Scalabilitatea în reţele de comunicaţii de date

Ion Stoica, Mihai Budiu — `istoica+@cs.cmu.edu`, `mihaib+@cs.cmu.edu`
`http://www.cs.cmu.edu/~istoica/`, `http://www.cs.cmu.edu/~mihaib/`

18 aprilie 1999

Subiect: Un algoritm scalabil pentru alocarea echitabilă a traficului în Internet

Cunoştinţe necesare: Cunoştinţe elementare despre funcţionarea reţelelor de calculatoare, răbdare

Cuvinte cheie: ruter, flux, rată de transmisie, disciplinele cozilor, FIFO, fair queuing, core-stateless fair queuing, scalabilitate

Cuprins

În acest articol ilustrăm una dintre direcţiile cercetării contemporane în domeniul Internetului: căutarea de algoritmi practici pentru a rezolva felurite probleme; problema pe care o ilustrăm este cea a alocării echitabile a traficului pe legăturile dintre calculatoare, în aşa fel încît toţi participanţii la trafic să primească aceeaşi cantitate. Deşi infrastructura curentă a Internetului nu permite acest lucru, implementarea acestui gen de algoritmi va fi necesară pentru a transforma Internetul într-o reţea care poate garanta calitatea serviciilor pe care le oferă.

1 Rutere şi fluxuri

1.1 Rutere

Internetul este o reţea compusă din agregarea unui mare număr de reţele diferite într-o construcţie unică. Dacă două organizaţii au reţelele lor proprii, folosind tehnologii eventual diferite, aceste două reţele pot fi conectate între ele cu ajutorul unui calculator special, care este cuplat la ambele reţele. Rolul acestui calculator este de a prelua dintr-o reţea datele care sunt destinate cuiva din cealaltă reţea şi de a le retransmite dincolo. Această funcţiune se numeşte *rutare*, iar calculatorul însuşi se numeşte *ruter*.

Un ruter are deci două funcţiuni:

(a) De a discuta cu celelalte rutere pentru a descoperi topologia reţelei.

(b) De a folosi informaţiile despre topologie pentru a trimite fiecare pachet primit spre destinaţia lui.

Într-un articol anterior din PC Report, Mihai a prezentat (a), rutarea în Internet[1]. Subiectul acelui articol era modul în care ruterele învaţă încotro trebuie trimis fiecare pachet primit, şi modul în care ruterele schimbă între ele informaţii despre structura reţelei.

În textul de faţă vom complementa informaţiile oferite cu acea ocazie, discutînd despre (b), modul în care ruterele transmit pachete.

1.2 Transmiterea pachetelor

Un ruter este legat la fiecare reţea pe care o deserveşte printr-o *interfaţă*. Un ruter va avea deci cel puţin două interfeţe. Cînd un pachet apare prin una din interfeţe, ruterul pune acel pachet într-o memorie internă. După aceea ruterul prelucrează pachetul şi decide în ce direcţie trebuie să-l trimită. Această informaţie este obţinută din adresa destinaţie, care se află în interiorul pachetului, şi din tabela de rutare, care conţine informaţii despre forma reţelei, şi pe care ruterul a construit-o conversînd cu vecinii săi. După ce decide încotro trebuie trimis pachetul, ruterul îl trimite pe interfaţa care este cea mai potrivită pentru destinaţie.

Figura 1 ilustrează arhitectura internă a unui ruter.

Aparent nimic nu este mai simplu: iei pachetul, cauţi destinaţia în tabelă, şi îl trimiţi mai departe. O astfel de operaţie se numeşte *stochează şi transmite* (*store and forward*), din motive evidente. Ce mai e de discutat aici?

Dacă veţi încerca însă să construiţi un ruter, veţi vedea că există o sumedenie de întrebări la care răspunsul nu este de loc evident. Iată unele din dificultăţile care se pot ivi:

Capacitate de prelucrare: O parte din probleme provin din faptul că viteza de prelucrare a unui ruter este finită: de exemplu se poate întîmpla ca mai multe intrări simultan să aducă pachete cu o viteză mai mare decît ruterul poate prelucra. Din cauza asta anumite pachete pot fi pur şi simplu pierdute, pentru că ruterul nu poate ţine pasul cu rata de transmisiune. (Astfel de pachete vor fi probabil retransmise de către sursă mai tîrziu.)

[1] Acest articol este disponibil din pagina sa de web.

