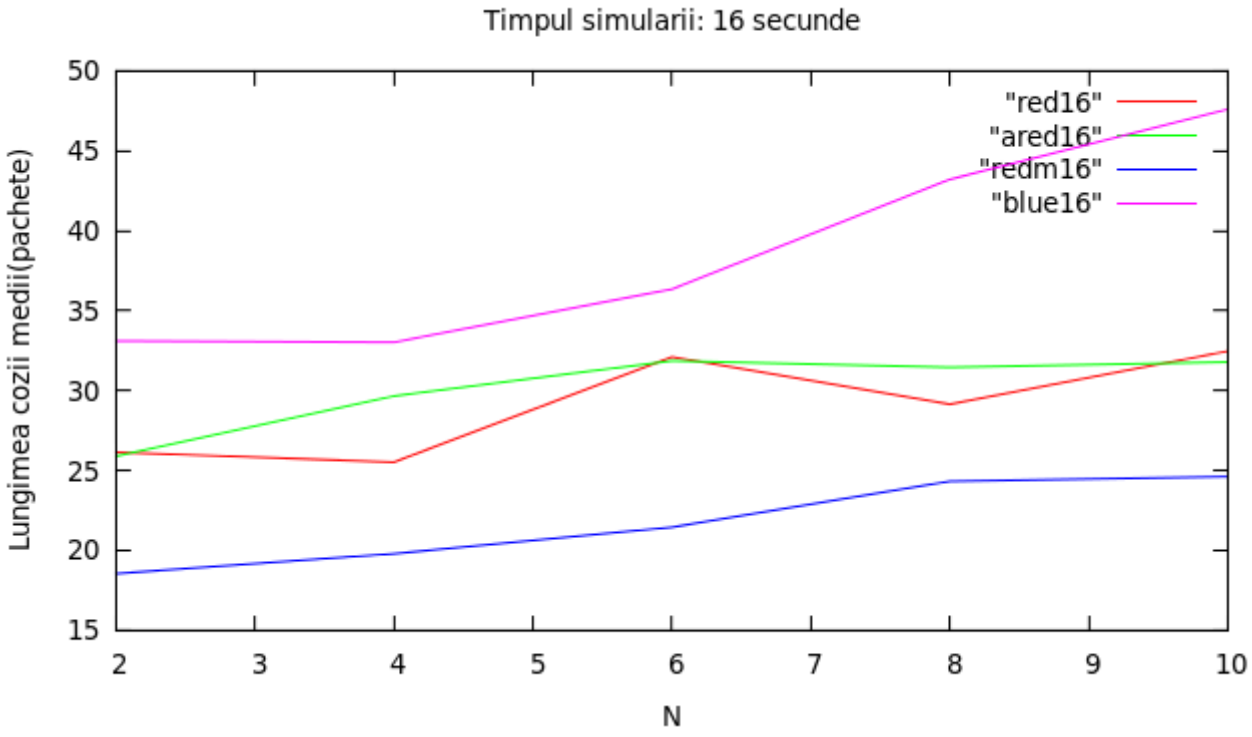

Figura 5.8 “Lungimea medie a cozii la momentul de timp 4 secunde”

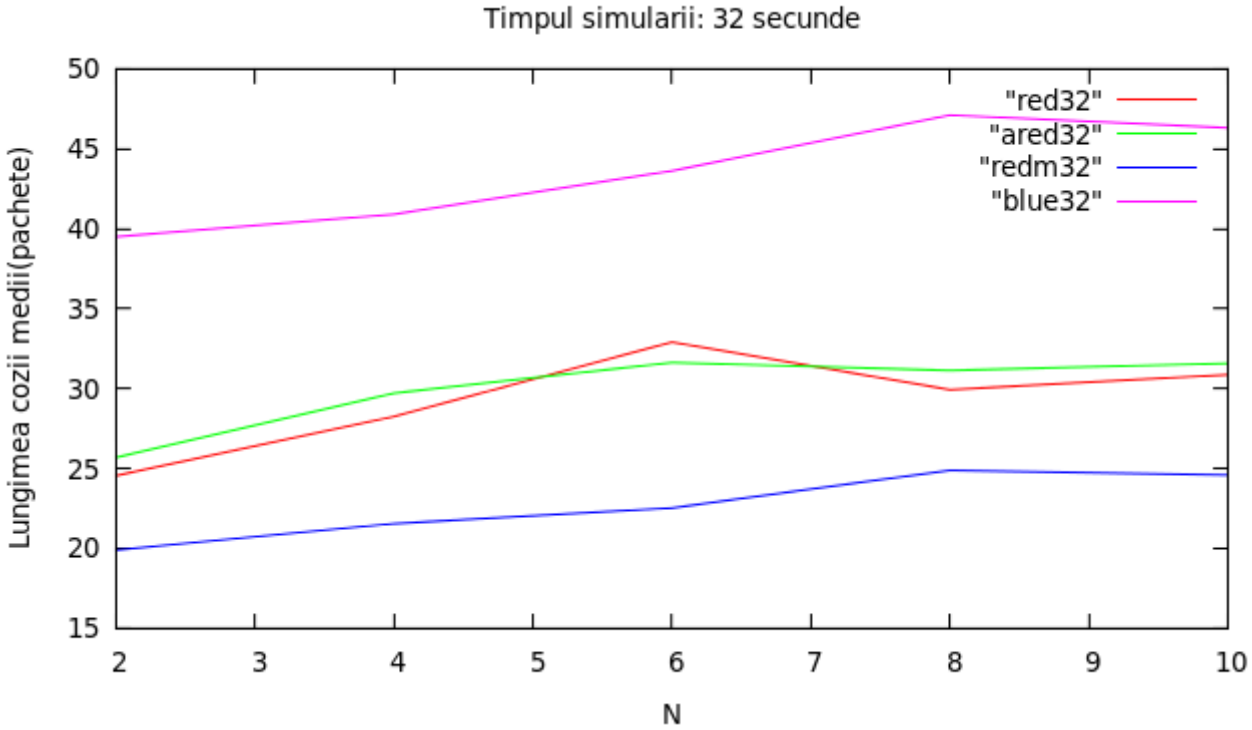

Figura 5.9 “Lungimea medie a cozii la momentul de timp 4 secunde”

