

Universitatea Politehnică București
Facultatea de Automatică și Calculatoare

LiSA - Sistem de comutare a pachetelor

Autor:

Nicu Ioan Petru

Coordonatori științifici:

Prof. dr. ing. Nicolae Țăpuș

Drd. ing. Octavian Purdilă

Iunie 2005

Cuprins

1	Introducere.....	4
2	Noțiuni teoretice.....	6
2.1	Rețele locale în tehnologie Ethernet.....	6
2.1.1	Originile tehnologiei Ethernet.....	6
2.1.2	Modul de funcționare al unui sistem Ethernet.....	7
2.1.3	Cadrul Ethernet și Adresele Ethernet.....	9
2.1.4	Îmbunătățirea performanțelor rețelelor Ethernet.....	10
2.1.5	Segmentarea rețelelor Ethernet folosind Switch-uri.....	11
2.1.6	Tabela de comutare.....	11
2.1.7	Microsegmentare.....	14
2.1.8	Comunicație Half-Duplex și Full-Duplex.....	15
2.1.9	Domeniu de broadcast. VLAN-uri.....	16
2.2	Particularități ale subsistemului de rețea din Linux Kernel.....	18
2.2.1	Definiții și concepte.....	18
2.2.2	Recepționarea de pachete în Linux.....	19
2.2.3	Softnet.....	20
2.2.4	Reordonarea pachetelor pe arhitecturi SMP.....	21
2.2.5	Soluția completă și sigură: IRQ Affinity.....	22
2.2.6	Efectul de colaps datorat congestiei și soluția NAPI.....	23
2.2.7	Performanțele algoritmului NAPI.....	25
2.2.8	Implementarea Socket Bufferelor în Linux.....	28
3	Arhitectura aplicației.....	32
3.1	Arhitectura Linux Multilayer Switch (LMS).....	32
3.2	Arhitectura aplicației de configurare (CLI).....	33
3.3	Arhitectura unui sistem ce rulează LiSA.....	34
4	Detalii de implementare.....	35
4.1	Implementarea modulului Linux Multilayer Switch (LMS).....	35
4.1.1	Interfațarea cu subsistemul rețea din Linux Kernel.....	36
4.1.2	Structurile de date folosite.....	39
4.1.3	Tabela de comutare (FDB).....	41
4.1.4	Operația de căutare și operațiile de modificare a listelor.....	42
4.1.5	Durata de viață a intrărilor din tabela de comutare.....	45
4.1.6	Algoritmul de comutare la nivel legătură de date.....	46
4.1.7	Algoritmul de difuzare. Optimizări.....	50
4.1.8	Interfațarea spațiu utilizator - kernel.....	52
4.2	Implementarea aplicației de configurare (CLI).....	54
4.2.1	Organizarea comenzilor.....	54
4.2.2	Analizorul de Comenzi.....	55
5	Studiul performanțelor și testare.....	56
5.1	Descrierea Platformei de testare.....	56
5.2	Indicatori de performanță.....	56
5.3	Rata de comutare a pachetelor.....	57

5.4 Rata de transfer.....	61
6 Concluzii.....	62
7 Anexă: Primitive și algoritmi de sincronizare în Linux Kernel.....	65
7.1 Zăvorâre și Operații atomice.....	65
7.1.1 Operații pe biți.....	66
7.1.2 Spinlock.....	66
7.1.3 RW Lock.....	67
7.1.4 Semafoare.....	67
7.2 Algoritmul Read-Copy-Update (RCU).....	68
7.2.1 Manipularea listelor înlănțuite folosind RCU.....	68

1 Introducere

Fără îndoială Internetul este una dintre cele mai mari invenții ale timpurilor noastre și a avut un efect profund asupra fiecărui aspect din viața noastră. Dezvoltarea acestuia a schimbat felul în care facem afaceri, felul în care comunicăm, obținem informații și ne educăm.

Apariția Internetului a revoluționat tehnica de calcul și comunicațiile într-un mod fără precedent. O dată cu extinderea sa și în paralel cu evoluția tehnicii de calcul către ceea ce reprezintă ea în zilele noastre, a avut loc și o dezvoltare a aplicațiilor software care au trebuit să țină pasul cu nevoile de comunicare ale utilizatorilor. Toate acestea au determinat creșterea exponențială a utilizării resurselor existente astfel încât, datorită problemelor de congestie și supraîncărcare, implementarea unei rețele locale¹ chiar și de dimensiuni medii a devenit o sarcină deloc banală pentru proiectanți.

Cea mai frecventă arhitectură pentru LAN este Ethernet². Tehnologia Ethernet este utilizată pentru transportul datelor între nodurile unei rețele, cum ar fi stații de lucru, imprimante sau servere.

Performanțele unui LAN Ethernet 802.3 pot fi influențate negativ de o serie de factori dintre care putem menționa: aplicațiile software ce utilizează excesiv lărgimea de bandă, numărul mare de utilizatori ce împart același segment partajat de rețea, întârzierile în transmisie datorate coliziunilor, numărul mare de broadcast-uri ș.a.

Un pas critic în prevenirea acestor efecte nedorite se poate realiza printr-o proiectare a rețelei care să urmărească câteva cerințe primordiale: *Fiabilitate (Reliability)*, *Scalabilitate (Scalability)*, *Adaptabilitate (Adaptability)*, *Disponibilitate (Disponibility)*.

Unul dintre principiile cunoscute în proiectarea LAN-urilor îl constituie utilizarea echipamentelor capabile să realizeze segmentarea rețelei. Printre acestea segmentarea LAN-urilor utilizând switch-uri este una dintre cele mai simple și mai uzuale opțiuni.

LiSA, sau după denumirea sa completă, **Linux Switching Appliance**, este un proiect în întregime open-source³, având la bază sistemul de operare Linux⁴, care își propune să pună la dispoziția utilizatorilor săi toate instrumentele necesare implementării unei soluții de switching performante la un cost cât mai scăzut. Această soluție a fost proiectată pentru a putea fi utilizată în special în rețelele de dimensiuni mici și medii.

Proiectul are în componența sa mai multe sub-proiecte. Componenta de bază a proiectului constă într-un modul pentru kernel-ul sistemului de operare Linux, precum și într-o serie de modificări în sursele acestuia menite să realizeze integrarea funcționalității adăugate în stiva de networking existentă. Acest sub-proiect l-am denumit *Linux Multilayer Switch* sau mai pe scurt **LMS**.

O a doua componentă **LiSA** este o aplicație în mod text care rulează din spațiul utilizator și care a fost proiectată în scopul gestionării parametrilor de funcționare ai modului de kernel. Datorită răspândirii și recunoașterii pe plan mondial a produselor

1 Rețea Locală, referită în literatura de specialitate sub prescurtarea LAN (în limba engleză, Local Area Network)

2 Se referă la familia de produse pentru LAN acoperite de standardul IEEE 802.3

3 <http://www.opensource.org/docs/definition.php>

4 <http://www.kernel.org>

*Cisco Systems*⁵, am ales ca această interfață să fie asemănătoare cu cea a binecunoscutului sistem de operare pentru echipamente de rețea *Cisco IOS*. Avantajul este, evident, faptul că orice administrator de rețea care a mai lucrat cu astfel de echipamente poate utiliza acest software fără a depune nici un efort de familiarizare. Această componentă am denumit-o *Command Line Interface*, sau pe scurt **CLI**.

În fine, a treia componentă a proiectului o constituie realizarea unei mini-distribuții de Linux destinată rulării pe un sistem dedicat. Această distribuție a fost realizată astfel încât să nu ocupe mai mult de câțiva megabytes de spațiu fizic pe disc și să conțină toate pachetele necesare pornirii și rulării unui sistem de operare Linux minimal, având în plus instalate și cele câteva aplicații componente ale modulului **CLI**.

Comutarea pachetelor este realizată în întregime în software. Deși la prima vedere aceasta constituie o penalitate pentru performanță, fiind evident că o implementare în hardware ar putea avea o eficiență sporită, avantajele pe care le oferă o astfel de abordare sunt suficient de satisfăcătoare, iar argumentele la această afirmație vor fi prezentate în continuare.

În primul rând merită menționat faptul că **LiSA** poate rula pe orice platformă hardware suportată de kernel-ul Linux. De asemenea codul de comutare de pachete este complet independent de hardware-ul plăcilor de rețea din sistem. Practic acesta poate funcționa cu orice chipset de placă de rețea pentru care există un device driver în kernel.

Totodată un avantaj major îl constituie posibilitatea de a refolosi hardware învechit. Astfel, un sistem PC ce nu mai face față cerințelor acute de resurse caracteristice aplicațiilor desktop actuale poate fi refolosit eficient pentru comutare sau chiar rutare de pachete într-o rețea de dimensiuni mici sau medii, obținându-se performanțe apreciabile la un cost aproape neglijabil.

Un alt avantaj major îl reprezintă kernel-ul linux în sine, luând în considerare facilitățile avansate de rutare, filtrare de pachete și traffic shaping deja existente și dovedite a fi stabile.

Inițial s-a ținut către modificarea configurației hardware a unui sistem PC x86 prin eliminarea tuturor componentelor inutile unui switch (floppy, usb, audio, video) și chiar eliminarea întregii stive de networking din linux pentru a fi înlocuită cu una optimizată pentru un anumit chipset. O altă facilitate ar fi fost realizarea unui BIOS minimal capabil să efectueze pe lângă inițializarea sistemului și câteva funcții suplimentare cum ar fi realizarea transferului unei imagini de sistem de operare de pe o interfață serială pe hard disk-ul sistemului sau restaurarea configurației în cazul pierderii datelor existente. Concluzia a fost că această abordare are mai multe inconveniente. Dintre acestea se pot menționa cu precădere dependența de un anumit hardware, dificultățile de procurare ale acestuia, precum și documentația aproape inexistentă necesară realizării unui software de tip BIOS. Ceea ce a fost realizat în această abordare a fost un programator de chip-uri de BIOS, împreună cu software-ul de comandă asociat, dar acestea nu constituie obiectul acestei lucrări.

În consecință, dorința de a realiza o aplicație dedicată s-a materializat în mini-distribuția de Linux. Astfel, ca aplicație, s-a instalat și testat această distribuție pe un sistem PC Embedded din seria LE-564 produs de firma Commell Systems⁶. Sistemul dispune de 3 porturi Fast Ethernet și un port Gigabit, precum și de un adaptor Compact Flash to IDE. În plus, au fost adăugate în configurația sistemului o memorie SDRAM de 128 MB

5 <http://www.cisco.com/>

6 <http://www.commell-sys.com/Product/SBC/LE-564.htm>

precum și o memorie Compact Flash de 128 MB pentru emularea unui hard disk IDE.

2 Noțiuni teoretice

2.1 Rețele locale în tehnologie Ethernet

2.1.1 Originile tehnologiei Ethernet

La sfârșitul anului 1972, Dr. Robert M. Metcalfe împreună cu colegii săi de la Xerox PARC au dezvoltat primul sistem experimental Ethernet menit să interconecteze mai multe calculatoare Xerox Alto⁷. Rețeaua ethernet experimentală era folosită să conecteze calculatoarele Alto unele cu celelalte și cu câteva servere și imprimante. Semnalul de ceas pentru interfața Ethernet experimentală era derivată din ceasul de sistem Alto, ceea ce a rezultat în rate de transfer de aproximativ 2,94 Mbps.

Prima rețea experimentală a lui Metcalfe a fost denumită *Alto Aloha Network*. În 1973 Metcalfe i-a schimbat numele în "Ethernet", pentru a sublinia că sistemul poate suporta orice calculator, nu numai Alto, dar și pentru a evidenția că noile sale tehnologii au evoluat mult peste sistemul Aloha. El a ales să deriveze numele din cuvântul "ether"⁸ pentru a descrie un atribut esențial al sistemului: mediul fizic de transmisie (cablul) transportă biții la toate stațiile, așa după cum vechiul "eter luminos" era considerat capabil să propage undele electromagnetice prin spațiu.

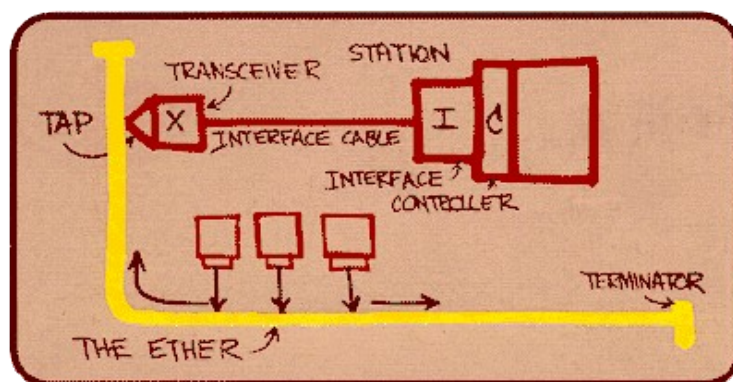


Figura 1 Schema primului sistem Ethernet

Specificațiile oficiale pentru Ethernet au fost publicate în 1980 de către un consorțiu alcătuit din mai mulți producători ce au alcătuit standardul DEC-Intel-Xerox (DIX). Acest efort a transformat Ethernet-ul experimental într-un sistem de producție ce opera la viteze de 10Mbps. Astfel, tehnologia Ethernet a intrat în vederea comitetului IEEE⁹ de standarde pentru LAN.