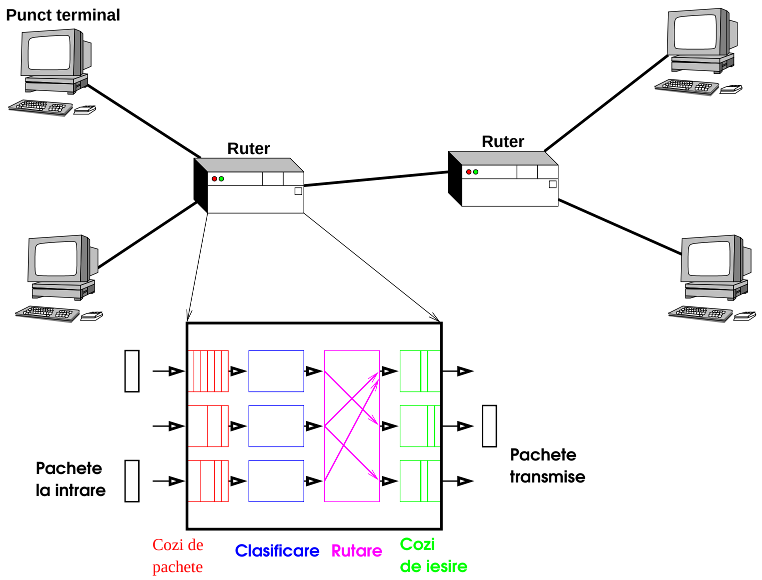

Figura 1: Operaţia de "înaintare" (forwarding) a unui ruter: un pachet este primit pe o interfaţă, este clasificat în funcţie de destinaţie, şi este înaintat pe interfaţa de ieşire. Pachetele adesea trebuie să aştepte în cozi prelucrarea pachetelor anterioare.

Capacitatea legăturilor: Dar, chiar presupunînd că ruterul poate clasifica pachete infinit de rapid, se pot ivi probleme: de exemplu imaginaţi-vă o reţea în care pachetele care vin pe două intrări sunt destinate pentru o singură ieşire; dacă cele trei reţele au aceeaşi capacitate, atunci în mod clar ruterul nu va putea trimite toate pachetele pe care le primeşte. Ce trebuie să facă cu pachetele pe care le primeşte şi nu le poate transmite?

Din cauza aceasta, ruterele conţin memorii, numite *buffere*, în care stochează pachetele pe care nu le pot prelucra. Pachetele se aşează în *cozi*, de unde sunt extrase unul cîte unul. Este clar însă că, oricît de multe buffere am avea, rafale mari de pachete pentru aceeaşi destinaţie pot umple cozile, cauzînd din nou pierderea pachetelor.

Calitate: Chiar presupunînd că avem o memorie infinită în ruter, timpul de aşteptare în cozi se poate dovedi inacceptabil pentru anumite aplicaţii. De exemplu, dacă avem o aplicaţie de telefonie prin Internet, dacă datele ajung după mai mult de 150ms de la un capăt la altul, se crează pauze neplăcute în conversaţie. Chiar mai rău este fenomenul care se petrece atunci cînd cozile variază în lungime: anumite pachete vor veni mai repede şi altele mai lent, distorsionînd vorbirea la receptor. Asta ne sugerează că anumitor pachete trebuie să li se dea prioritate. De asemenea, anumite pachete pot fi preferate pentru a fi "distruse" (de exemplu pachetele de voce care au întîrziat deja prea mult), pentru că nu mai sunt utile.

Una peste alta, un ruter trebuie să ia următoarele decizii:

- Care pachete şi cînd pot fi aruncate;
- În ce ordine deserveşte pachetele din cozi.

Modul în care ia aceste decizii are consecinţe foarte importante pentru comportarea reţelei. În acest articol vom vedea cîteva posibilităţi şi vom explora consecinţele lor, care sunt adesea foarte diferite.

Trebuie spus însă că majoritatea covîrşitoare a disciplinelor studiate *nu* sunt implementate în sisteme reale, fiind deocamdată doar un subiect de cercetare. Practic toate ruterele operaţionale astăzi în lume folosesc o disciplină foarte simplă: au o singură coadă pentru fiecare interfaţă, în care pun toate pachetele. Cînd coada este plină, noile pachete venite sunt pierdute. Această disciplină se numeşte FIFO: First In First Out, primul venit, primul plecat.

Din nefericire, deşi extrem de simplu de implementat, FIFO are nişte proprietăţi indezirabile, despre care vom discuta puţin mai jos, în secţiunea despre Fair Queuing, unde ilustrăm şi o alternativă. FIFO este inadecvată din multe puncte de vedere, aşa că multe eforturi în cercetare sunt acum îndreptate spre studiul altor discipline.

Însă, înainte de a vedea exemple de alte discipline, vom pune sub reflector o proprietate extrem de importantă a reţelei, care este influenţată de modul în care ruterele şi punctele terminale operează. Este vorba despre congestie.