Din graficele pentru N = 4, N = 6 putem observa că pentru un mediu puțin congestiv în etapa de început a simulării început întârzierile pentru algoritmul modificat sunt mult reduse deoarece reușește să păstreze o lungime medii a cozii cu aproape 40% mai mică decât ceilalți algoritmi(cazul N = 6). Pentru celelalte valori ale lui N diferența este mai mică, lungimea medie a cozii păstrată de algoritmul modificat este mai mică cu valori cuprinse în intervalul 15% - 20% decât varianta lui originală.

Din observarea celor graficelor celor 3 momente ale simulării putem observa că varianta de RED modificat are cea mai stabilă evoluție din punctul de vedere al numărului de conexiuni. Din graficul simulării la 4 secunde putem observa că mai ales la începutul realizării conexiunilor varianta modificată reușeste să aibă cea mai robustă evoluție fără a varia puternic precum algoritmul BLUE. Odată cu avansarea în timp a simulării variația în funcție de numărul de conexiuni devine din ce în ce mai mică pentru toți algoritmii dar dintre toți algoritmii variația cea mai mică este obținută de algoritmul modificat lucru care înseamnă ca pentru algoritmul modificat se poate realiza o estimare a întârzierilor prin ruter mult mai exactă.

Concluzii

Domeniul AQM este un domeniu extrem de vast și destul de complex și extrem de necesar pentru timpul în care trăim. Articolele și publicațiile care apar sunt numeroase dar noi tehnici AQM sunt puțin implemente în realitate. Cel mai popular algoritm implementat rămâne în continuare RED deoarece este cel mai simplu și mai robust. Cu toate acestea el poate fi îmbunătățit pentru un control mai bun al congestiei și am demonstrat acest lucru prin experimentul realizat.

Printr-o simplă modificare a modului de calculare al probabilității am reușit să implementez o variantă de RED mai robustă în fața congestiei reducând pierderile de pachete cu un ordin de magnitude oferind rată de transfer mai mare și întârzieri mai mici datorate unei lungimi a cozii medii cu cel puțin 16% mai mică decât în cazul RED.

Varianta de RED modificat se apropie de performanţele BLUE, care este un algoritm superior algoritmului RED [10], păstrând o rată de transfer mai mare cu o creștere redusă a întârzierilor. RED modificat spre deosebire de BLUE reduce întârzierile prin păstrarea unei lungimii mai cozii mai mici și se apropie ca performanţe de rata de transfer pe care o oferă BLUE.

Prin acest experiment am extins biblioteca NS3 în ceea ce privește algoritmii AQM dar ceea ce ar avea biblioteca nevoie este de un modul special care să aibă mai multe implementări ale algoritmilor AQM cum are librăria NS2, varianta mai veche.

Pe lângă algoritmii prezentați în această lucrare am mai implementat și RRED [Anexa 5], care este o variantă robustă a lui RED și este prezentată în [16] dar care pentru experimentul de față ar fi avut aceeași performanță ca RED, motiv pentru care l-am exclus. Pe lângă RRED am început să portez FRED pe care l-am găsit în librăria NS2 dar modificările NS3 pentru a implementa FRED sunt mult mai complexe decât în cazul BLUE și ARED și va trebui să înțeleg mult mai bine modul de funcţionare al librăriei.

În final prin acestă lucrare am încercat să atrag atenția unei probleme moderne care stă oarecum în umbră. Această problemă este congestia iar din cauza domeniului de specialitate am studiat acest fenomen strict din perspectiva rețelelor de calculatoare comparând performanţele unor algoritmi cunoscuți și al unui algoritm personal în cadrul unei experiment menit să simuleze un mediu congestiv.

Bibliografie

[1] http://tools.ietf.org/html/rfc896 - accesat la data de 20.06.2014

[2] http://www.mathcs.emory.edu/~cheung/Courses/558a/Syllabus/6-transport/TCP.html - accesat la data de 26.06.2014

[3] http://www.ietf.org/proceedings/49/slides/aaa-8/sld021.htm - accesat la data de 24.06.2014

[4] http://pages.cs.wisc.edu/~mjbrim/personal/osqual/net/JK88 - accesat la data de 26.06.2014

[5] http://www.cse.wustl.edu/~jain/cis788-95/ftp/tcpip_cong/ - accesat la data de 26.06.2014

[6] http://www.cs.sjsu.edu/faculty/pollett/158a.2.09s/Lec04052009.html#(4) - accesat la data de 24.05.2014

[7] http://tools.ietf.org/html/rfc2001 - accesat la data de 20.06.2014

[8] Random Early Detection Gateways for Congestion Avoidance, Sally Floyd and Van Jacobson

[9] Adaptive RED: An Algorithm for Increasing the Robustness of RED's Active Queue Management, Sally Floyd, Ramakrishna Gummadi, and Scott Shenker

[10] The BLUE Active Queue Management Algorithms , Wu-chang Feng, Kang G. Shin, Dilip D. Kandlur, Debanjan Saha

[11] http://www.nsnam.org/wiki/HOWTO_configure_Eclipse_with_ns-3 - accesat la data de 24.06.2014