Standardul IEEE a fost publicat pentru prima dată în 1985, cu titlul oficial "IEEE

7 Unul dintre primele modele de stație de lucru personală, cu interfață utilizator grafică.

8 În limba română, eter.

9 Institute of Electrical and Electronics Engineers (IEEE 802).

802.3 Carrier Sense Multiple Access with Collision Detection (CSMA/CD) Access Method and Physical Layer Specifications". Standardul a fost adoptat și de Organizația Internațională pentru Standardizare (ISO), ceea ce l-a transformat într-un standard mondial de networking.

Toate echipamentele Ethernet, din 1985 până acum, au fost fabricate în conformitate cu standardul IEEE 802.3. Pentru acuratețe, toate acestea ar trebui referite ca tehnologie "IEEE 802.3 CSMA/CD". Totuși, majoritatea lumii le cunoaște după numele original de Ethernet, așa cum vor fi referite și în această lucrare.

2.1.2 Modul de funcționare al unui sistem Ethernet

Un sistem Ethernet este constituit din trei elemente de bază:

- canalul fizic utilizat pentru transportul semnalelor Ethernet între calculatoare.
- un set de reguli de control al accesului înglobat în fiecare interfață Ethernet, ce permite mai multor calculatoare să acceseze în mod echitabil un canal Ethernet partajat.
- un cadru Ethernet, constituit dintr-un set standard de biți, folosit la transportul datelor prin sistem.

Orice calculator dotat cu tehnologie Ethernet, pe care îl putem numi stație, operează independent de toate celelalte stații conectate la aceeași rețea (nu există nici un controller central). Toate stațiile atașate unui Ethernet sunt conectate la un sistem de semnalizare partajat, numit și mediu. Semnalele Ethernet sunt transmise serial, bit cu bit, prin canalul de semnalizare partajat către fiecare stație atașată. Pentru a transmite date, o stație ascultă mai întâi canalul și atunci când acesta este liber, stația trimite datele sub forma unui cadru sau pachet¹⁰ Ethernet.

După fiecare transmitere a unui cadru, toate stațiile din rețea trebuie să concureze în mod egal pentru posibilitatea transmiterii următorului cadru. Aceasta asigură echitatea accesului la canalul de rețea și exclude posibilitatea ca o singură stație să acapareze canalul, blocând accesul celorlalte stații. Accesul la canalul partajat este determinat de mecanismele de control al accesului la mediu¹¹ integrate în interfețele Ethernet din fiecare stație. Mecanismul de control al accesului la mediu este bazat pe un sistem denumit *CSMA/CD* (*Carrier Sense Multiple Access with Collision Detection*).

Înainte de efectuarea unei transmisii fiecare interfață trebuie să aștepte până când nu mai există nici un semnal pe canalul partajat. Dacă o altă interfață transmite, va exista un semnal pe canal, numit *purtătoare*¹². Toate celelalte interfețe trebuie să aștepte până când semnalul purtător încetează înainte să încerce să transmită, iar acest proces se numește *detecție de purtătoare*¹³. Prin termenul de *acces multiplu*¹⁴ se înțelege că toate interfețele Ethernet sunt egale ca prioritate în abilitatea de a trimite cadre pe rețea. Deoarece timpul de

10 Termenul precis din specificația Ethernet este "frame" sau cadru, dar termenul pachet este de asemenea des folosit

11 Medium Access Control, sau MAC

12 Carrier

13 Carrier sense

14 Multiple access

propagare al unui semnal prin rețea este finit, primii biți ai unei transmisii de cadru nu ajung la toate nodurile rețelei simultan. Astfel, există posibilitatea ca două interfețe să considere că rețeaua este liberă și să înceapă transmiterea de cadre simultan. Când se întâmplă aceasta, sistemul Ethernet are posibilitatea de detecție a "coliziunii" semnalelor, oprind transmisia și încercând retransmisia cadrelor. Această proprietate se numește *detecție a coliziunilor*¹⁵.

Protocolul CSMA/CD a fost proiectat pentru a oferi acces echitabil la canalul partajat, astfel încât toate stațiile să aibă posibilitatea de folosire a rețelei. După fiecare transmisie de cadru toate stațiile folosesc protocolul CSMA/CD pentru a determina ce stație va urma să folosească canalul Ethernet.

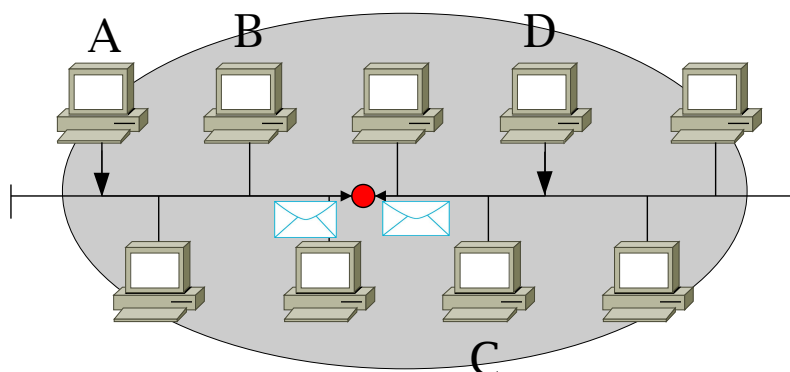


Figura 2 Aparitia Coliziunilor într-o rețea Ethernet

Dacă se întâmplă ca mai mult de o stație să transmită în același timp pe un canal Ethernet, atunci se spune că semnalele colizionează. Stațiile sunt notificate de acest eveniment și își reprogreamază transmisia folosind un algoritm special de așteptare¹⁶. O parte a acestui algoritm implică alegerea unor intervale de timp aleatoare pentru planificarea retransmisiei cadrelor, ceea ce previne blocarea sistemului.

Principiul de funcționare al acestui sistem asigură că majoritatea coliziunilor pe un canal Ethernet care nu este supraîncărcat vor fi rezolvate în timpi de ordinul microsecundelor. O coliziune nu rezultă în pierderi de date. Interfața Ethernet așteaptă un număr aleator de microsecunde după care retransmite automat datele.

Într-o rețea cu trafic intens se poate întâmpla să apară multe coliziuni pentru o încercare de transmitere a unui cadru. În eventualitatea apariției coliziunilor repetate pentru aceeași încercare de transmisie, stațiile implicate își extind seturile de potențiali timpi de așteptare din care se aleg timpii aleatori pentru retransmisie. Coliziunile repetate pentru o anumită transmisie indică o rețea aglomerată. Numai după o serie de 16 coliziuni succesive o interfață va renunța la trimiterea cadrului. Aceasta se poate întâmpla numai în cazul în care canalul Ethernet este supraîncărcat pentru o perioadă destul de îndelungată, sau, bineînțeles, în cazul unei avarii.

¹⁵ Collision detection

¹⁶ Referit și ca backoff algorithm, în literatura de specialitate.

2.1.3 Cadrul Ethernet și Adresele Ethernet

La baza sistemului Ethernet se găsește cadrul Ethernet, care este folosit pentru livrarea datelor între nodurile rețelei. Cadrul constă într-un set de biți organizați în mai multe câmpuri. Acestea includ câmpurile de adresă, un câmp de date variabil ca dimensiune ce poate conține de la 46 până la 1500 de octeți de date și un câmp FCS¹⁷ folosit pentru a verifica integritatea cadrului la primire.

Primele două câmpuri din cadru conțin adrese pe 48 de biți, mai exact adresa destinație și adresa sursă. IEEE controlează distribuția acestor adrese administrând o porțiune din câmpul de adresă. IEEE face acest lucru oferind identificatori pe 24 de biți, numiți OUI, sau "Organizationally Unique Identifiers", fiecărui fabricant de interfețe Ethernet. Producătorii creează adrese de 48 de biți, din care primii 24 sunt OUI-ul asociat. Aceste adrese pe 48 de biți mai sunt cunoscute sub numele de adrese fizice, adrese hardware sau adrese de MAC.

Atunci când un cadru Ethernet este trimis pe canalul partajat, toate interfețele analizează primii 48 de biți ai acestuia, ce conțin adresa destinație. Interfețele compară apoi adresa destinație a cadrului cu propria lor adresă. Interfața cu aceeași adresă ca adresa destinație a cadrului va citi în întregime cadrul și îl va livra nivelului rețea de pe același sistem, în timp ce toate celelalte interfețe vor abandona citirea cadrului atunci când descoperă că adresa destinație a acestuia diferă de cea proprie.

Există o clasă specială de adrese MAC, numite adrese de multicast. O adresă de multicast permite ca un cadru Ethernet să poată fi recepționat de un grup de stații. Software-ul de rețea poate configura interfața Ethernet a unei stații să "asculte" după anumite adrese de multicast. Aceasta face posibil ca mai multe stații să fie dispuse într-un grup căruia i s-a atribuit o adresă specifică de multicast. Un singur cadru trimis la o adresă asociată acelu grup de multicast va fi primit de toate stațiile ce au subscris la acesta.

Un caz particular de adresă de multicast este cunoscută sub numele de adresă de broadcast. Aceasta este o adresă MAC în care toți biții au valoarea 1. Toate interfețele Ethernet ce detectează un cadru cu această adresă destinație îl vor citi și îl vor livra nivelului rețea din sistem.

2.1.4 Îmbunătățirea performanțelor rețelelor Ethernet

Volumul de trafic din rețelele locale crește proporțional cu numărul de calculatoare conectate la acestea. Dacă aceasta ar fi singura sursă de creștere a traficului, atunci îmbunătățirea ar putea fi limitată la introducerea de backbone-uri care să interconecteze mai multe rețele locale. Pentru segmentarea rețelelor locale de dimensiuni mari în mai multe rețele de dimensiuni mai mici ar putea fi folosite bridge-uri sau routere, fără a se face cheltuieli proporționale cu numărul de stații și menținând astfel raportul număr de stații pe segment de rețea la un nivel ce ar asigura performanțe acceptabile.

Cu toate acestea, progresul tehnologic a dus la apariția de calculatoare desktop și stații de lucru din ce în ce mai inteligente și mai rapide. Acestea, precum și aplicațiile ce folosesc intens rețeaua (aplicații client-server, file sharing, transmisie video sau voce prin

¹⁷ Frame Check Sequence

Internet) au determinat o creștere a nevoilor utilizatorilor pentru o lărgime de bandă mai mare decât cea disponibilă pe un canal Ethernet 802.3 partajat. În rețelele actuale se înregistrează o creștere în transmisia fișierelor de dimensiuni mari: imagini, video și aplicații multimedia, precum și o creștere a numărului de utilizatori.

O dată cu creșterea numărului de utilizatori care folosesc rețeaua pentru a transfera fișiere de dimensiuni mari, pentru a accesa servere de fișiere și pentru a se conecta la Internet, apare un fenomen numit *congestie*. Efectul acestuia se materializează în timpi de răspuns mari, durate mai mari pentru transferul fișierelor și o scădere a productivității utilizatorilor datorită întârzierilor rețelei.

Pentru a înlătura efectele congestiei este necesară o lărgime de bandă mai mare sau folosirea eficientă a celei existente.

Deoarece creșterea lărgimii de bandă implică înlocuirea echipamentelor și cablării existente, aceasta constituie o soluție costisitoare și nu va fi luată în considerare. În schimb soluția de luat în considerare este segmentarea rețelei deja existente folosind echipamente precum bridge-uri, routere sau switch-uri.

2.1.5 Segmentarea rețelelor Ethernet folosind Switch-uri

Segmentarea rețelelor locale Ethernet folosind switch-uri are avantajul de a elimina impactul coliziunilor, congestia și problemele legate de lărgimea de bandă disponibilă per stație de lucru păstrând infrastructura de echipamente și cabluri deja existentă.