2 Congestie şi controlul fluxului

Atunci cînd un segment al reţelei este înecat cu pachete pe care nu le poate transmite se produce un fenomen de *congestie*. Congestia este extrem de periculoasă, pentru că dacă nu este tratată corect poate periclita funcţionarea întregii reţele pentru o perioadă nedeterminată.

Scenariul cel mai sumbru arată astfel: o anumită legătură primeşte mai multe pachete decît capacitatea ei de transport. Ca atare, după ce cozile se umplu, inevitabil ruterul care deserveşte acea legătură începe să piardă pachete. După o vreme sursele care au trimis acele pachete încep să dea semne de îngrijorare, pentru că nu primesc confirmarea de primire; ca atare încep să retransmită pachetele pierdute. Acestea vor cauza creşterea mai mare a congestiei, pentru că folosesc acelaşi traseu. Cu cît reţeaua este mai blocată, cu atît mai multe pachete duplicate vor apărea (din cauza retransmisiilor), şi cu atît mai blocată va fi. Din această situaţie nu există ieşire, dacă nu se intervine imediat; reţeaua devine rapid nefuncţională.

Observaţi că acest lucru se poate petrece chiar dacă fiecare calculator terminal nu emite prea multe date; efectul se produce datorită *agregării* traficului de la mai multe conexiuni pe o legătură fizică comună. De exemplu, în figura 1, traficul între calculatoare din stînga şi oricare calculator din dreapta trebuie să treacă pe linia dintre cele două rutere.

În teoria reţelelor de transmisiuni de date se face distincţie între două clase de algoritmi:

- Algoritmi care controlează traficul între cele două puncte terminale ale comunicaţiei, asigurînd faptul că un calculator mai rapid nu îneacă o destinaţie lentă cu pachete pe care nu le poate primi. O legătură creată între două puncte pentru a transmite date se numeşte *flux* (*flow*). Algoritmii care păstrează fluxul între limite admisibile pentru ambii participanţi se numesc *algoritmi de control al fluxului* (flow-control).

- Algoritmi care controlează traficul pe legături din interiorul reţelei; aceştia sunt algoritmi de control al congestiei (congestion control).

Între aceste două clase există o suprapunere, pentru că o reducere a fluxului în mod implicit va reduce şi congestia. De fapt secţiunea următoare discută singura soluţie actualmente implementată pe scară largă în Internet, folosită de protocolul TCP, care administrează fluxuri între două calculatoare.

2.1 Soluţia TCP

Soluţia folosită la ora actuală de majoritatea implementărilor existente ale protocolului TCP a fost propusă de van Jacobson în 1988. Pentru a o înţelege ar trebui să studiem un tip special de protocol, numit protocolul cu *fereastră glisantă* (*sliding window*). Subiectul este deosebit de interesant, dar prea generos pentru acest articol, care are are un alt subiect. Sperăm să putem trata acest subiect într-un alt articol; pentru cititorul impacient trimitem la cartea lui Tanenbaum publicată de editura Agora, "Reţele de Calculatoare".

Din protocolul cu fereastră glisantă vom reţine numai însuşirile care ne interesează.

TCP este un protocol care garantează trimiterea cu succes a tuturor pachetelor. Pentru asta, toate pachetele care nu au fost confirmate la primire de către capătul celălalt sunt păstrate de emiţător, chiar şi după ce au fost deja transmise. Pentru că între trimiterea unui pachet şi recepţia confirmării se poate scurge mult timp, emiţătorul este gata să emită mai multe pachete la rînd, chiar dacă nu a primit confirmări pentru nici unul. Numărul de pachete care se pot afla în tranzit fără confirmare se numeşte *fereastră* (window).

Fereastra glisantă este folosită de TCP cu succes şi pentru controlul fluxului. Iată cum:

1. Să zicem că la un moment dat un flux TCP are o fereastră de 30 de pachete.

2. Să ne imaginăm că subit apare o congestie în reţea, pe o linie folosită şi de fluxul nostru. Din cauza asta un ruter va pierde pachete ale acestui flux.

3. După o vreme TCP observă ca nu a primit confirmările pentru pachetele trimise. Atunci, în loc să le retrimită pe toate, TCP reduce fereastra curentă la un pachet (primul trimis care nu a primit confirmare).

4. TCP retransmite apoi pachetul din fereastră.

5. După o vreme toate fluxurile care foloseau linia congestionată au pierdut pachete şi şi-au redus ferestrele. În reţea au început să intre din ce în ce mai puţine pachete noi, deci linia cu probleme s-a decongestionat.