[12] Analyzing the Performance of Active Queue Management Algorithms, G.F. Ali Ahammed, Reshma Banu

[13] BLUE: Active Queue Management, Sunitha Burri

[14] Evaluation of Queue Management Algorithms, Ningning Hu, Liu Ren, Jichuan Chang

[15] http://reproducingnetworkresearch.wordpress.com/2012/06/05/choosing-the-default-initial congestion-window/ - accesat la data de 17.06.2014

[16] RRED: Robust RED Algorithm to Counter Low-Rate Denial-of-Service Attacks, Changwang Zhang, Jianping Yin, Zhiping Cai, and Weifeng Chen

Anexe

Anexa 1 – Cod BLUE

blue.h

```
#ifndef BLUE_H
#define BLUE_H

#include "drop-tail-queue.h"
#include "ns3/random-variable-stream.h"

namespace ns3 {

class Blue : public DropTailQueue {
  public:
      static TypeId GetTypeId (void);
      Blue();
      virtual ~Blue();

      typedef struct {
            uint32_t unforcedDrop;
            uint32_t forcedDrop;
            uint32_t qLimDrop;
            uint32_t total;
      } Stats;

      Stats GetStats(void);
      Stats stats;


      uint32_t GetQueueSize (void);
      int64_t AssignStreams (int64_t stream);
      bool DoEnqueue(Ptr<Packet>);
      Ptr<Packet> DoDequeue (void);

      void reset(void);
      void inc_marking_prob(void);
      void dec_marking_prob(void);

      double marking_prob_;
      double inc_factor_;
      double dec_factor_;
      double last_update_time_;
      /*double bandwidth_;*/
      uint32_t setECNbit_;
      uint32_t idle_;
      double idletime_;
      double freezetime_;
      /*double ptc_;*/
      uint32_t mean_pktsize_;
      uint32_t drop_front_;
      uint32_t qib_;
      double blue_l_;
      //trace
      uint32_t curq_;
      double marking_prob_trace_;
```

```
  private:
      Ptr<UniformRandomVariable> m_uv;
};

}

#endif
```

blue.cc

```
#include "ns3/log.h"
#include "ns3/enum.h"
#include "ns3/uinteger.h"
#include "ns3/double.h"
#include "ns3/random-variable-stream.h"
#include "ns3/simulator.h"
#include "ns3/abort.h"
#include "blue.h"
#include "ns3/ipv4-header.h"

NS_LOG_COMPONENT_DEFINE ("Blue");

namespace ns3 {

NS_OBJECT_ENSURE_REGISTERED (Blue);

TypeId
Blue::GetTypeId (void)
{
  static TypeId tid = TypeId ("ns3::Blue")
      .SetParent<DropTailQueue> ()
      .AddConstructor<Blue> ()
      .AddAttribute ("dec_factor_", "dec_factor_", DoubleValue (0.0000025),
MakeDoubleAccessor (&Blue::dec_factor_),  MakeDoubleChecker<double>  ())
      .AddAttribute ("inc_factor_", "inc_factor_", DoubleValue (0.000025),
MakeDoubleAccessor (&Blue::inc_factor_), MakeDoubleChecker<double> ())
      .AddAttribute ("freezetime_", "freezetime_", DoubleValue (0.0001),
MakeDoubleAccessor (&Blue::freezetime_), MakeDoubleChecker<double> ())
      .AddAttribute ("marking_prob_", "marking_prob_", DoubleValue (0),
MakeDoubleAccessor (&Blue::marking_prob_), MakeDoubleChecker<double> ())

  ;
  return tid;
}


Blue::Blue() :
      DropTailQueue()
{
      NS_LOG_FUNCTION (this);
      m_uv = CreateObject<UniformRandomVariable> ();
}

int64_t
Blue::AssignStreams (int64_t stream)
{
  NS_LOG_FUNCTION (this << stream);
  m_uv->SetStream (stream);
  return 1;
}
```

```
Blue::~Blue()
{
      NS_LOG_FUNCTION (this);
}

uint32_t
Blue::GetQueueSize (void)
{
        if (DropTailQueue::GetMode () == QUEUE_MODE_BYTES)
          {
            return DropTailQueue::m_bytesInQueue;
          }
        else if (DropTailQueue::GetMode () == QUEUE_MODE_PACKETS)
          {
            return DropTailQueue::m_packets.size ();
          }
        else
          {
            NS_ABORT_MSG ("Unknown RED mode.");
          }
}

void
Blue::inc_marking_prob()
{
      NS_LOG_FUNCTION (this);
      double now = Simulator::Now ().GetSeconds();

      if ( now - freezetime_ > last_update_time_ ) {
                  last_update_time_ = now;
                  marking_prob_ += inc_factor_;
                  if ( marking_prob_ > 1 ) marking_prob_ = 1;
      }

}

void
Blue::dec_marking_prob()
{
      NS_LOG_FUNCTION (this);
      double now = Simulator::Now ().GetSeconds();

      if ( now - freezetime_ > last_update_time_ ) {
            marking_prob_ -= dec_factor_;
            if ( marking_prob_ < 0 ) marking_prob_ = 0;
      }
}

Blue::Stats Blue::GetStats() {
      return stats;
}

bool
Blue::DoEnqueue(Ptr<Packet> p)
{
      NS_LOG_FUNCTION (this);
      bool enqueued = true;

      stats.total++;
```

```
	double u = m_uv->GetValue();

	if ( u <= marking_prob_ ) {
		Drop(p);
		enqueued = false;
		stats.unforcedDrop++;
	} else {
		enqueued = DropTailQueue::DoEnqueue(p);
		if ( !enqueued ) stats.qLimDrop++;
	}

	if ( !enqueued ) inc_marking_prob();

	return enqueued;
}


Ptr<Packet>
Blue::DoDequeue()
{
	NS_LOG_FUNCTION (this);
	Ptr<Packet> p = DropTailQueue::DoDequeue();

	if (p != 0) {
		idle_ = 0;
	} else {
		dec_marking_prob();
		idle_ = 1;
		idletime_ = Simulator::Now ().GetSeconds();
	}

	return p;
}

}
```