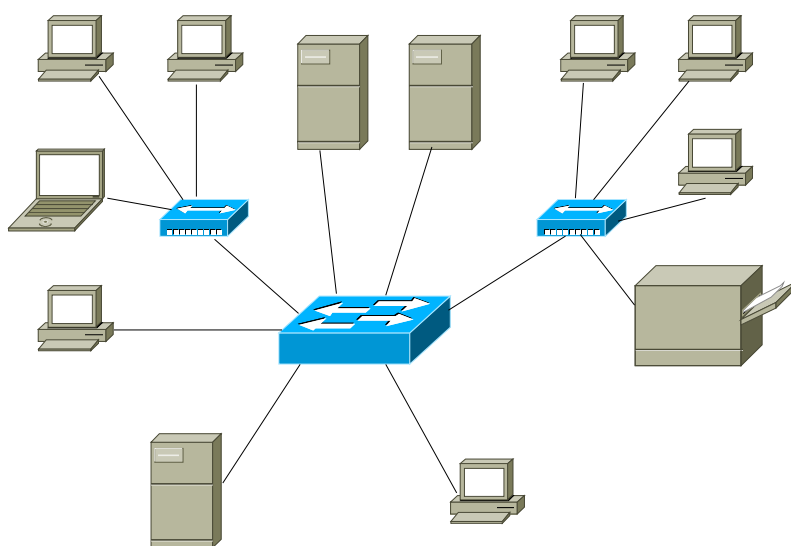


Figura 3 LAN Switch

Un switch este un dispozitiv ce funcționează la nivelul legătură de date din modelul OSI¹⁸ și permite transferul simultan de cadre între un număr mare de LAN-uri și stații de lucru. Din punct de vedere conceptual, acesta poate fi asemănat cu un bridge paralel sau bridge multi-port, având proprietatea de a permite transferul de date între oricare două porturi simultan.

¹⁸ Open System Interconnection (OSI) reference model

2.1.6 Tabela de comutare

Decizia de comutare este luată numai pe baza adreselor MAC destinație din antetele cadrelor primite.

Un switch inspectează fiecare cadru primit pentru a citi câmpul de adresă MAC destinație. Apoi, pe baza unei tabele interne de asociere a adreselor MAC la porturi, decide portul corespunzător de ieșire. Această tabelă internă mai este cunoscută sub numele de *tabelă de comutare*¹⁹.

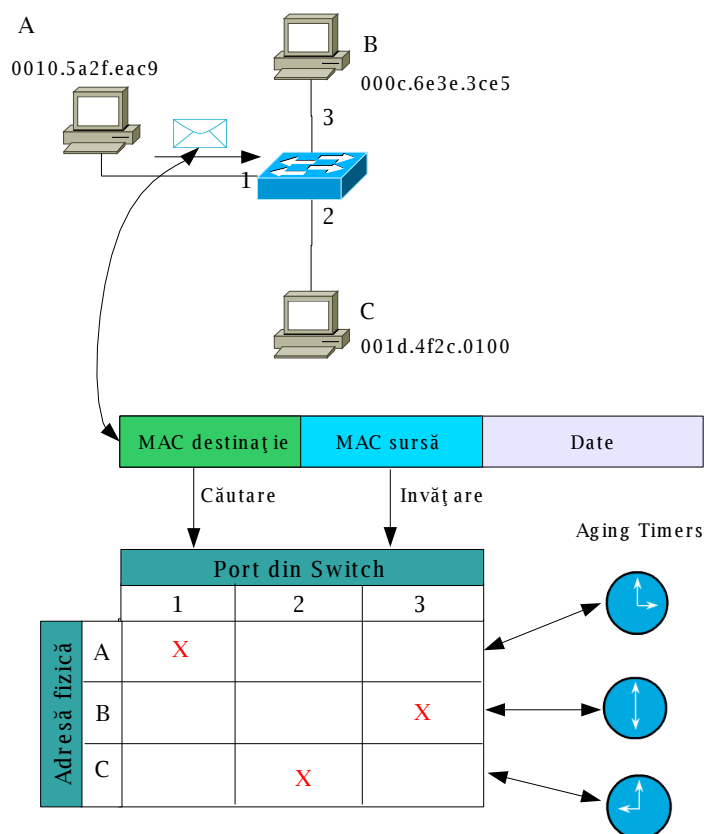


Figura 4 Tabelă de comutare

Așa cum se poate observa în Figura 4, atunci când o stație de lucru trimite un cadru către altă stație din rețea, switch-ul citește adresa MAC destinație din cadru și caută portul corespunzător de ieșire în tabela de comutare. Dacă portul de ieșire coincide cu portul de intrare, switch-ul va ignora acel cadru (stația destinație se află pe același segment de rețea cu stația sursă). În cazul în care căutarea nu întoarce nici un rezultat, switch-ul va trimite cadrul primit către toate porturile sale cu excepția portului de intrare, procedeu cunoscut sub numele de *flooding*.

Tabela de comutare poate avea intrări *statice* sau *dinamice*. Intrările statice din tabelă pot apărea doar în cazul *switch-urilor configurabile*²⁰, acestea fiind introduse de administratorul de rețea. Comutarea bazată numai pe intrări statice în tabela de comutare

¹⁹ Switching table, forwarding database

²⁰ Numite în industrie switch-uri cu management.

nu este o metodă recomandabilă în practică, deoarece stațiile de lucru își pot modifica locația fizică, iar administratorul de rețea trebuie să aibă grijă să facă modificările corespunzătoare de fiecare dată când se întâmplă aceasta. Intrările statice în tabela de comutare sunt utile numai în cazul stațiilor care nu își modifică niciodată locația.

Din fericire, majoritatea switch-urilor își construiesc tabela de comutare dinamic. Atunci când un cadru ajunge la switch, acesta analizează și adresa sursă din antetul acestuia, iar dacă aceasta nu se găsește în tabela de comutare, switch-ul va crea o nouă înregistrare în tabela sa care va indica faptul că adresa respectivă de MAC poate fi găsită pe segmentul de rețea asociat cu portul de intrare al cadrului. Astfel, spunem că un switch "învăță" adrese de MAC.

Pentru a evita problemele de comutare provocate de mutarea unei stații pe un segment de rețea din alt port al switch-ului, intrările dinamice din tabela de comutare au o durată finită de viață. Astfel, la învățarea unei adrese de MAC, intrării create în tabela de comutare i se asociază un ceas setat la un interval de timp de o valoare prestabilită (sau configurabilă în cazul switch-urilor configurabile). Atunci când expiră²¹ acest interval de timp, intrarea asociată din tabela de comutare este ștearsă. Dacă după momentul activării ceasului, dar înainte de expirarea acestuia, apare un cadru nou ce confirmă faptul că aceeași adresă de MAC poate fi găsită pe același port, atunci ceasul asociat intrării respective din tabela de comutare este resetat la valoarea inițială și apoi este reactivat.

Atunci când o adresă destinație nu se găsește în tabela de comutare se aplică metoda de flooding a cadrului pe toate celelalte porturi din switch. Aceasta va avea ca efect învățarea locației adresei respective de MAC la un moment ulterior, în cazul în care se va primi un răspuns la acel cadru.

Există două metode de comutare a cadrelor printr-un switch. Fiecare dintre acestea are avantajele și dezavantajele sale:

- *Store-and-Forward*: cadrul este recepționat complet înainte ca orice comutare să aibă loc. Apoi sunt citite adresele sursă și destinație și sunt aplicate regulile corespunzătoare de comutare. Această metodă are dezavantajul introducerii latenței. Latența este introdusă de așteptarea primirii întregului cadru și poate avea valori ridicate în cazul cadrelor de dimensiuni mai mari. Avantajul îl constituie posibilitatea de a efectua detecția erorilor.
- *Cut-through*: se citește adresa MAC destinație și se începe imediat transmiterea cadrului, înainte de recepționarea sa completă. Această metodă reduce latența la comutare, dar are dezavantajul ne-detectării erorilor. Există posibilitatea comutării unor cadre cu erori ceea ce va duce la retransmisia acestora.

2.1.7 Microsegmentare

Un switch poate împărți un LAN în microsegmente formate dintr-o singură stație de lucru. Aceasta asigură transformarea unui domeniu mare de coliziuni în mai multe domenii fără coliziuni.

²¹ Proprietate cunoscută sub numele de "mac aging"

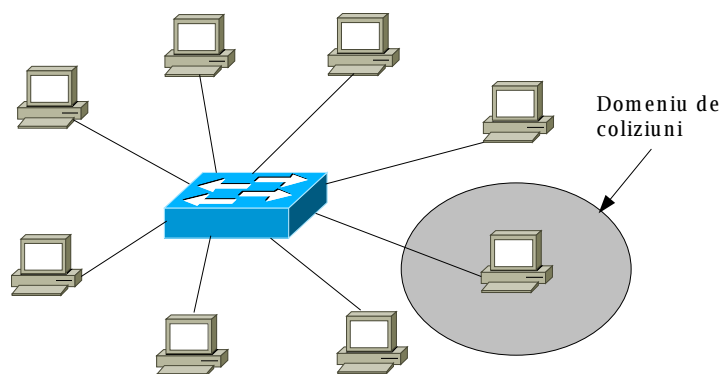


Figura 5 Microsegmentare LAN cu Switch

O astfel de configurație garantează o utilizare foarte eficientă a lărgimii de bandă. De exemplu, în cazul în care rețeaua este implementată în tehnologie FastEthernet, se pot obține rate de transfer de 100Mbps între oricare două noduri conectate în switch.

Datorită modului foarte eficient de utilizare a lărgimii de bandă, se obține în acest caz un throughput mai mare decât în cazul rețelelor conectate prin bridge-uri sau hub-uri. Într-o astfel de configurație, lărgimea de bandă disponibilă tinde spre 100%.

Cu toate acestea, dacă mai multe stații încearcă să acceseze simultan aceeași stație este posibilă apariția unei probleme de congestie.

Există două soluții la această problemă, ambele fiind implementate în switch-urile de producție:

- conectarea nodurilor unde converge majoritatea traficului în porturi de viteză superioară. Această soluție mai este cunoscută și sub numele de *comutare asimetrică*. Uzual serverele și alte resurse partajate sunt conectate în porturile de capacitate mai mare din switch, în timp ce stațiile de lucru, sau clienții sunt conectați în porturi de capacitate mai mică.
- Posibilitatea de a conecta resursele partajate în mai multe porturi din switch. Această soluție are ca efect creșterea lărgimii de bandă fără a fi nevoie de schimbarea tehnologiilor folosite.

2.1.8 Comunicație Half-Duplex și Full-Duplex

Ethernet a fost conceput ca un sistem de comunicație half-duplex. În timp ce datele pot fi transferate în ambele direcții, la un anumit moment o stație poate doar să trimită sau să primească. Pe mediile fizice folosite original cu Ethernet (cablu coaxial), aceasta era singura modalitate de comunicație posibilă, deoarece era utilizat același fir atât pentru recepție cât și pentru transmisie.

O dată cu apariția tehnologiei Ethernet twisted pair, există perechi de fire separate pentru transmisie și recepție. Cu toate acestea, deoarece mai multe stații partajează mediul de transmisie, este nevoie de un mijloc de prevenire a transmisiilor simultane. Pentru aceasta 10Base-T folosește detectarea informațiilor primite în timpul unei transmisii pentru a indica o coliziune stațiilor ce transmit, invocând algoritmul de așteptare și retransmisie necesar pentru funcționarea corectă Ethernet.

Într-un mediu microsegmentat există certitudinea că întotdeauna va fi un singur nod care va folosi o pereche de fire, deoarece există numai un singur nod terminal conectat la fiecare port din switch. Numai acest nod va comunica cu switch-ul (folosind perechea Tx a cablului) și numai switch-ul va comunica înapoi cu acesta (folosind perechea Rx a cablului). Nu există nici un fel de concurență la folosirea cablului precum în mediile standard Ethernet.