6. Acum fluxurile TCP încep să primească confirmări, deci încep încet-încet să-şi mărească ferestrele la loc.

7. Acest proces se repetă permanent. Traficul pe acea legătură va oscila uşor în jurul valorii maxime posibile, fiind automat ajustat de către participanţii la trafic.

Soluţia aceasta este deosebit de elegantă. Principala ei calitate este că nu implică de loc reţeaua: numai nodurile terminale rulează acest algoritm, fără nici un fel de informaţii de la rutere. În plus, această soluţie funcţionează indiferent de politica după care ruterele din reţea aruncă pachete.

2.2 Problemele soluţiei TCP

Controlul congestiei prin variaţia ferestrei are mai multe dezavantaje, dar noi aici ne vom opri asupra unuia singur dintre ele. Să ne imaginăm că avem cinci fluxuri printr-o legătură congestionată. Să ne mai imaginăm că 4 dintre ele sunt implementări "normale" ale TCP, dar că unul dintre ele este o implementare foarte agresivă, care în loc să-şi reducă traficul, îl păstrează neschimbat.

Ce se va întîmpla este uşor de imaginat: cele patru fluxuri civilizate vor reduce traficul aşa cum trebuie, iar cel agresiv va folosi toată capacitatea posibilă a reţelei. În măsura în care acest flux de unul singur nu congestionează reţeaua, va funcţiona bine-mersi, furînd din porţia tuturor celorlalţi.

Dacă toate fluxurile ar fi lacome reţeaua s-ar congestiona iremediabil, şi nimeni n-ar primi nimic. Deci schema aceasta funcţionează numai dacă aproape toată lumea este rezonabilă.

Partea neplăcută este că această situaţie este deja întîlnită, într-o formă sau alta, în Internet:

- Anumiţi programatori lipsiţi de scrupule pot să facă această modificare pe maşinile lor, pentru a beneficia de mai multe resurse;

- O cantitate din ce în ce mai mare de aplicaţii de fapt nu folosesc protocolul TCP pentru transmisiune. TCP este un protocol care garantează zero pierderi, cu costul eventual al multor retransmiteri. Pentru aplicaţii ca audio, video, jocurile în reţea, etc. aceste trăsături sunt indezirabile: mai bine pierzi un pachet din cînd în cînd şi le aduci pe celelalte la timp, decît să întîrzii atît de mult.

Aplicaţiile de genul celor de mai sus folosesc protocolul UDP pentru transmisiune, care nu este fiabil, şi nu face nici un fel de control al fluxului sau congestiei.

De altfel puteţi face chiar dumneavoastră un experiment simplu dacă aveţi acces la două calculatoare legate în reţea, oarecare cunoştinţe de programare de reţea şi o oră liberă:

1. Deschideţi între aceste calculatoare nişte legături TCP pe care transmiteţi date fictive; măsuraţi cîte pachete primiţi în fiecare secundă avînd 1, 2, 3, etc. legături în paralel. În mod normal, datorită metodei de control a fluxului, aceste fluxuri vor oscila simultan între valorile maximă şi minimă, oferind aceeaşi rată medie de transmisiune pe o perioadă mai mare de timp (cîteva secunde).

2. Pentru ca acest experiment să funcţioneze trebuie să reuşiţi să congestionaţi reţeaua. Dacă limita de transmisiune va fi impusă nu de reţea, ci de hardware-ul sau software-ul pe care îl posedaţi, acest experiment probabil nu va funcţiona întocmai. Din momentul în care aţi congestionat reţeaua, suma ratelor medii de transport ale legăturilor TCP va trebui să fie constantă (adică fie că aveţi 5, fie 10, rata[2] totală de transport va fi egală cu capacitatea legăturii dintre cele două calculatoare).

3. Deschideţi apoi o legătură UDP între cele două calculatoare, păstrînd legăturile TCP deschise. Pompaţi date în toate cu viteza maximă. Măsuraţi din nou traficul. Rata fiecărui flux va arăta ca în figura 2.

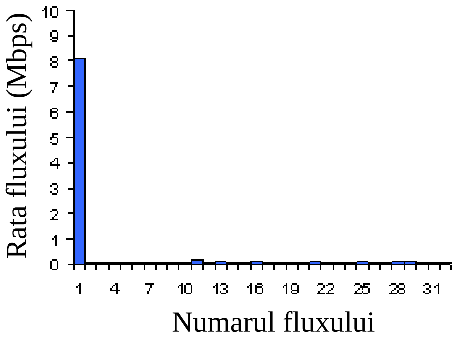

Figura 2: Acest grafic arată rata a 32 de fluxuri — un flux UDP (numărul 1) şi 31 de fluxuri TCP — pe o legătură Ethernet de 10 Mbps. Observaţi că fluxurile TCP nici nu se văd, iar UDP consumă aproape toată capacitatea canalului, cu circa 8 Mbps.