Anexa 2 – Cod pentru generarea topologiei

```
#include "ns3/core-module.h"
#include "ns3/network-module.h"
#include "ns3/internet-module.h"
#include "ns3/flow-monitor-module.h"
#include "ns3/point-to-point-module.h"
#include "ns3/applications-module.h"

using namespace ns3;

NS_LOG_COMPONENT_DEFINE ("AqmTests");

uint32_t checkTimes;
double avgQueueSize;

// The times
double global_start_time;
double global_stop_time;
double sink_start_time;
double sink_stop_time;
double client_start_time;
```

```
double client_stop_time;

std::vector<NodeContainer> containers;
std::vector<PacketSinkHelper> sinkhelpers;
std::vector<ApplicationContainer> sinkapps;
std::vector<NetDeviceContainer> devices;
std::vector<Ipv4InterfaceContainer> interfaces;
std::stringstream filePlotQueue;
std::stringstream filePlotQueueAvg;
std::stringstream filePlotFlowThroughput;

uint32_t numberOfNodes = 22;
uint32_t aqm = 4;
// 1 RED
// 2 ARED
// 3 REDM
// 4 BLUE
// 5 FRED

void
CheckQueueSize (Ptr<Queue> queue)
{
  uint32_t qSize = StaticCast<Blue> (queue)->GetQueueSize ();

  avgQueueSize += qSize;
  checkTimes++;

  // check queue size every 1/100 of a second
  Simulator::Schedule (Seconds (0.01), &CheckQueueSize, queue);

  std::ofstream fPlotQueue (filePlotQueue.str ().c_str (), std::ios::out|
std::ios::app);
  fPlotQueue << Simulator::Now ().GetSeconds () << " " << qSize << std::endl;
  fPlotQueue.close ();

  std::ofstream fPlotQueueAvg (filePlotQueueAvg.str ().c_str (), std::ios::out|
std::ios::app);
  fPlotQueueAvg << Simulator::Now ().GetSeconds () << " " << avgQueueSize /
checkTimes << std::endl;
  fPlotQueueAvg.close ();

}

void
BuildAppsTest ()
{
    // SINKs
      uint16_t port = 1;
      for(uint32_t i = ceil(containers.size()/2.0); i < containers.size(); i++)
{
             Address sinkLocalAddress (InetSocketAddress (Ipv4Address::GetAny
(), port));
             sinkhelpers.push_back(PacketSinkHelper("ns3::TcpSocketFactory",
sinkLocalAddress));
             sinkapps.push_back(sinkhelpers[sinkhelpers.size()-1].Install
(containers[i].Get (1)));
             sinkapps[sinkapps.size()-1].Start (Seconds (sink_start_time));
             sinkapps[sinkapps.size()-1].Stop (Seconds (sink_stop_time));
      }

      for(uint32_t i = 0; i < containers.size()/2; i++) {
```

```
            Address sinkLocalAddress (InetSocketAddress (Ipv4Address::GetAny
(), port));
            sinkhelpers.push_back(PacketSinkHelper("ns3::TcpSocketFactory",
sinkLocalAddress));
            sinkapps.push_back(sinkhelpers[sinkhelpers.size()-1].Install
(containers[i].Get (0)));
            sinkapps[sinkapps.size()-1].Start (Seconds (sink_start_time));
            sinkapps[sinkapps.size()-1].Stop (Seconds (sink_stop_time));
      }

    // Connections
      for ( uint32_t i = 0; i < containers.size()/2; i++) {
            OnOffHelper clientHelper ("ns3::TcpSocketFactory", Address ());
            clientHelper.SetAttribute("OnTime", StringValue
("ns3::ConstantRandomVariable[Constant=1]"));
            clientHelper.SetAttribute("OffTime", StringValue
("ns3::ConstantRandomVariable[Constant=0]"));
            clientHelper.SetAttribute("DataRate", DataRateValue (DataRate
("10Mb/s")));
            clientHelper.SetAttribute("PacketSize", UintegerValue (1000));

            ApplicationContainer clientApps;
            AddressValue remoteAddress(InetSocketAddress
(interfaces[interfaces.size() - i - 1].GetAddress (1), port));
            clientHelper.SetAttribute ("Remote", remoteAddress);
            clientApps.Add (clientHelper.Install (containers[i].Get (0)));
            clientApps.Start (Seconds (client_start_time + i%3));
            clientApps.Stop (Seconds (client_stop_time - i%3));
      }

      for ( uint32_t i = ceil(containers.size()/2.0); i < containers.size(); i+
+) {
            OnOffHelper clientHelper ("ns3::TcpSocketFactory", Address ());
            clientHelper.SetAttribute("OnTime", StringValue
("ns3::ConstantRandomVariable[Constant=1]"));
            clientHelper.SetAttribute("OffTime", StringValue
("ns3::ConstantRandomVariable[Constant=0]"));
            clientHelper.SetAttribute("DataRate", DataRateValue (DataRate
("10Mb/s")));
            clientHelper.SetAttribute("PacketSize", UintegerValue (1000));

            ApplicationContainer clientApps;
            AddressValue remoteAddress(InetSocketAddress (interfaces[i -
ceil(containers.size()/2.0)].GetAddress (0), port));
            clientHelper.SetAttribute ("Remote", remoteAddress);
            clientApps.Add (clientHelper.Install (containers[i].Get (1)));
            clientApps.Start (Seconds (client_start_time + i%3 ));
            clientApps.Stop (Seconds (client_stop_time - i%3));
      }
}

int
main (int argc, char *argv[])
{
      //LogComponentEnable ("FRedQueue", LOG_LEVEL_FUNCTION);

  std::string redLinkDataRate = "5Mbps";
  std::string redLinkDelay = "10ms";

  global_start_time = 0.0;
  global_stop_time = 40;
```

```
  sink_start_time = global_start_time;
  sink_stop_time = global_stop_time + 3.0;
  client_start_time = sink_start_time + 0.2;
  client_stop_time = global_stop_time - 2.0;

  NS_LOG_INFO ("Create nodes");
  NodeContainer c;
  c.Create (numberOfNodes);
  for ( uint32_t i = 0; i < c.GetN() / 2 - 1; i ++ )
containers.push_back(NodeContainer (c.Get (i), c.Get (c.GetN() / 2 - 1)));
  containers.push_back(NodeContainer (c.Get (c.GetN() / 2 - 1), c.Get
(c.GetN() / 2)));
  for ( uint32_t i =  c.GetN() / 2 + 1; i < c.GetN() ; i ++ )
containers.push_back(NodeContainer (c.Get (c.GetN() / 2), c.Get (i)));

  Config::SetDefault ("ns3::TcpL4Protocol::SocketType", StringValue
("ns3::TcpReno"));
  // 42 = headers size
  Config::SetDefault ("ns3::TcpSocket::SegmentSize", UintegerValue (1000 - 42));
  Config::SetDefault ("ns3::TcpSocket::DelAckCount", UintegerValue (1));
  GlobalValue::Bind ("ChecksumEnabled", BooleanValue (false));

  uint32_t meanPktSize = 500;

  if ( aqm!= 4 && aqm != 5 ) {
        Config::SetDefault ("ns3::RedQueue::Mode", StringValue
("QUEUE_MODE_PACKETS"));
        Config::SetDefault ("ns3::RedQueue::MeanPktSize", UintegerValue
(meanPktSize));
        Config::SetDefault ("ns3::RedQueue::Wait", BooleanValue (true));