Având posibilitatea de eliminare a concurenței, nu mai este nevoie de detecția coliziunilor și de algoritmi de așteptare și retransmisie. Acestea pot fi eliminate și atât stația cât și switch-ul pot transmite după voie în ambele sensuri simultan, așa cum este ilustrat și în Figura 6:

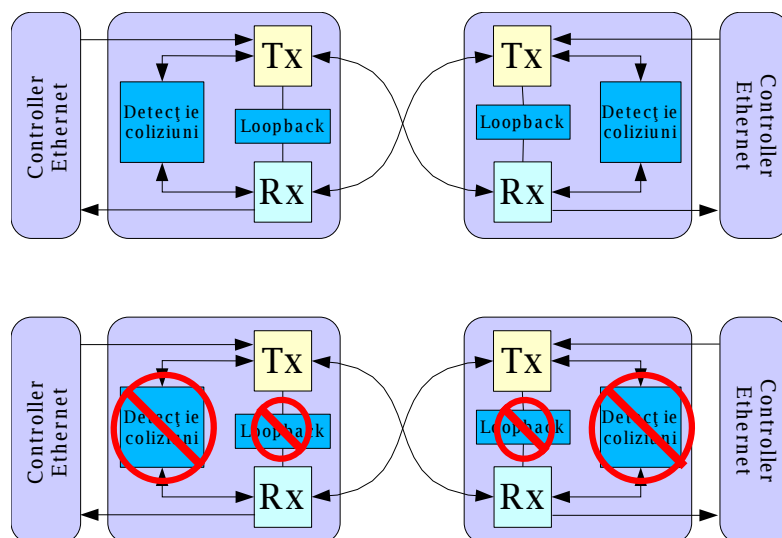


Figura 6 Ethernet Half-Duplex și Full-Duplex

Rata de transfer de date a fost astfel dublată, deoarece se pot transmite date la întreaga capacitate a canalului în ambele direcții. Aceasta este valabilă indiferent de rata de transfer half-duplex (de exemplu: semnalarea full-duplex poate fi folosită atât pe 10Base-T cât și pe 100Base-Tx). Singurele cerințe sunt:

- să fie folosite perechi de fire separate pentru transmisie și recepție (Ex: 10Base-T, 100Base-Tx, 100Base-FX)
- în fiecare port din switch să fie conectat un singur nod. Aceasta elimină competiția pentru folosirea canalului în orice direcție.
- Hub-ul central să fie un switch. Hub-urile tradiționale nu permit funcționarea full-duplex.

Trebuie specificat că numai modul de funcționare al aplicațiilor va determina dacă o stație va profita de această creștere a capacității.

Aplicațiile actuale nu folosesc banda simetric. De exemplu transferurile de fișiere, ce ar trebui să profite în mod normal de creșterea benzii, sunt asimetrice prin natura lor: în mare parte datele sunt transferate într-un singur sens. O stație care face transferuri intensive de fișiere nu va profita de dublarea benzii oferite de comutarea full-duplex.

În mod normal serverele utilizează canalul în ambele sensuri simultan. În timp ce orice stație folosește rețeaua asimetric, serverul va profita de modul de operare full-duplex pentru a gestiona transferuri dinspre o stație și simultan către alta.

2.1.9 Domeniu de broadcast. VLAN-uri.

După cum se preciza în secțiunea 2.1.5, un switch are proprietatea de a separa domeniile de coliziuni, iar în cazul microsegmentării LAN-ului de a le elimina complet.

Cu toate acestea toate stațiile conectate în switch vor împărtăși același domeniu de broadcast. Astfel, atunci când o stație de pe un segment de rețea conectat în switch trimite un cadru cu adresa MAC destinație ff:ff:ff:ff:ff:ff (broadcast) switch-ul îl va comuta pe toate segmentele de rețea adiacente.

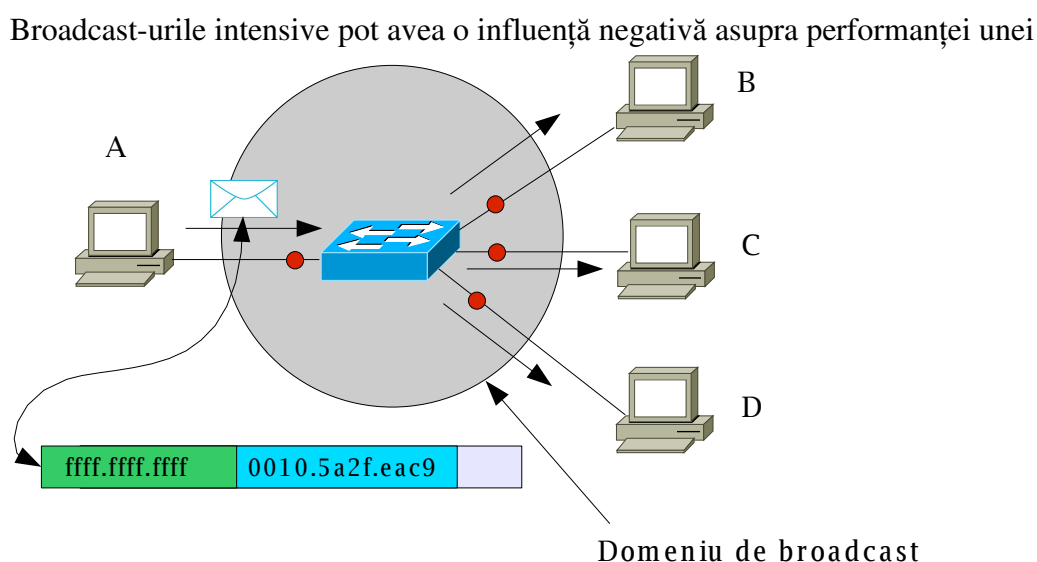


Figura 7 Broadcast

rețele. Cu toate acestea ele sunt necesare în funcționarea anumitor protocoale de rețea. De exemplu, protocolul *ARP* funcționează pe bază de broadcast: se trimite mai întâi un cadru de tip *ARP Request* cu adresă destinație broadcast în care se cere adresa de MAC asociată unui anumit IP. Cererea este recepționată de toate stațiile din rețea, iar stația care își recunoaște adresa IP din cerere răspunde cu un *ARP Reply* în care completează adresa sa MAC.

În anumite cazuri întâlnite în practică se dorește separarea acestor domenii de broadcast. Aceasta nu numai din motive de performanță a rețelei, dar și din considerente de securitate. Un exemplu practic ar fi dorința de a împărtăși o rețea locală în grupuri, independent de locația stațiilor²², fiecare grup având acces la resursele proprii (servere de fișiere, baze de date etc.), dar nu și la resursele altui grup. Aceasta ar asigura conectivitate numai între stațiile din același grup, ele alcătuind o rețea virtuală. De aici a apărut și ideea de LAN-uri virtuale, sau VLAN-uri.

Desigur, atunci când un switch implementează VLAN-uri, se realizează împărțirea

²² De exemplu, pentru o firmă s-ar putea dori separarea în grupuri după departament.

domeniilor de broadcast. Astfel un cadru de broadcast nu mai este trimis către toate stațiile de pe rețea, ci numai către stațiile ce aparțin aceluiași VLAN cu stația ce a trimis acel cadru.

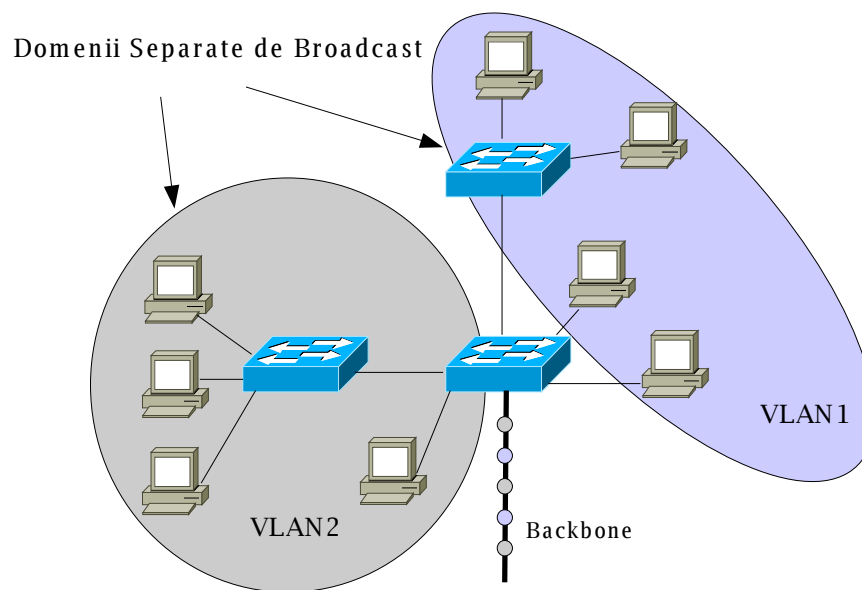


Figura 8 Switch cu VLAN-uri

Din perspectiva switch-ului apar două tipuri de porturi:

- porturi asociate unor VLAN-uri: pe acestea circulă cadre numai din VLAN-urile respective.
- porturi trunchi: pe acestea pot circula cadre din mai multe VLAN-uri (uzual portul din switch poate fi configurat să accepte cadre emise dintr-o anumită mulțime de VLAN-uri).

În cazul porturilor configurate în mod trunchi, este permisă trecerea cadrelor din mai multe VLAN-uri. Problema evidentă ce apare este identificarea VLAN-ului de care aparține un astfel de cadru. Soluția la această problemă a fost propusă de protocolul IEEE 802.1q și constă în extinderea antetului unui cadru cu 4 octeți care vor conține informații legate de VLAN. Această metodă este cunoscută sub numele de marcaj²³ VLAN 802.1q.

2.2 Particularități ale subsistemului de rețea din Linux Kernel

2.2.1 Definiții și concepte

Subsistemul de rețea din sistemul de operare Linux tinde să aibă un design orientat pe obiecte, așa cum o are o mare parte din kernel. Obiectele de bază cu care se lucrează sunt:

²³ Tagging în limba engleză.

- *Device* sau *Interfață*: O interfață de rețea este o entitate ce trimite și primește pachete. Aceasta reprezintă uzual codul de interfațare a unui dispozitiv fizic, ca de exemplu o placă de rețea. Totuși, unele dispozitive sunt implementate integral prin software, ca de exemplu interfața de loopback folosită într-un sistem pentru a trimite date către el însuși.
- *Protocol*: Fiecare protocol este practic un limbaj distinct de networking. Unele protocoale există pur și simplu pentru că anumiți producători au ales să folosească modalități proprii de comunicare, iar altele au fost proiectate pentru scopuri speciale. În interiorul kernel-ului Linux fiecare protocol este reprezentat printr-un modul separat de cod ce oferă servicii nivelului socket.
- *Socket*: Denumirea provine din conceptele de "fișă și priză"²⁴. Un socket reprezintă o conexiune la nivel de rețea și este accesibil programelor utilizator sub forma unui file descriptor. În kernel fiecare socket este reprezentat printr-o pereche de structuri ce constituie interfața socket de nivel înalt și interfața protocolului de nivel scăzut.
- *Socket Buffer* (*sk_buff*)²⁵: Toate bufferele utilizate în nivelul de rețea sunt *socket buffere*. Acestea oferă facilitățile generale de buffering și flow control necesare protocoalelor de rețea.

2.2.2 Recepționarea de pachete în Linux

Se va explica mai întâi ce se întâmplă la nivelul cel mai de jos, mai exact cum sunt recepționate pachetele la nivelul plăcilor de rețea.

Kernel-ul Linux suportă un număr destul de mare de tipuri de interfețe. Pe lângă interfețele uzuale Ethernet 10/100BaseT sau Gigabit, există suport și pentru adaptoare isdn, fddi, wireless lan ș.a. Fiecare dintre acestea are modul său specific de primire a unui cadru. În acest capitol se va considera ce se întâmplă doar în cazul adaptoarelor Ethernet.

Memoria on-board a plăcii este de obicei împărțită în două regiuni folosite pentru recepție și transmisie de cadre. De exemplu, plăcile de rețea 3c509²⁶ au o zonă de tampon pentru pachete de dimensiune 4kB împărțită egal în 2kB pentru primire (Rx) și 2kB pentru trimitere (Tx). Un model mai nou, 3c509B dispune de 8kB on-board ce pot fi împărțiți în 4kB Rx / 4kB Tx, 5kB Rx / 3 kB Tx sau 6kB Rx / 2kB Tx.

Cadrele recepționate sunt stocate în acea regiune de memorie într-o structură FIFO circulară²⁷ denumită *inel-rx*. La recepționarea unui cadru, placa de rețea va genera o întrerupere²⁸ pentru a informa procesorul de apariția acestui eveniment. În consecință se va rula un handler de întrerupere înregistrat de către metoda *open* a device-ului. Atunci se alocă spațiu pentru o nouă structură specifică kernel-ului linux, numită *socket buffer* (*sk_buff*), iar cadrul este copiat în aceasta.

Există mai multe metode de transfer a cadrului din memoria tampon a plăcii de rețea:

²⁴ Plug and socket în limba engleză

²⁵ orice pachet ce trece prin Linux kernel este abstractizat printr-o structură *sk_buff* (declarată în *include/linux/skbuff.h*). Această structură stă la baza codului de networking din kernel-ul Linux.

²⁶ Model de placă de rețea produsă de firma 3Com Corporation

²⁷ sunt cunoscute și sub denumirea de ring-buffers

²⁸ IRQ sau Interrupt Request

- I/O programat (NE200, 3c509)
- shared memory (WD90x03, 3c503)
- Direct Memory Access (DMA) și bus mastering (Lance, DEC21040)

Dacă se folosește DMA, atunci socket buffer-ul poate fi prealocat și mapat la regiunea DMA. Această optimizare va reduce utilizarea procesorului, dar cu toate acestea nu va popula cache-ul acestuia.