Veţi observa că legătura UDP obligă toate legăturile TCP să-şi reducă traficul, folosind de una singură toată capacitatea reţelei. Suma dintre UDP şi toate TCP-urile va fi constantă, TCP-urile vor fi aproximativ egale între ele, dar la o valoare mult mai scăzută decît legătura UDP.

3 Fair queuing: o disciplină echitabilă

De fapt, dacă ruterele tratează pachetele în ordinea sosirii (FIFO), şi le aruncă pe cele de la coadă, nici nu putem face mare lucru. O disciplină simplă ca FIFO nu poate asigura o distribuţie echitabilă a capacităţii în caz de congestie.

O soluţie, propusă în literatură, pentru aceasta problema este utilizarea de discipline mai "inteligente" în rutere. O astfel de disciplină este *Fair Queueing* ("cozi echitabile", FQ). În continuare vom descrie FQ şi vom arăta că, spre deosebire de FIFO, această disciplină asigură o distribuţie echitabilă a capacităţii, independent de algoritmii de adaptare folosiţi de surse.

FQ este definit în contextul unui sistem ideal, în care traficul este transmis la nivel de biţi, şi nu de pachete. FQ se aplică la fiecare coadă de ieşire separat: fiecare interfaţă a ruterului va calcula pentru propriul ei canal. Într-un astfel de sistem, FQ se reduce la tratarea fluxurilor în mod circular (*round-robin*: în ordine fluxul 1, 2, 3, ..., n, 1, 2, 3, etc.). Mai precis, în cadrul fiecărei runde, FQ transmite un bit de la fiecare flux care este activ.

Spre deosebire de FIFO, FQ asigură alocarea echitabilă a capacităţii. În particular, capacitatea alocată unui flux *nu* depinde de rata fluxului. De exemplu, dacă avem o conexiune de 1 Mbps şi două fluxuri, unul cu rata de 0.5 Mbps şi altul de 2 Mbps, atunci fiecare flux va primi 0.5 Mbps. Aceasta se întîmplă deoarece atîta timp cît cele două fluxuri sunt active, fiecare va avea exact un bit transmis în cadrul fiecărei runde.

Evident, într-un ruter real în care transmisia se face la nivel de pachete şi nu de biţi, FQ nu poate fi implementată exact. O primă idee pentru implementarea FQ la nivel de pachet ar fi să utilizăm din nou round-robin. Din păcate, un moment de reflexie ne spune că această soluţie nu este cea mai potrivită. Următorul exemplu ilustrează problema: dacă avem doua fluxuri, unul care transmite pachete de cîte 1000 octeţi, iar celălalt care transmite pachete de cîte 100 octeţi, este

[2] Rata unui flux este definită ca numărul de biţi transmişi în unitatea de timp.

uşor de văzut ca aplicînd round robin la nivel de pachet are drept rezultat alocarea unei capacităţi de 10 ori mai mare primului flux! De aceea, în practică, pentru implementarea FQ-ului se utilizează algoritmi mai sofisticaţi.

Un exemplu de astfel de algoritm este următorul: pentru fiecare pachet se calculează momentul de timp la care ultimul bit al acestuia ar urma să fie transmis într-un ruter ideal, în care traficul este transmis bit cu bit; apoi pachetele sunt transmise în ordinea crescătoare a acestor timpi.

Descrierea de mai sus sugerează principalul dezavantaj al disciplinei FQ: complexitatea. În esenţă ruterul trebuie sa ţină evidenţa pentru fiecare flux şi sa execute următoarele operaţii:

1. Să identifice cărui flux îi aparţine fiecare pachet;

2. Să hotărască ce pachet să elimine din memorie cînd aceasta este depăşită;

3. Să transmită pachetele în ordinea crescătoare a timpilor, aşa cum am discutat mai sus.

Deci, pe de o parte, avem FIFO, care este foarte uşor de implementat la viteze mari, dar nu asigură o alocare echitabilă a capacităţii, iar pe de alta parte avem FQ care asigură o distribuţie echitabilă a capacităţii, dar este mult mai complex de implementat. O întrebare firească este atunci dacă există o cale de mijloc, care sa îmbine simplitatea FIFO cu serviciul asigurat de FQ. Vom vedea o astfel de soluţie mai jos, dar înainte vom studia mai atent care este trăsătura indezirabilă a algoritmului FQ.