        Config::SetDefault ("ns3::RedQueue::Gentle", BooleanValue (false));
        Config::SetDefault ("ns3::RedQueue::QW", DoubleValue (0.002));
        Config::SetDefault ("ns3::RedQueue::MinTh", DoubleValue (12));
        Config::SetDefault ("ns3::RedQueue::MaxTh", DoubleValue (48));
        Config::SetDefault ("ns3::RedQueue::QueueLimit", UintegerValue (60));
  }

  if ( aqm != 2 ) Config::SetDefault ("ns3::RedQueue::Adapt", BooleanValue
(false));
  else Config::SetDefault ("ns3::RedQueue::Adapt", BooleanValue (true));
  Config::SetDefault ("ns3::RedQueue::LinkBandwidth",  DataRateValue (DataRate
("5Mbps")));
  Config::SetDefault ("ns3::RedQueue::LinkDelay",   TimeValue (MilliSeconds
(50)));
  if ( aqm != 3 ) Config::SetDefault ("ns3::RedQueue::AnotherP", BooleanValue
(false));
  else Config::SetDefault ("ns3::RedQueue::AnotherP", BooleanValue (true));

  NS_LOG_INFO ("Install internet stack on all nodes.");
  InternetStackHelper internet;
  internet.Install (c);

  NS_LOG_INFO ("Create channels");
  PointToPointHelper p2p;

  for( uint32_t i = 0; i < containers.size(); i++ ) {
        if ( i!=c.GetN() / 2 - 1 ) {
              p2p.SetQueue ("ns3::DropTailQueue");
              p2p.SetDeviceAttribute ("DataRate", StringValue ("10Mbps"));
              p2p.SetChannelAttribute ("Delay", StringValue ("5ms"));
        } else {
```

```
              if ( aqm !=4 ) p2p.SetQueue ("ns3::RedQueue", "LinkBandwidth",
StringValue (redLinkDataRate), "LinkDelay", StringValue (redLinkDelay));
              else p2p.SetQueue ("ns3::Blue" );
              p2p.SetDeviceAttribute ("DataRate", StringValue
(redLinkDataRate));
              p2p.SetChannelAttribute ("Delay", StringValue (redLinkDelay));
        }

        devices.push_back( p2p.Install (containers[i]) ) ;
  }

  NS_LOG_INFO ("Assign IP Addresses");
  Ipv4AddressHelper ipv4;

  std::ostringstream oss;
  std::string s;
  const char *ip = s.c_str();

  for( uint32_t i = 0; i < devices.size(); i++ ) {
        std::ostringstream oss;
        oss << (i + 1);
        s = "10.1." + oss.str() + ".0";
        ip = s.c_str();
        ipv4.SetBase (ip, "255.255.255.0");
        interfaces.push_back( ipv4.Assign (devices[i]) );
  }

  // Set up the routing
  Ipv4GlobalRoutingHelper::PopulateRoutingTables ();

  BuildAppsTest ();

  FlowMonitorHelper flowmon;
   Ptr<FlowMonitor> monitor = flowmon.InstallAll();

  std::stringstream stmp;

  //write for plot
  filePlotQueue << "./" << "red-queue.plotme";
  filePlotQueueAvg << "./" << "red-queue_avg.plotme";

  remove (filePlotQueue.str ().c_str ());
  remove (filePlotQueueAvg.str ().c_str ());
  Ptr<PointToPointNetDevice> nd = StaticCast<PointToPointNetDevice>
(devices[c.GetN() / 2 - 1].Get (0));
  Ptr<Queue> queue = nd->GetQueue ();
  Simulator::ScheduleNow (&CheckQueueSize, queue);

  Simulator::Stop (Seconds (sink_stop_time));
  Simulator::Run ();

  stmp << "./red.flowmon";
  monitor->CheckForLostPackets();

  if ( aqm != 4 ) {
      Ptr<PointToPointNetDevice> nd = StaticCast<PointToPointNetDevice>
(devices[c.GetN() / 2 - 1].Get (0));
      RedQueue::Stats st = StaticCast<RedQueue> (nd->GetQueue ())->GetStats ();
      std::cout << "\t " << st.unforcedDrop << " drops due prob mark" <<
std::endl;
      std::cout << "\t " << st.forcedDrop << " drops due hard mark" <<
```

```
std::endl;
      std::cout << "\t " << st.qLimDrop << " drops due queue full" << std::endl;
      std::cout << "\t " << st.total << " total pachets" << std::endl;
      std::cout << "\t " << (st.unforcedDrop + st.forcedDrop + st.qLimDrop ) /
(double)st.total << "loss rate" << std::endl;
      nd = StaticCast<PointToPointNetDevice> (devices[c.GetN() / 2 - 1].Get
(1));
      st = StaticCast<RedQueue> (nd->GetQueue ())->GetStats ();
      std::cout << "\t " << st.unforcedDrop << " drops due prob mark" <<
std::endl;
      std::cout << "\t " << st.forcedDrop << " drops due hard mark" <<
std::endl;
      std::cout << "\t " << st.qLimDrop << " drops due queue full" << std::endl;
      std::cout << "\t " << st.total << " total pachets" << std::endl;
      std::cout << "\t " << (st.unforcedDrop + st.forcedDrop + st.qLimDrop ) /
(double)st.total << "loss rate" << std::endl;
  } else {
      Ptr<PointToPointNetDevice> nd = StaticCast<PointToPointNetDevice>
(devices[c.GetN() / 2 - 1].Get (0));
      Blue::Stats st = StaticCast<Blue> (nd->GetQueue ())->GetStats ();
      std::cout << "\t " << st.unforcedDrop << " drops due prob mark" <<
std::endl;
      std::cout << "\t " << st.forcedDrop << " drops due hard mark" <<
std::endl;
      std::cout << "\t " << st.qLimDrop << " drops due queue full" << std::endl;
      std::cout << "\t " << st.total << " total pachets" << std::endl;
      std::cout << "\t " << (st.unforcedDrop + st.forcedDrop + st.qLimDrop ) /
(double)st.total << "loss rate" << std::endl;
      nd = StaticCast<PointToPointNetDevice> (devices[c.GetN() / 2 - 1].Get
(1));
      st = StaticCast<Blue> (nd->GetQueue ())->GetStats ();
      std::cout << "\t " << st.unforcedDrop << " drops due prob mark" <<
std::endl;
      std::cout << "\t " << st.forcedDrop << " drops due hard mark" <<
std::endl;
      std::cout << "\t " << st.qLimDrop << " drops due queue full" << std::endl;
      std::cout << "\t " << st.total << " total pachets" << std::endl;
      std::cout << "\t " << (st.unforcedDrop + st.forcedDrop + st.qLimDrop ) /
(double)st.total << "loss rate" << std::endl;
  }

  filePlotFlowThroughput << "./" << "flow.plotme";
  remove (filePlotFlowThroughput.str ().c_str ());

  double avgThroughput = 0;

   std::map<FlowId, FlowMonitor::FlowStats> stats = monitor->GetFlowStats ();
   for (std::map<FlowId, FlowMonitor::FlowStats>::const_iterator i = stats.begin
(); i != stats.end (); ++i)
     {
            std::ofstream fPlotFlowThroughput (filePlotFlowThroughput.str
().c_str (), std::ios::out|std::ios::app);
            fPlotFlowThroughput <<  i->first << " " << i->second.rxBytes * 8.0 /
(i->second.timeLastRxPacket.GetSeconds() - i-
>second.timeFirstTxPacket.GetSeconds())/1024/1024 << std::endl;
            fPlotFlowThroughput.close ();
      }

  monitor->SerializeToXmlFile("flow.flowmon", true, true);

  Simulator::Destroy ();
```

```
  return 0;
}
```