O interfață de rețea în linux este abstractizată printr-o structură de date numită *net_device*²⁹. Această structură cuprinde o multitudine de date printre care numele interfeței, memoria I/O, irq, informații despre coada de tx și câțiva pointeri la funcții ce execută diverse operații asupra device-ului: inițializare, configurare, colectare de statistici, transmitere de cadre etc.

Timul petrecut în rutina de întrerupere este critic și trebuie limitat la o valoare cât mai mică. Pe toată perioada execuției rutinei de tratare a întreruperii întreg mecanismul de întreruperi este dezactivat. Dacă rutina ar executa operații complexe, consumatoare de timp, atunci există posibilitatea să se acumuleze prea multe pachete în inelul rx (care în anumite cazuri poate avea o capacitate de stocare doar pentru 2 pachete), iar aceasta va rezulta în pierdere de pachete. Printre alte efecte nedorite generate în această situație putem menționa că procesele din spațiul utilizator nu vor mai avea acces la procesor și în consecință sistemul va părea blocat în acest interval.

2.2.3 Softnet

În linux există mai multe abordări pe cazul de primire de cadre. Abordarea existentă în kernelele mai vechi (inclusiv versiunea 2.4) se numește *softnet*. În această abordare, din rutina de tratare a întreruperii se transferă cadrul în socket buffer, apoi acesta este înlănțuit într-o coadă de primire și se face întoarcerea din întrerupere. Această coadă este referită ca *softnet* și este unică pentru toate interfețele pe un sistem cu un singur procesor. Mai exact, *softnet_data[NR_CPUS]* este un tablou de NR_CPUS structuri *softnet_data*, mai exact câte una pentru fiecare procesor din sistem, ceea ce evită problemele legate de serializare și excludere mutuală. Pachetele intră și părăsesc această coadă într-o manieră FIFO.

O dată ce pachetul este înlănțuit, se poate face fără probleme întoarcerea din întrerupere, după ce în prealabil se anunță kernel-ul că va trebui să extragă acest pachet din coadă la un moment ulterior. Mecanismul folosit în acest scop este numit "întrerupere software" sau bottom half.

Cu toate că întoarcerea din întrerupere se dorește a fi cât mai rapidă posibil, trebuie avut în vedere și controlat cazul de congestie. Rezolvarea acestei probleme în versiunile 2.4.x este simplă: înlănțuirea bufferelor se face până când lungimea cozii atinge *netdev_max_backlog* (fixată la valoarea 300) moment în care nu se mai înlănțuiesc buffere până când aceasta revine din nou la valoarea zero. De asemenea se refuză toate pachetele ce sosesc în acest timp.

²⁹ structură declarată în include/linux/netdevice.h

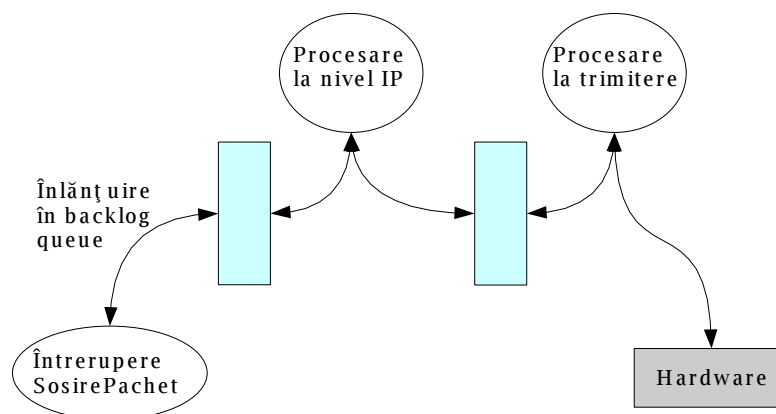


Figura 9 Soluția Softnet

2.2.4 Reordonarea pachetelor pe arhitecturi SMP

Datorită implementării mecanismului de primire din softnet, unde există câte o coadă pentru fiecare procesor din sistem, softnet a trebuit să implementeze o strategie de reordonare a pachetelor.

Pentru a explica problema să considerăm două pachete destinate unui socket client ce sosesc unul după altul într-un sistem dual-procesor. Planificatorul de întreruperi încredințează procesorului CPU0 prelucrarea primului pachet, ce este plasat în coada de backlog a acestuia. Al doilea pachet, ce este destinat aceluiași socket client sosește ulterior în sistem, iar planificatorul încredințează prelucrarea procesorului CPU1. Nu există nici o garanție de ordine a prelucrării efectuate de cele două procesoare. Această ordine depinde de mai mulți factori, printre care gradul de încărcare per procesor. Dacă CPU1 termină primul prelucrarea, aceasta va determina ca al doilea pachet să ajungă primul la nivelul TCP și socket înaintea primului (care a intrat în sistem via CPU0).

Linux-ul a rezolvat această problemă implementând RFC 2883³⁰. Din punctul de vedere a implementării TCP din linux, atâta timp cât partenerul de comunicație implementează SACK, problema este ameliorată substanțial, însă nu complet rezolvată.

2.2.5 Soluția completă și sigură: IRQ Affinity

Problema se poate înlătura complet doar atașând static fiecare interfață la un anumit procesor, prin *IRQ Affinity*³¹.

Începând cu versiunile de kernel 2.4.x, în linux există posibilitatea de asociere a anumitor IRQ-uri la anumite procesoare (sau grupuri de procesoare). Această capabilitate este cunoscută sub numele de *IRQ Affinity*.

Ca o paranteză, vom arăta cum se utilizează aceasta. "Afinitatea SMP" poate fi

30 RFC 2883 – O extensie la Opțiunea de "Selective Acknowledgement" (SACK) pentru TCP

31 Documentată sumar în /usr/src/linux-2.4/Documentation/IRQ-affinity.txt

controlată manipulând fișierele din directorul `/proc/irq`. În acest director există subdirectoare ce corespund IRQ-urilor existente în sistem. Fiecare din aceste subdirectoare conține un fișier numit `smp_affinity`. Conținutul acestui fișier este o mască de biți reprezentând selecția de procesoare către care sunt rutate întreruperile pentru IRQ-ul respectiv. Spre exemplu:

```
# cat /proc/irq/10/smp_affinity
fffffff
```

Fiecare "f" din exemplul de mai sus reprezintă un grup de 4 procesoare, cel mai din dreapta grup fiind cel mai puțin semnificativ. Pentru a exemplifica ne vom limita numai la primele 4 procesoare (deși putem adresa un număr de până la 32). Pe scurt, luăm în considerare doar ultimul "f", considerând restul biților 0 (i.e. considerăm valoarea măștii: 0000000f). Reprezentarea binară a lui "f" este "1111", fiecare poziție din aceasta corespunzând unui procesor din sistem, ceea ce înseamnă ca valoarea 00001 (hex 1) corespunde procesorului CPU0, 0010 (hex 2) procesorului CPU1, 0100 (hex 4) procesorului CPU2 iar 1000 (hex 8) procesorului CPU3. Mască finală este rezultată dintr-o operație "SAU logic" între aceste patru valori. Deci dacă dorim ca toate întreruperile IRQ 10 să fie tratate de procesorul CPU0, vom scrie valoarea 1 în fișierul `/proc/irq/10/smp_affinity`:

```
# echo 1 > /proc/irq/10/smp_affinity
```

2.2.6 Efectul de colaps datorat congestiei și soluția NAPI

Mecanismul de întreruperi folosit la primirea de cadre conduce către un fenomen denumit "colaps datorat congestiei". Colapsul la congestie apare atunci când, deși un număr foarte mare de pachete pe secundă intră într-un router linux, nu mai iese nici un pachet. Testele au demonstrat că limita se situează în jur de 60 Kpps³² pentru un PC Pentium II ce rulează Linux 2.3.99. Aceleași teste au arătat de asemenea că Maximum Loss Free Forwarding Rate³³ (prescurtat MLFFR) se găsește în jur de 27 Kpps cu procesorul utilizat 100% în procesare de networking și blocând astfel orice fel de procesare în spațiul utilizator. Acest comportament nedorit se datorează fenomenului numit *interrupt livelock*: pentru fiecare pachet care intră în sistem este generată o întrerupere și este pierdută o fracțiune de timp. La un număr foarte mare de întreruperi, procesorul nu mai are timp pentru a efectua nimic altceva. Astfel se atinge punctul de colaps datorat congestiei.

Pentru a elimina acest neajuns a fost necesară găsirea unui mecanism care să prevină întreruperea procesorului de către placa de rețea. Rezolvarea acestei probleme s-a materializat în strategia NAPI³⁴, strategie adoptată de majoritatea device driverelor din versiunile de linux 2.6.x.

32 Kpps = kilo pachete pe secundă

33 Numărul maxim de pachete pe secundă la care pachetele sunt expediate fără nici o pierdere

34 NAPI este prescurtarea de la New Api

Obiectivele luate în considerare la proiectarea strategiei NAPI au fost:

1. menținerea paralelismului și scalabilității oferite de softnet
2. eliminarea re-ordonării pachetelor în sistemele SMP
3. reducerea numărului de întreruperi la supraîncărcare astfel încât să se obțină un palier de maxim când se ajunge la MLFFR.
4. mecanisme de tip Drop-Early la supraîncărcare
5. înlăturarea sau reducerea problemelor de inechitate
6. echilibru între latență și throughput
7. independență de hardware

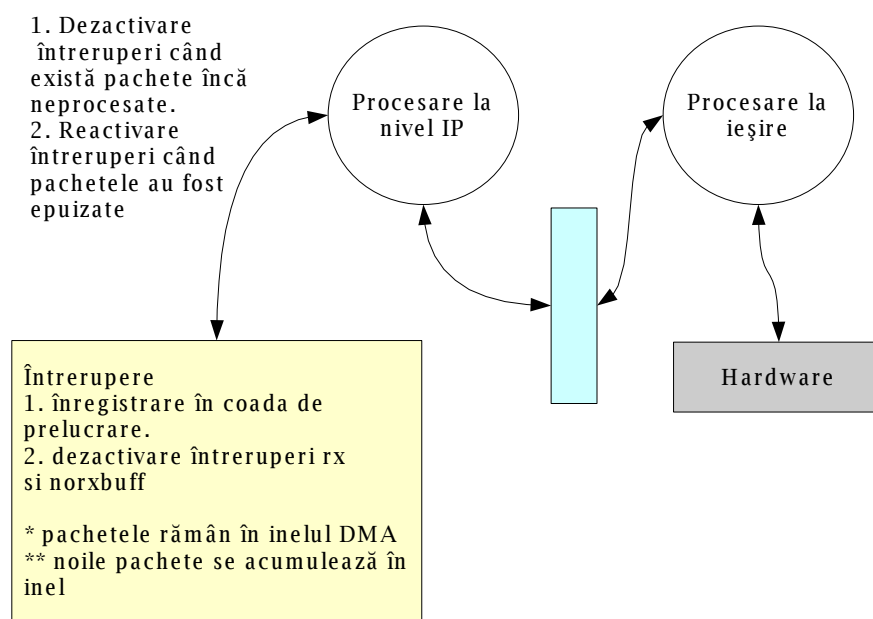


Figura 10 Soluția NAPI

NAPI este constituit dintr-o combinație între întreruperi și mecanisme de interogare³⁵. Deși interogarea este utilă pe cazul de încărcare excesivă, aceasta introduce latență mare în cazul de încărcare mică și putem afirma chiar că abuzează de procesor pentru a interoga device-urile care nu au nici un pachet de oferit. Pe de altă parte întreruperile scad latența sub încărcare mică, dar fac sistemul vulnerabil la efectul de livelock atunci când numărul de întreruperi depășește MLFFR.

Datorită acestor considerații, NAPI folosește o cale de mijloc. Interfețelor le este permis să producă întrerupere la sosirea primului pachet dintr-o serie. Apoi acestea se înregistrează într-o listă de procesare. Ulterior, interfețele dezactivează orice întrerupere ce poate fi cauzată de primirea unui pachet nou sau de rămânerea fără buffere de primire într-un inel DMA. Orice pachet sosește după ce inelul DMA este umplut va fi respins fără a întrerupe sistemul (obiectivul nr. 4).

La un moment ulterior, se activează un softirq pentru a interoga toate interfețele care au anunțat că au de oferit pachete. Tuturor interfețelor li se acordă oportunitatea de a trimite pachete în limita unui număr configurabil, cunoscut sub denumirea de *quota*. O dată

³⁵ În limba engleză polling

ce această limită este depășită, interfața este înlănțuită la loc la sfârșitul listei de procesare, dacă aceasta mai are de oferit pachete, altfel ea este scoasă din lista de procesare și i se permite să genereze întreruperi din nou. Folosirea quotei produce efectul de echitate între interfețe și este în acord cu obiectivul nr. 5 prezentat mai sus.