4 Scalabilitatea

Care este diferenţa dintre FIFO şi FQ care face FQ mai dificil de implementat? Dacă încercăm să scriem un program care să le implementeze, vom observa că FIFO manipulează structuri de date extrem de simple: de fapt o singură structură de date, care este o listă înlănţuită de pachete, stocate în ordinea sosirii. Listei i se adaugă pachete la coadă şi i se iau pachete de la cap. Aceste operaţii se pot implementa în cîteva linii de cod, deci se pot executa extrem de rapid.

Pe de altă parte, dacă studiem FQ, o să notăm faptul că ruterul trebuie să facă contabilitate pentru fiecare flux care îl traversează. Asta înseamnă o sumedenie de lucruri destul de neplăcute:

1. Ruterul trebuie să se uite la fiecare pachet şi să observe pachetele care deschid noi conexiuni, pentru a le rezerva structuri de date; fiecare conexiune va avea propria ei coadă;

2. Ruterul trebuie să clasifice fiecare pachet venit în funcţie de conexiunea pe care circulă, şi să îl pună în coada lui;

3. Ruterul trebuie să reevalueze la trimiterea fiecărui pachet starea cozii sale, pentru a vedea cînd următorul are dreptul să plece;

4. Ruterul trebuie să "vîneze" pachetele care termină conexiunea, pentru a distruge cozile respective.

Chiar dacă operaţiile acestea sunt relativ simple, structurile manipulate sunt foarte complicate; avem de-a face cu cozi care cresc independent, deci ne trebuie un alocator de memorie destul de sofisticat[3], care trebuie invocat la fiecare pachet venit sau plecat. În plus, numărul de structuri de date manipulate depinde de numărul de fluxuri din ruter.

Probleme extrem de greu de rezolvat apar dacă un calculator cu o conexiune deschisă cade: el nu va trimite niciodată semnalul de închiderea conexiunii, deci ruterul va păstra o coadă care teoretic nu mai dispare niciodată. Vom vedea într-un alt articol cum se rezolvă genul ăsta de neplăceri, folosind o tehnică numită *soft state*.

Cît de mare este numărul de fluxuri dintr-un ruter care operează în centrul reţelei? Informaţii de la MCI, care este unul din cei mai mari operatori de reţea din Statele Unite, arată că numărul de fluxuri active simultan într-un ruter de mare capacitate se poate apropia de un milion!

Fiecare pachet venit trebuie deci clasificat într-unul din aproape un milion de fluxuri! Asta înseamnă, chiar cu algoritmi inteligenţi, o prelucrare substanţială pentru fiecare pachet venit. Asta este virtual imposibil pentru un ruter care deserveşte mai multe fluxuri de sute de megabiţi pe secundă[4]: pur şi simplu nu are destul timp pentru a prelucra pachetele!

Aceasta este diferenţa principală în ceea ce priveşte posibilitatea de a realiza FIFO şi FQ: FQ trebuie să facă din ce în ce mai multe operaţii, dacă numărul de fluxuri care traversează ruterul creşte. Operaţia lui FIFO pe de altă parte este aceeaşi, independent de cîte fluxuri avem.

Internetul este unul dintre cele mai mari sisteme construite vreodată de om (şi în viitor va deveni probabil cel mai mare); ca atare dimensiunile cu care trebuie operat sunt uriaşe în multe circumstanţe (de pildă un milion de fluxuri). Din cauza asta algoritmii executaţi de rutere trebuie să fie eficienţi atît la un număr mic de conexiuni/pachete, cît şi la valori astronomice. Această proprietate a unui algoritm se numeşte *scalabilitate*: capacitatea de a funcţiona eficient la valori foarte variate ale încărcăturii de date prelucrate.

Dacă un cercetător propune pentru managementul Internetului un algoritm care nu este scalabil, poate să fie absolut sigur că acel algoritm nu va fi folosit niciodată în afara unor sisteme mici, experimentale. Din cauza asta FQ nu este un algoritm practic; ne trebuie o alternativă.

5 O soluţie scalabilă pentru fair queuing: CSFQ

Aşa cum am văzut, complexitatea algoritmului FQ se datorează în principal faptului că ruterul trebuie să menţină starea fiecărui flux. O schemă care să reducă în mod semnificativ complexitatea algoritmului FQ trebuie deci să reducă sau să elimine într-un mod sau altul această stare. Problema este că, dacă eliminăm pur şi simplu starea unui flux, nu mai este posibil să "protejăm" acel flux atunci cînd linia este congestionată. Suntem confruntaţi cu următoarea dilemă: pe de o parte dorim să eliminăm starea pentru a reduce complexitatea, dar pe de altă parte avem nevoie de această stare pentru a asigura protecţia fluxurilor.