Anexa 3 – Modificare RED pentru a include ARED

Adaugarea funcţiei şi parametrilor pe care îi conţine:

```
void
RedQueue::AdaptMaxP()
{
  Estimator(GetQueueSize());

  if (m_qAvg > m_minTh + 0.6 * (m_maxTh - m_minTh) && m_curMaxP <= m_targetMax)
    {
      m_curMaxP += m_alpha;
    }
  else if (m_qAvg < m_minTh + 0.4 * (m_maxTh - m_minTh) && m_curMaxP >=
m_targetMin)
    {
      m_curMaxP *= m_beta;
    }
  m_adaptEv = Simulator::Schedule(m_interval, &RedQueue::AdaptMaxP, this);
}
```

Anexa 4 – Modificarea modului de calcul al probabilităţii pentru RED

Adaugare funcţiei şi apelarea ei.

```
void RedQueue::anotherP() {
      double inflexionPoint = 0.9 * m_queueLimit;
      double intervalLength;
      double cuantizationRate;
      double placeInInterval;

      if ( m_packets.size() < inflexionPoint ) {
            intervalLength = inflexionPoint;
            cuantizationRate = intervalLength/10;
            placeInInterval = m_packets.size() / cuantizationRate - 10;
            m_vProb = 1/(1 + std::pow(2.71, -2 * placeInInterval));
      } else {
            cuantizationRate = (m_queueLimit - inflexionPoint)/10;
            placeInInterval = (m_packets.size() - inflexionPoint) /
cuantizationRate;
            m_vProb = 1/(1 + std::pow(2.71, -2 * placeInInterval));
      }
}
```