Sub o încărcare excesivă sistemul interoghează interfețele înregistrate. În acest sens este atins și obiectivul nr.3, în sensul că MLFFR este dependent de performanțele sistemului și că întreruperile sunt activate în măsura în care sistemul le poate procesa. Singura cerință necesară este ca interfețele să posede hardware DMA. Această abilitate este comună în zilele noastre, deci și obiectivul nr.7 este îndeplinit. Oricum, pentru a păstra compatibilitate cu hardware-ul mai vechi, ce nu e capabil de DMA, a fost păstrată și vechea interfață pentru drivere.

Așa cum putem observa din Figura 10, coada de backlog a dispărut complet. În schimb, pachetele sunt lăsate în inelul hardware DMA. Aceasta înlesnește serializarea pachetelor către sistem și în consecință atingerea obiectivului nr.2.

Cerința nr. 6 este îndeplinită și ea, deoarece NAPI comută între modul de întreruperi și cel de interogare. Sub încărcare mică, înainte de atingerea MLFFR, sistemul converge către un sistem ce este comandat de întreruperi, astfel încât raportul număr de pachete/întreruperi este mult mai mic și latența este redusă. În cazul unui sistem cu încărcare foarte mare, raportul număr de pachete/întreruperi este mai mare și latența este mărită.

Îndeplinirea obiectivul nr. 1 a fost probată experimental, după cum vom arăta în continuare.

2.2.7 Performanțele algoritmului NAPI

Configurația aleasă pentru experiment este evidențiată în Figura 11:

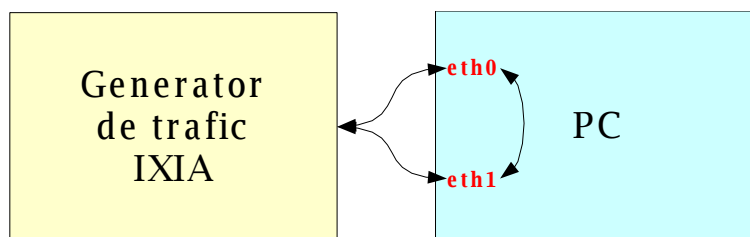


Figura 11 Configurație Experimentală

Echipamentul IXIA este capabil să măsoare throughput-ul, latența și reordonarea pachetelor. PC-ul este un dual PII 350 Mhz, placă de bază ASUS cu 128 M de memorie RAM. Plăcile de rețea sunt Zynx 4-port 32-bit 33 Mhz PCI.

Au fost folosite trei implementări de driver tulip din kernel 2.4.7: Plain (driver-ul tulip obișnuit), FF și NAPI.

Sistemul nu a avut încărcare produsă de alți factori decât de traficul generat.

Când s-a încercat emularea unui sistem uniprocessor, interfețele eth0 și eth1 au fost ambele asociate unuia dintre procesoare prin IRQ affinity.

S-au rulat teste de throughput, latență și reordonare a pachetelor atât pentru configurația SMP cât și pentru cea uniprocessor.

Rezultatele testelor de throughput:

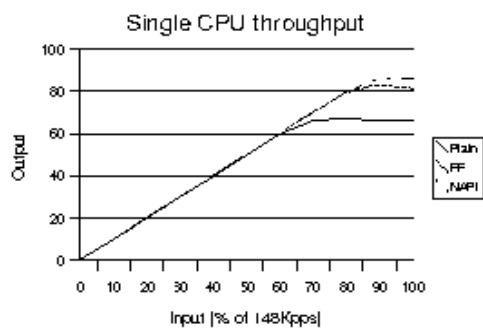


Figura 12 Rezultate throughput pentru un singur procesor

Testul nu a măsurat utilizarea CPU, însă s-a constatat că listarea unui director a durat aproximativ de 3-4 ori mai mult cu versiunea Plain decât cu versiunea NAPI.

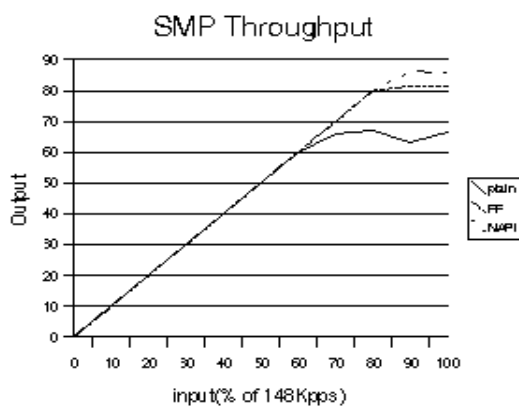


Figura 13 Rezultate throughput pentru SMP

Rezultatele obținute pe configurația SMP sunt similare cu cele din cazul uniprocessor.

Rezultatele testelor de latență:

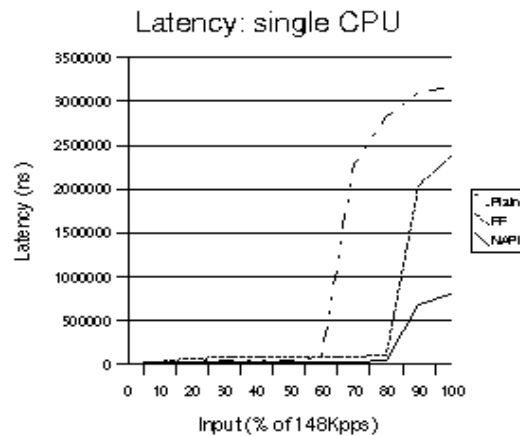


Figura 14 Rezultate latență pentru un singur procesor

De aici putem observa că NAPI scalează foarte bine, cu latențe sub 1 ms în cazul cel mai defavorabil. Driver-ul obișnuit, așa cum era de așteptat prezintă latențele cele mai mari, apropiindu-se de 4 ms în cazul cel mai defavorabil.

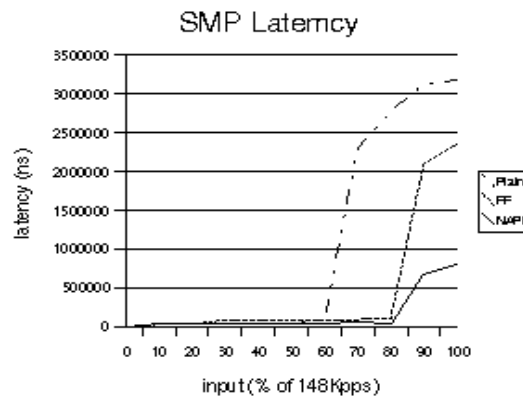


Figura 15 Rezultate latență pentru SMP

Ca și în testul anterior, se obțin rezultate asemănătoare pentru configurația SMP.

Rezultatele testului de reordonare de pachete:

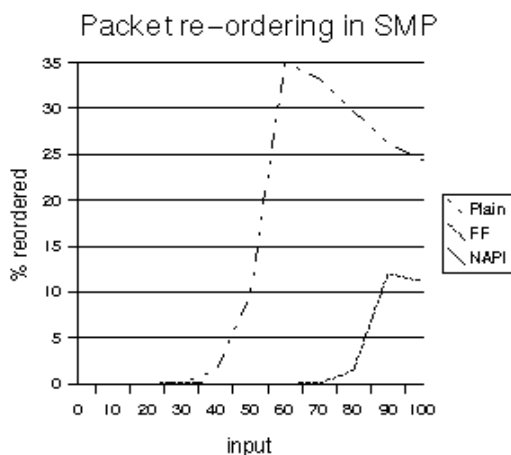


Figura 16 Rezultate reordonare pachete

În acest test au fost trimise 17203 pachete cu numere de secvență incrementale. La primire au fost contorizate toate pachetele ce nu au sosit în ordinea numărului de secvență.

Așa cum era de așteptat nu a existat nici o problemă de reordonare în cazul NAPI.

2.2.8 Implementarea Socket Bufferelor în Linux

Un socket buffer (*sk_buff*) este o structură de control ce are atașat un bloc de memorie. Există două categorii de funcții în biblioteca *sk_buff*. Mai întâi funcții ce manipulează liste dublu înlănțuite de structuri *sk_buff* și apoi funcții de gestionare a memoriei atașate. Bufferele sunt organizate în liste înlănțuite optimizate pentru operațiile frecvente de adăugare la sfârșit și extragere de la început. Deoarece mare parte din funcționalitatea de networking se execută în întreruperi, aceste rutine au fost proiectate să fie atomice.

Operațiile pe liste sunt folosite pentru a gestiona grupuri de pachete la sosirea de pe rețea precum și la trimiterea către interfețele fizice. Rutinele de manipulare a memoriei asociate sunt folosite pentru gestionarea conținutului pachetelor într-o manieră standardizată și eficientă.

Un exemplu mult simplificat de gestionare a unei liste de buffere ar arăta astfel:

```

/* Înlanțuire socket buffer în listă */
void append_frame(char *buf, int len) {
    struct sk_buff *skb = alloc_skb(len, GFP_ATOMIC)
    if (skb == NULL)
        dropped++;
    else {
        skb_put(skb, len);
        memcpy(skb->data, data, len);
        skb_append(&my_list, skb);
    }
}

```

```

/* Extragere socket buffer din coadă și procesare */
void process_queue(void) {
    struct sk_buff *skb;
    while ((skb = skb_dequeue(&my_list))!=NULL) {
        process_data(skb);
        kfree_skb(skb, FREE_READ);
    }
}

```

Aceste două fragmente de cod simplificate descriu destul de realistic mecanismul de primire de pachete. Funcția *append_frame()* este similară codului apelat dintr-o întrerupere de către un device driver la primirea unui pachet, iar *process_frame()* este similară codului apelat pentru a trimite date către handlerii de protocoale.

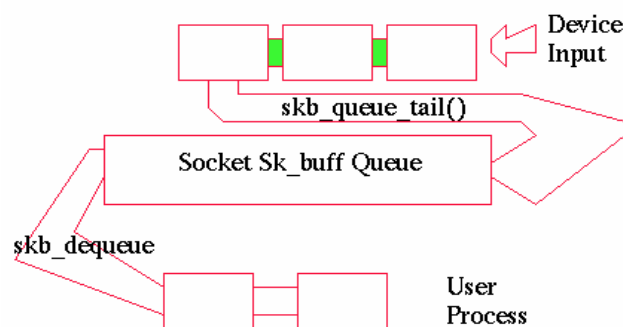


Figura 17 Fluxul datelor la recepție

Blocul de memorie asociat³⁶ unei structuri *sk_buff* este compus din 3 zone: o zonă neutilizată la început, numită *headroom*, apoi o zonă de date, în care este stocat cadrul, urmată de o zonă neutilizată la sfârșit, numită *tailroom*. Pentru delimitarea acestor zone

³⁶ Practic zona de stocare a cadrului, zona de date utile.

există 4 pointeri în structura *sk_buff*:

- head: pointer la începutul zonei de headroom
- data: pointer la începutul zonei unde este stocat cadrul
- tail: pointer la sfârșitul zonei de date și începutul zonei de tailroom
- end: pointer la sfârșitul zonei tailroom

Zona de headroom este special rezervată pentru ca diversele protocoale să aibă suficient spațiu pentru a-și putea adăuga antetele la începutul cadrului. Pentru aceasta există funcția *skb_push()* ce mută pointerul de început al cadrului (data) către adrese mai mici de memorie.

Imediat după ce un buffer a fost alocat, tot spațiul disponibil se găsește la sfârșit. O altă funcție, numită *skb_reserve()* poate fi apelată pentru a specifica faptul că o parte din spațiul disponibil ar trebui să fie la început. Astfel, marea majoritate a rutinelor de trimitere conțin apeluri de genul:

```
skb = alloc_skb(len+headspace, GFP_KERNEL);
skb_reserve(skb, headspace);
skb_put(skb, len);
memcpy_from_fs(skb->data, len);
pass_to_m_protocol(skb);
```

În unele sisteme de operare precum BSD Unix, nu se poate ști dinainte ce spațiu va fi necesar, deoarece acestea folosesc pentru bufferele lor de rețea lanțuri de buffere mici (mbufs). Linux optează pentru buffere liniare și prealocare de spațiu, câteodată irosindu-se câțiva bytes pentru a avea spațiu suficient pe cazul cel mai defavorabil, dar câștigându-se mult la capitolul eficiență.