Soluţia se bazează pe o idee foarte simplă: în loc să menţinem starea în rutere, o punem în pachete! Fiecare pachet oricum trebuie prelucrat de fiecare ruter, deci complexitatea procesării creşte doar cu un factor constant (iar nu cu un factor depinzînd de numărul de fluxuri). Algoritmul general arată atunci în felul următor:

1. Sursa, sau eventual primul nod în reţea, evaluează starea fluxului, şi o pune în pachetele acestuia.

[3] Vedeţi şi articolul lui Mihai publicat cu ceva timp în urmă în PC Report despre alocarea memoriei.

[4] Cel puţin pentru rutere comerciale obişnuite; inginerii care construiesc supercalculatoare au mai multe tehnologii în mînecă pentru a rezolva astfel de probleme.

2. Nodurile următoare tratează pachetele pe baza informaţiei conţinute în pachete.

În acest mod un nod interior nu are nevoie să menţină starea pentru fiecare flux, ceea ce face posibilă implementarea eficientă a algoritmului. Pe de altă parte, sursele sau nodurile de la marginea reţelei, care pun starea în pachete, trebuie să menţină stare pentru fiecare flux, pentru a eticheta corect pachetele. Din fericire, aceste noduri suportă un trafic mult mai mic decît nodurile interioare, şi în consecinţă nu ridică probleme de scalabilitate. Aceasta soluţie "împinge" starea din nodurile interioare la marginea reţelei şi apoi foloseşte pachetele pentru a transmite această stare.

În continuare descriem pe scurt un algoritm, numit Core-Stateless Fair Queueing (CSFQ), care implementează FQ în cadrul arhitecturii prezentate mai sus. "Core-stateless" înseamnă că ruterele din "miezul" reţelei nu menţin nici un fel de informaţii despre starea conexiunilor. Pentru a defini algoritmul trebuie să precizăm:

- Informaţia de stare transportată de pachete, care este înscrisă de ruterele de la margine;
- Operaţiile executate de un nod interior la primirea unui pachet.

În CSFQ, starea înscrisă în pachete reprezintă *rata fluxului*; sursa sau primul nod din reţea calculează dinamic rata fiecărui flux şi o pune în pachetele acestuia. Nodurile interioare calculează pe baza traficului primit şi a celui transmis *rata echitabilă* care trebuie sa fie alocată fiecărui flux. Intuitiv, această este rata maximă care ar fi alocată unui flux de algoritmul FQ la ruterul respectiv (în exemplul dat mai sus, cu două conexiuni pe o linie de 1 Mbps, rata echitabilă era de 0.5 Mbps). Din lipsă de spaţiu nu prezentăm amănuntele calculării acestei rate. Cititorul interesat poate găsi detaliile la `http://www.cs.cmu.edu/~istoica/csfq`. În continuare vom prezenta numai operaţia executată de un ruter interior la primirea unui pachet.

Ne vom concentra asupra unei singure ieşiri din ruter; pentru fiecare ieşire vom menţine o altă valoare a ratei echitabile.

Să ne uităm la un ruter pentru care rata echitabilă este r_b. Această rată depinde numai de numărul de fluxuri şi de capacitatea canalului de ieşire. Să zicem că acest ruter primeşte un pachet în care este înscrisă rata r. Atunci ruterul transmite pachetul cu probabilitatea $P = \max(1, r_b/r)$ şi îl "aruncă" cu probabilitatea $1 - P$. Se poate arăta ca aceasta operaţie foarte simplă duce la o buna aproximare a algoritmului FQ. Intuitiv, aceasta poate fi văzut în următorul exemplu didactic:

Să presupunem că avem un flux cu pachete de lungime egală şi cu o rată constantă $r > r_b$, unde valoarea r este înscrisă în fiecare pachet de ruterele de la margine. În acest caz $P = r_b/r$. Drept urmare, din n pachete vor fi transmise în medie numai $n \times r_b/r$ pachete. Deoarece fiecare pachet are aceeaşi lungime, rata fluxului se va reduce în medie cu r_b/r, ceea ce înseamnă ca rata finală va fi $r \times r_b/r = r_b$. În acest mod, fluxului îi este alocată probabilistic o rată de cel mult r_b, rată care este egală cu rata echitabilă (adică cea alocată fluxului dacă FQ ar fi fost folosit)! În final, după ce aruncă din pachete, ruterul schimbă rata înscrisă în fiecare pachet cu r_b. Informaţia din pachet rămîne astfel consistentă cu noua rată a fluxului după ce străbate ruterul congestionat.

Astfel, CSFQ este capabil să asigure o bună aproximare a algoritmului FQ la nodurile interioare, fără a menţine starea fluxurilor. În fapt, complexitatea acestui algoritm la nodurile interioare este apropiată de cea a algoritmului FIFO. În concluzie, CSFQ reuşeşte să îmbine simplitatea FIFO cu funcţionalitatea algoritmului FQ.