Anexa 5 – RRED

rred.h

```
#ifndef RRED_QUEUE_H
#define RRED_QUEUE_H

#include "red-queue.h"

#define MIN(a,b)              ((a) > (b) ? (b) : (a))
#define MAX(a,b)              ((a) > (b) ? (a) : (b))

#define N_HBINS 100
#define N_HLEVELS 100

namespace ns3 {

      struct RRedQueueBin {
            double last_drop_time;
            uint32_t score;
      };

      class RRedQueue : public RedQueue {
       public:
              static TypeId GetTypeId (void);
              RRedQueue ();
              virtual ~RRedQueue ();
       private:
            bool DoEnqueue(Ptr<Packet> p);
            void reportDrop(Ptr<Packet> p);
            uint32_t hashPkt(Ptr<Packet> pkt, uint32_t ilevel);
            uint32_t GetPacketSource(Ptr<Packet> p);
            uint32_t GetPacketDestination( Ptr<Packet> p);
            void resetBins(uint32_t v);
            void printBins();
            double updateBinsDroptime(Ptr<Packet> p);
            uint32_t dropAnomaly(Ptr<Packet> p);
            uint32_t hash_bins_;
            uint32_t hash_levels_;
            uint32_t score_max_;
            uint32_t score_min_;
            uint32_t score_pass_;
            double last_drop_time_;
            double drop_related_period_;
            struct RRedQueueBin bins_[N_HLEVELS][N_HBINS];
      };
};

#endif
```