În Figura 18 se prezintă modul în care este afectată zona de date a unui cadru de câteva dintre funcțiile de lucru cu socket buffere:

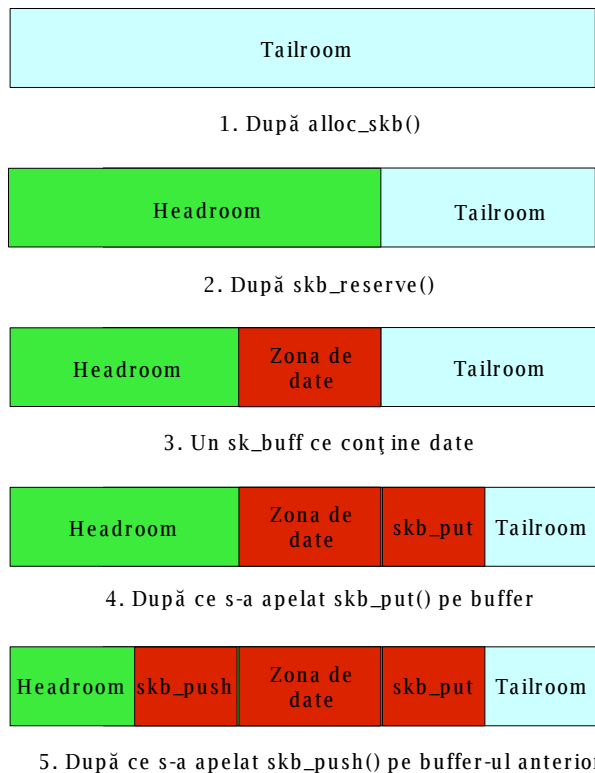


Figura 18 Funcții de manipulare a zonei de date din structurile `sk_buff`

Alte funcții utile:

- `skb_clone`: alocă spațiu pentru o nouă structură `sk_buff`. Câmpurile structurii `sk_buff` care se clonează (inclusiv pointerii la zona de date) sunt copiate în noua structură alocată. Zona de date asociată socket buffer-ului va rămâne partajată între cele două socket buffere. Funcția setează flag-ul `cloned` în cele două structuri rezultate. Folosirea acestei funcții este utilă atunci când se dorește utilizarea partajată a unui pachet, de exemplu atunci când se dorește trimiterea unui pachet pe mai multe interfețe, prevenindu-se eliberarea zonei de memorie de date utile asociate socket buffer-ului atât timp cât aceasta este partajată.
- `skb_copy`: diferența față de `skb_clone` este că aici se alocă spațiu și pentru zona de date utile, obținându-se o copie exclusivă a pachetului.
- `skb_copy_expand`: funcționează la fel ca `skb_copy`, cu mențiunea că în plus alocă un extra spațiu în headroom, de dimensiune specificabilă printr-un parametru.

3 Arhitectura aplicației

În acest capitol vor fi prezentate din punct de vedere arhitectural toate componentele proiectului *LiSA*.

Așa cum se preciza mai devreme *LiSA* este prescurtarea de la "Linux Switching Appliance" și își propune realizarea unei platforme de comutare de pachete bazată în întregime pe sistemul de operare Linux.

Din punct de vedere structural, putem împărți proiectul în trei componente:

- un modul kernel (LMS, sau Linux Multilayer Switch).
- o aplicație de configurare interactivă (CLI, sau Command Line Interface).
- o mini - distribuție de Linux, optimizată pentru rularea pe sisteme dedicate.

3.1 Arhitectura Linux Multilayer Switch (LMS)

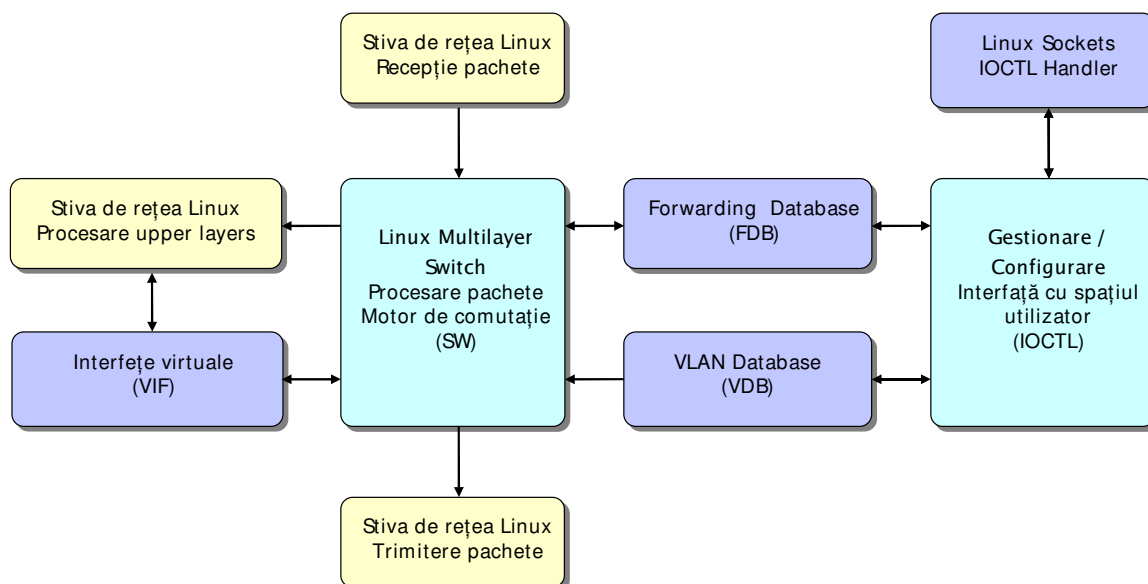


Figura 19 Arhitectura modului de kernel (LMS)

Așa cum se observă în Figura 19, modulul de kernel conține mai multe componente:

- *Motor de comutare*. Realizează funcția de bază, mai exact primirea de cadre, decizia de comutare și algoritmi de trimitere efectivă a cadrelor pe porturile switch-ului.
- *Tabelă de comutare* (Forwarding Database, sau FDB). Această componentă conține toate funcțiile de accesare și modificare a structurii de date folosite în implementarea tabelii de comutare a switch-ului.
- *Bază de date de VLAN-uri* (VLAN Database, sau VDB). Încapsulează toată funcționalitatea legată de modificarea și accesarea structurilor folosite în implementarea

bazei de date de VLAN-uri a switch-ului.

- *Interfețe virtuale* (Virtual Interfaces, sau VIF). Reprezintă implementarea unui *net_device* generic, precum și metodele asociate acestuia.
- *Componenta de interfațare cu spațiul utilizator* (IOCTL). Aici sunt tratate comenzile de configurare primite din spațiul utilizator.

3.2 Arhitectura aplicației de configurare (CLI)

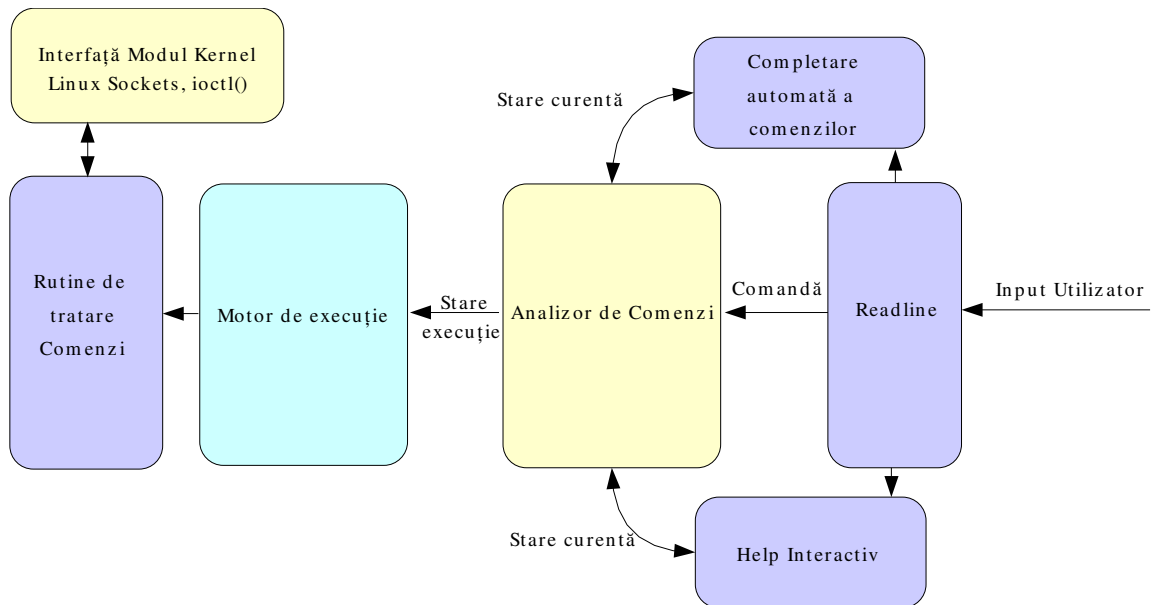


Figura 20 Arhitectura CLI

Aplicația de configurare a switch-ului are următoarele componente:

- *Readline*. Pentru citirea comenzilor a fost folosită librăria *readline*. Avantajele sunt numeroase: posibilitatea de reținere a istoricului comenzilor, facilități de implementare a completării automate a comenzilor, posibilitatea de asociere a unor taste sau combinații de taste la anumite funcții de tratare etc.
- *Analizor de Comenzi*. Interacționează cu alte trei componente: completarea automată a comenzilor, help-ul interactiv și motorul de execuție de comenzi. Analizorul primește ca parametru de intrare o comandă (completă sau parțială) și în funcție de componenta din care a fost invocat produce la ieșire o stare curentă de completare sau o stare de execuție.
- *Motor de execuție comenzi*. Primește ca parametru de intrare starea de execuție determinată de analizor și apelează rutina corespunzătoare de tratare a unei comenzi.
- *Completare automată de comenzi*.
- *Sistem de help interactiv*.

3.3 Arhitectura unui sistem ce rulează LiSA

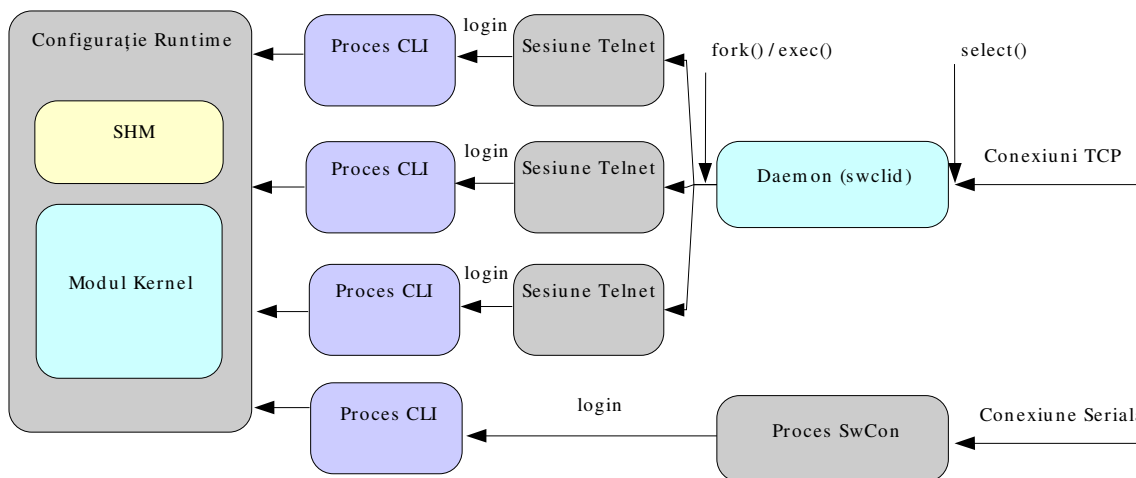


Figura 21 Arhitectura unui sistem LiSA

În Figura 21 este ilustrat modul de funcționare a unui sistem ce are instalat ca sistem de operare un kernel linux compilat cu suport pentru modulul LMS și are instalat pachetul de instrumente de configurare LiSA - CLI.

Sistemul are posibilitatea de a fi configurat atât printr-o conexiune pe portul serial cât și de la distanță printr-o sesiune de telnet. Facilitatea de configurare pe portul serial este foarte importantă în stadiul inițial, atunci când switch-ul este neconfigurat. În această etapă este necesar ca administratorul să stabilească parolele de acces la sistem și să definească o interfață virtuală căreia să-i atribuie o adresă de IP pentru management, acest lucru facilitând configurarea ulterioară de la distanță.

Pentru accesul de la distanță în sistem rulează un daemon ce ascultă pe portul 23³⁷ și multiplexează conexiunile primite. Pentru fiecare conexiune primită se creează un proces fiu ce va executa un program de autentificare (swlogin). Dacă autentificarea se realizează cu succes, programul de login va lansa în execuție programul de gestiune al configurației (CLI).

În cazul configurării de pe o conexiune serială, programul swcon va lansa în execuție programul de autentificare doar în cazul în care sistemul a trecut deja printr-o configurare inițială, când au fost stabilite parolele de acces.