Figura 3 arată pentru comparaţi valorile fluxului obţinute cu un simulator, în exact condiţiile din experimentul anterior cu TCP.

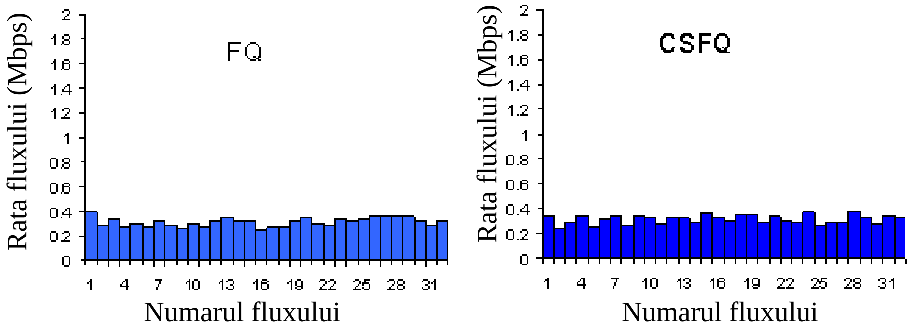

Figura 3: Valorile fluxurilor pentru o conexiune UDP (numărul 1) şi 31 de conexiuni TCP peste o reţea de 10 Mbps, atunci cînd disciplina de coadă a ruterelor este FQ, respectiv CSFQ. Observaţi că fiecare legătură primeşte cam aceeaşi valoare din flux, undeva în jur de 0,30 Mbps. Între CSFQ şi FQ nu există diferenţe notabile, arătînd că CSFQ realizează promisiunea de a aproxima FQ pentru această situaţie. Valorile sunt obţinute cu un simulator de reţea, care este accesibil din pagina de web a lui Ion.

Există o grămadă de alte detalii despre funcţionarea CSFQ, inclusiv simulări detaliate şi estimări analitice ale calităţii sale; acestea sunt toate descrise pe larg în articolul "Core Stateless Fair Queueing: Achieving Approximately Fair Bandwidth Allocations in High Speed Networks", de Ion Stoica, Scott Shenker şi Hui Zhang, prezentat în 1998 la conferinţa SIGCOMM, a cărui copie este disponibilă din pagina de web a lui Ion.

6 Concluzii

În acest articol ilustrăm modul în care se poate proiecta o soluţie pentru rezolvarea unei probleme specifice în Internet. Articolul însă aparent bate mult timp cîmpii, pentru că trebuie să construim o serie de concepte. Iată aici o vedere de ansamblu a schemei articolului;

1. Întîi discutăm despre modul în care ruterele procesează pachetele pe care le înaintează în Internet. Vedem că ruterele pot decide ordinea în care deservesc cozile de pachete şi ordinea în care aruncă pachetele pe care nu le pot deservi.

2. Apoi discutăm despre situaţiile în care ruterele *trebuie* să piardă pachete, pentru că reţeaua primeşte mai mult decît poate transmite pe anumite segmente. Aceasta se numeşte congestie.

3. Petrecem puţin timp pentru a vedea cum algoritmii contemporani (TCP) rezolvă problema congestiei fără a avea nici un fel de ajutor din partea reţelei. Detecţia congestiei se bazează pe observaţia că anumite pachete nu primesc confirmări.

4. Vedem apoi că o astfel de soluţie se bazează implicit pe buna-credinţă a tuturor participanţilor, şi că nu este deloc protejată împotriva unor indivizi malefici, care în cazul congestiei în loc să reducă rata de transmisiune o măresc, beneficiind de reducerea făcută de toţi ceilalţi.

5. Discutăm apoi o soluţie pentru această problemă, care are nevoie de cooperarea ruterelor din reţea: fair queuing. Ruterele procesează pachetele nu în ordinea primirii, ci pe rînd, cîte unul de la fiecare sursă. Asta înseamnă că sursele care emit prea mult sunt penalizate mai mult.

6. Vedem apoi că fair queuing, deşi foarte dezirabil, suferă de problema *scalabilităţii*.

7. Discutăm apoi problema scalabilităţii pe scurt: modul în care complexitatea unui algoritm creşte în funcţie de datele de intrare. Observăm că în Internet avem de-a face adesea cu probleme extrem de mari (milioane de date de intrare), deci algoritmii ineficienţi sunt nepractici.

8. În fine, vedem o soluţie de compromis, care este scalabilă şi *aproximează* fair queuing suficient de bine.

Cred că e cinstit să stăvilim aici fluxul de informaţii pe care le transportă acest articol. Vom reveni.