rred.cc

```
#include "ns3/log.h"
#include "ns3/enum.h"
#include "ns3/uinteger.h"
#include "ns3/double.h"
#include "ns3/simulator.h"
#include "ns3/abort.h"
#include "ns3/random-variable-stream.h"
#include "rred-queue.h"
#include "ns3/internet-module.h"
#include "ns3/ipv4-header.h"

//#define RREDDEBUG
```

```
NS_LOG_COMPONENT_DEFINE ("RRedQueue");

namespace ns3 {
      NS_OBJECT_ENSURE_REGISTERED (RRedQueue);

      TypeId RRedQueue::GetTypeId (void)
      {
        static TypeId tid = TypeId ("ns3::RRedQueue")
          .SetParent<RedQueue> ()
          .AddConstructor<RRedQueue> ()
          .AddAttribute ("hash_bins_",
                         "hash_bins_",
                         UintegerValue (23),
                         MakeUintegerAccessor (&RRedQueue::hash_bins_),
                         MakeUintegerChecker<uint32_t> ())
          .AddAttribute ("hash_levels_",
                                 "hash_levels_",
                                 UintegerValue (2),
                                 MakeUintegerAccessor
(&RRedQueue::hash_levels_),
                                 MakeUintegerChecker<uint32_t> ())
            .AddAttribute ("score_max_",
                                 "score_max_",
                                 UintegerValue (10),
                                 MakeUintegerAccessor (&RRedQueue::score_max_),
                                 MakeUintegerChecker<uint32_t> ())
            .AddAttribute ("score_min_",
                                 "score_min_",
                                 UintegerValue (-1),
                                 MakeUintegerAccessor (&RRedQueue::score_min_),
                                 MakeUintegerChecker<uint32_t> ())
            .AddAttribute ("score_pass_",
                                 "score_pass_",
                                 UintegerValue (0),
                                 MakeUintegerAccessor (&RRedQueue::score_pass_),
                                 MakeUintegerChecker<uint32_t> ())
          .AddAttribute ("last_drop_time_",
                         "last_drop_time_",
                         DoubleValue (0),
                         MakeDoubleAccessor (&RRedQueue::last_drop_time_),
                         MakeDoubleChecker <double> ())
          .AddAttribute ("drop_related_period_",
                                 "drop_related_period_",
                                 DoubleValue (0.2),
                                 MakeDoubleAccessor
(&RRedQueue::drop_related_period_),
                                 MakeDoubleChecker <double> ())
      ;

        return tid;
      }

      RRedQueue::RRedQueue() :
            RedQueue()
      {
             NS_LOG_FUNCTION (this);
      }

      RRedQueue::~RRedQueue ()
      {
        NS_LOG_FUNCTION (this);
```

```
}

bool RRedQueue::DoEnqueue(Ptr<Packet> p) {

      NS_LOG_FUNCTION( this << p );

      bool enqueued = true;

      if (dropAnomaly(p)) {
            reportDrop(p);
            updateBinsDroptime(p);
            Drop(p);
            enqueued = false;
            last_drop_time_ = Simulator::Now ().GetSeconds() ;
            std::cout << "anomaly"<< std::endl;
      } else {
            enqueued = RedQueue::DoEnqueue(p);
            if ( !enqueued ) {
                  last_drop_time_ = Simulator::Now ().GetSeconds() ;
            }
      }

      return enqueued;

}


void RRedQueue::reportDrop(Ptr<Packet> p) {
      double drop_time=updateBinsDroptime(p);
      last_drop_time_ = drop_time;
}

uint32_t RRedQueue::hashPkt(Ptr<Packet> p, uint32_t ilevel) {
      uint32_t ibin=0;

      uint32_t param1=(GetPacketSource(p));
      std::cout << param1 << std::endl;
      uint32_t param2=(GetPacketDestination(p));

      ibin=((param1<<ilevel)+param2)%hash_bins_;

      return ibin;

}

void RRedQueue::resetBins(uint32_t v) {
      uint32_t i,j;
      for (i=0;i<hash_levels_;i++) {
            for (j=0;j<hash_bins_;j++) {
                  bins_[i][j].score=v;
                  bins_[i][j].last_drop_time=0;
            }
      }
}

void RRedQueue::printBins() {
      double now = Simulator::Now ().GetSeconds();

      uint32_t i,j;
      std::cout << "hash_levels_:" << hash_levels_ << "bins:" <<
hash_bins_ << "last_drop" << last_drop_time_ << now <<  hash_levels_ <<
```

```
hash_bins_ <<last_drop_time_<< std::endl;
            for (i=0;i<hash_levels_;i++) {
                  for (j=0;j<hash_bins_;j++) {
                        std::cout << bins_[i][j].score << " " << bins_[i]
[j].last_drop_time << std::endl;
                  }
                  std::cout << std::endl;
            }
      }

      double RRedQueue::updateBinsDroptime(Ptr<Packet> p) {
            double now = Simulator::Now ().GetSeconds();

            uint32_t ilevel=0;
            uint32_t ibin=0;
            for (ilevel=0;ilevel<hash_levels_;ilevel++){
                  ibin=hashPkt(p, ilevel);
                  bins_[ilevel][ibin].last_drop_time = now;
            }

            return now;
      }

      uint32_t RRedQueue::dropAnomaly(Ptr<Packet> p) {
            uint32_t rtn=1;
            double now = Simulator::Now ().GetSeconds();

            #ifdef RREDDEBUG
            std::cout<< now << "saddr" << GetPacketSource(p) << "daddr" <<
GetPacketDestination(p)<< std::endl;
            #endif

            uint32_t ilevel=0;
            uint32_t ibin=0;
            for (ilevel=0;ilevel<hash_levels_;ilevel++){
                  ibin = hashPkt(p, ilevel);

                  #ifdef RREDDEBUG
                  std::cout << "ilevel: " << ilevel << "ibin: " << ibin <<
std::endl;
                  #endif

                  // Need further analysis
                  std::cout << last_drop_time_ << ":" << now -
MAX(last_drop_time_, bins_[ilevel][ibin].last_drop_time) << " < " <<
drop_related_period_ << std::endl;
                  if (now - MAX(last_drop_time_, bins_[ilevel]
[ibin].last_drop_time) < drop_related_period_) {
                  //if (now-bins_[ilevel]
[ibin].last_drop_time<drop_related_period_) {
                  //if (now-last_drop_time_<drop_related_period_) {
                        if (bins_[ilevel][ibin].score>score_min_){
                              bins_[ilevel][ibin].score=bins_[ilevel]
[ibin].score-1;
                        }
                  } else {
                        if (bins_[ilevel][ibin].score<score_max_){
                              bins_[ilevel][ibin].score=bins_[ilevel]
[ibin].score+1;
                        }
                  }
```

```
                  std::cout << bins_[ilevel][ibin].score << " >= " <<
score_pass_ << std::endl;

                  if (bins_[ilevel][ibin].score>=score_pass_) {
                        rtn=0;
                  }

                  #ifdef RREDDEBUG
                  std::cout << "score: " << bins_[ilevel][ibin].score <<
"last_drop_time: " << bins_[ilevel][ibin].last_drop_time << std::endl;
                  #endif
            }

            #ifdef RREDDEBUG
            std::cout << std::endl;
            if (rtn==1) {
                  std::cout << "Anomaly Detected !!\n";
            }
            printBins();
            #endif

            return rtn;
      }

      uint32_t
      RRedQueue::GetPacketSource(Ptr<Packet> p)
      {
        Ipv4Address src_ip;
        Ptr<Packet> q = p->Copy();

        PacketMetadata::ItemIterator metadataIterator = q->BeginItem();
        PacketMetadata::Item item;
        while (metadataIterator.HasNext())
        {
          item = metadataIterator.Next();
          NS_LOG_FUNCTION("item name: " << item.tid.GetName());

          if(item.tid.GetName() == "ns3::Ipv4Header")
          {
            Callback<ObjectBase *> constr = item.tid.GetConstructor();
            NS_ASSERT(!constr.IsNull());

            ObjectBase *instance = constr();
            NS_ASSERT(instance != 0);

            Ipv4Header* ipv4Header = dynamic_cast<Ipv4Header*> (instance);
            NS_ASSERT(ipv4Header != 0);

            ipv4Header->Deserialize(item.current);
            src_ip = ipv4Header->GetSource();

            delete ipv4Header;
            break;
          }
        }
        return (&src_ip)->Get();
      }

      uint32_t
      RRedQueue::GetPacketDestination(Ptr<Packet> packet)
```

```
    {
      Ipv4Address dest_ip;
      Ptr<Packet> q = packet->Copy();

      PacketMetadata::ItemIterator metadataIterator = q->BeginItem();
      PacketMetadata::Item item;
      while (metadataIterator.HasNext())
      {
        item = metadataIterator.Next();
        NS_LOG_FUNCTION("item name: " << item.tid.GetName());

        if(item.tid.GetName() == "ns3::Ipv4Header")
        {
          Callback<ObjectBase *> constr = item.tid.GetConstructor();
          NS_ASSERT(!constr.IsNull());

          ObjectBase *instance = constr();
          NS_ASSERT(instance != 0);

          Ipv4Header* ipv4Header = dynamic_cast<Ipv4Header*> (instance);
          NS_ASSERT(ipv4Header != 0);

          ipv4Header->Deserialize(item.current);

          dest_ip = ipv4Header->GetDestination();

          delete ipv4Header;
          break;
        }
      }

      std::cout<< "destination:" << (&dest_ip)->Get() << std::endl;
      return (&dest_ip)->Get();
    }

}
```