Configurația de start a sistemului este păstrată într-un fișier text. La pornirea sistemului se rulează un program³⁸ care încarcă această configurație. Pentru simplitate, fișierul în care este salvată configurația de start conține comenzi CLI, iar încărcarea presupune doar citirea linie cu linie a fișierului și apelarea analizorului de comenzi al aplicației CLI. În ceea ce privește configurația de rulare a sistemului, aceasta este ținută în două locuri:

³⁷ Port asociat uzual serviciului telnet.

³⁸ Programul se numește swcfgload și într-adevăr este rulat la pornirea sistemului pe mini-distribuția de linux LiSA. Dacă se rulează LiSA peste o altă distribuție de linux, administratorul va trebui să configureze script-urile de pornire astfel încât să ruleze acest program.

- în modulul de kernel. Aici sunt ținute datele legate de configurația activă a sistemului (configurația porturilor, vlan-uri, mac-uri statice, de multicast, interfețe virtuale, adrese IP etc.).
- într-un segment de memorie partajată din spațiul utilizator. Aici sunt ținute datele legate de configurația pasivă a sistemului (parole de acces, hostname, numărul și configurația terminalelor virtuale(vty), lista sesiunilor de acces).

4 Detalii de implementare

În acest capitol se vor prezenta pe rând părțile componente ale proiectului *LiSA*, problemele apărute, soluțiile și optimizările introduse.

4.1 Implementarea modulului Linux Multilayer Switch (LMS)

Componenta de bază a proiectului, fără de care nu s-ar putea discuta de funcționalitate de comutare de pachete, este un modul pentru kernelul sistemului de operare Linux.

În continuare se va prezenta justificarea pentru implementarea acestui modul. Trebuie menționat că în kernelul Linux există deja un modul care oferă funcționalitate de comutare pachete la nivel legătură de date, și anume modulul *bridge*. Apare aici întrebarea justificată: de ce un modul nou, când există deja cod în kernel ce poate face comutare de pachete?

Răspunsul este simplu. Obiectivul principal al proiectului *LiSA* este realizarea unei implementări de comutare de pachete cât mai *eficientă*. Switch-ul realizat trebuie să aibă suport pentru VLAN-uri și trebuie să fie capabil să realizeze rutare de pachete între VLAN-uri. În kernel-ul linux, funcționalitatea de comutare de pachete (modulul *bridge*) și suportul pentru VLAN-uri 802.1q (modulul 8021q) sunt separate. Desigur, utilizând cele două module se poate obține funcționalitatea unui switch de nivel 3, însă faptul că cele 2 module sunt independente unul de celălalt rezultă într-o abordare total ineficientă.

Funcționalitatea modulul 802.1q constă în crearea de interfețe (sau *net_device*-uri) ce realizează adăugarea și scoaterea marcajului de VLAN la nivelul cadrelor Ethernet. Folosirea acestui modul presupune crearea a câte o interfață virtuală pentru primirea de cadre cu marcaj 802.1q pentru fiecare VLAN. Modulul *bridge* oferă posibilitatea asocierii mai multor interfețe (*net_device*) într-o interfață logică, sau *bridge* (care este de asemenea un *net_device*).

Dezavantajele pe care le implică o astfel de abordare sunt:

- pentru ca broadcast-urile să nu fie vizibile între VLAN-uri trebuie să se creeze un *bridge* (*net_device*) pentru fiecare VLAN.
- în cazul porturilor în trunchi trebuie creată câte o interfață virtuală capabilă să primească cadre marcate pentru fiecare VLAN.
- efectuarea de copieri inutile de socket buffere la adăugare / ștergere marcaj VLAN, în cazul comutării unui cadru între două porturi în mod trunchi. În acest caz se va face o eliminare de marcaj pe primul port urmată de o adăugare de marcaj pe al doilea, deși nu

ar fi fost necesară nici o modificare a cadrului, deoarece acesta ar fi trebuit să circule cu marcaj de VLAN prin ambele porturi. Se va arăta ulterior că strategia algoritmului *LiSA* de comutare este de a realiza un număr minim de copieri de cadre, în cazul cel mai defavorabil realizându-se o singură copiere.

- fiecare interfață virtuală de VLAN (pe care cadrul circulă cu marcaj 802.1q) își va face o copie exclusivă a cadrului pe care o va modifica (adăugare sau extragere de marcaj VLAN). Această abordare este inefficientă în cazul de multicast/broadcast.
- suport inexistent pentru adrese de multicast și pentru adrese MAC statice.

Implementarea *LiSA* elimină aceste dezavantaje, realizând o abordare elegantă și eficientă în raport cu toate problemele menționate mai sus. În același timp a fost urmărită integrarea modulului cu subsistemul de rețea din Linux și utilizarea pe cât posibil a tuturor facilităților deja existente în acesta, precum abstractizările oferite prin structurile *net_device*, *sk_buff* etc.

4.1.1 Interfațarea cu subsistemul rețea din Linux Kernel

Pentru a putea integra funcționalitatea oferită de modulul switch în subsistemul de rețea Linux, a fost necesară crearea unui punct de intrare în codul de procesare a cadrelor primite. Un astfel de punct de intrare mai este cunoscut în terminologia folosită în kernel-ul Linux sub numele de *hook*.

Acest *hook* a fost plasat în rutina *netif_receive_skb()* ce realizează funcția de procesare a cadrelor primite. Aceasta este apelată din codul device driverelor pentru plăci de rețea, după recepționarea completă a cadrului, atunci când acesta trebuie procesat.

Așa cum a fost arătat în secțiunea 2.2.6, în cazul abordării NAPI fiecare *net_device* are o metodă de interogare, numită *poll()* ce procesează coada de primire. Din metoda de *poll* a device-ului se apelează funcția de primire pachet specifică driver-ului, care rezervă spațiu pentru un socket buffer și zona de date asociată în care copiază cadrul, pe care apoi îl pasează rutinei de procesare *netif_receive_skb*. Pentru exemplificare se va ilustra imaginea stivei de apel în cazul unui device driver pentru o placă de rețea Realtek 8139:

```
rtl8139_poll()
rtl8139_rx()
dev_alloc_skb()
eth_copy_and_sum()
eth_type_trans()
netif_receive_skb()
```

Rutina *netif_receive_skb* se găsește complet în afara codului driver, care este dependent de hardware-ul plăcii de rețea. Un punct de intrare în această rutină presupune o adăugare a unei funcționalități generice, independente de hardware. Totuși arhitectura modulară a kernel-ului linux impune ca introducerea acestui *hook* în codul de primire de pachete să poată fi configurabilă.

Această arhitectură modulară impune mai multe restricții asupra unei funcționalități noi introduse:

- trebuie să fie posibilă dezactivarea unei facilități fără a se afecta funcționalitatea codului în lipsa acesteia.
- trebuie să existe posibilitatea ca acea facilitate să poată fi adăugată și eliminată în timp ce kernel-ul rulează fără a fi necesară o repornire a sistemului.
- codul adăugat să poată fi integrat în imaginea de kernel (built-in).

Abordarea folosită în kernel-ul linux este folosirea unui fișier de configurare ce conține mai mulți identificatori, câte unul pentru fiecare secțiune de cod opțională, aceștia având trei valori posibile:

- N - funcționalitatea este complet dezactivată.
- Y - funcționalitatea este integrată în imaginea de kernel, sau built-in. Adăugarea de suport pentru diverse facilități în acest mod presupune recompilarea imaginii de kernel și repornirea sistemului.
- M - funcționalitatea este compilată ca o facilitate opțională și poate fi încărcată și descărcată în timpul rulării.

Opțiunile sunt grupate logic în categorii și subcategorii, iar pentru configurarea acestora există mai multe metode ce pot fi folosite printre care editarea "de mână"³⁹ a fișierului .config din rădăcina surselor de kernel sau configurarea interactivă prin execuția uneia dintre comenzile: make config, make menuconfig, make xconfig, make gconfig.

În cazul modulului switch această opțiune a fost introdusă în categoria *Device Drivers/Networking support/Networking options*:

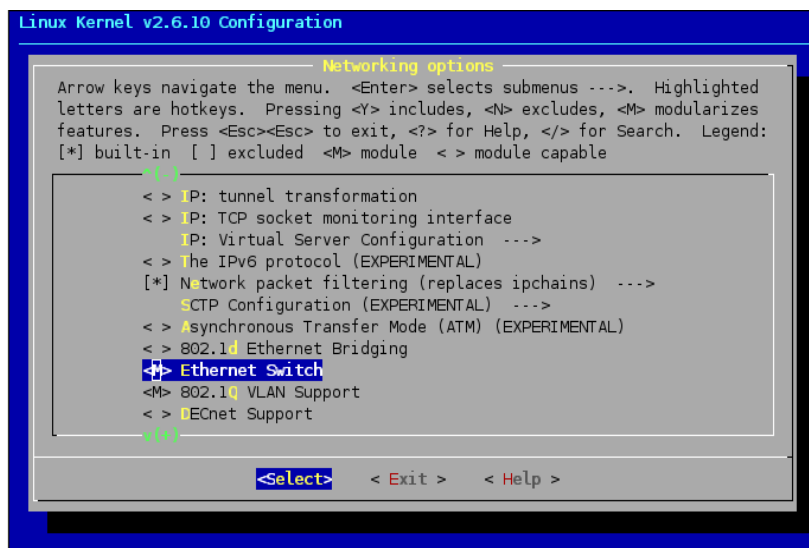


Figura 22 Configurarea suportului pentru modulul switch (LMS)

Funcția hook apelată din *netif_receive_skb* apelează funcția *handle_frame* din modulul switch, care reprezintă punctul de intrare în acesta pe cazul de procesare pachete

³⁹ Poate fi extrem de dificilă, având în vedere numărul mare de opțiuni prezente în kernelele actuale.

la primire.

În cazul trimiterii de pachete, lucrurile sunt mult mai simple și nu necesită modificări în codul de kernel existent. Trimiterea de pachete se face prin intermediul cozilor de trimitere generice dintr-un `net_device`.

O dată stabilită interfața pe care trebuie să iasă un pachet singurul lucru ce trebuie făcut este modificarea pointer-ului la interfață (`net_device`) din structura `sk_buff` corespunzătoare pachetului și apelarea unei funcții ce înlănțuie bufferul în coada de trimitere a device-ului respectiv (`dev_queue_xmit`).

Desigur, încă un aspect ce trebuie menționat este acela că o dată adăugat, un astfel de *hook* trebuie să trateze cadrele venite numai pe anumite interfețe din sistem. Aici apare conceptul de adăugare a unei interfețe în switch. Modulul switch va realiza comutare de pachete numai între acele interfețe înregistrate la el. Acest lucru trebuie să fie configurabil printr-un utilitar din spațiul utilizator. Practic, diferențierea între un `net_device` care face parte din switch (pentru care ar trebui să se execute algoritmul de comutare) și un alt `net_device` ce nu face parte din switch s-a realizat prin introducerea unui pointer la o structură numită `net_switch_port`⁴⁰ în definiția structurii `net_device`. Această structură precum și altele vor fi prezentate mai pe larg într-o secțiune următoare.

4.1.2 Structurile de date folosite

Pentru o înțelegere cât mai bună a modului de funcționare a acestui modul de kernel se vor prezenta structurile de date cele mai semnificative ce stau la baza funcționării acestuia.

Structura de date ce constituie unitatea de bază a modulului switch se numește `net_switch` și are următoarea formă⁴¹:

```
struct net_switch {
    /* Lista tuturor porturilor din switch */
    struct list_head ports;
    /* Tabela de comutare (forwarding database)*/
    struct net_switch_bucket fdb[SW_HASH_SIZE];
    /* Baza de date de VLAN-uri */
    struct net_switch_vdb_entry *vdb[SW_MAX_VLAN+1];
    /* Lista de interfețe virtuale pentru VLAN-uri */
    struct list_head vif[SW_VIF_HASH_SIZE];
    ...
};
```

Această structură pune în evidență componentele principale ale modulului switch:

- o listă înlănțuită de porturi asociate.
- o tabelă de comutare implementată printr-o tabelă de dispersie (hashtable).

⁴⁰ Acest pointer va avea valoarea implicită NULL, și va fi inițializat cu adresa unei structuri `net_switch_port` numai la adăugarea interfeței în switch.

⁴¹ Aici ca și în prezentarea altor structuri vor fi evidențiate câmpurile cele mai importante.

- o bază de date de VLAN-uri (de asemenea hashtable).
- în plus, din nevoia de a realiza rutare de pachete între VLAN-uri, au apărut interfețele virtuale, care sunt `net_device`-uri, câte una pentru fiecare VLAN pentru care se dorește rutare.

O structură de date importantă este `net_switch_port`. După cum se arăta mai devreme, această structură reprezintă legătura dintre o interfață fizică și modulul switch.

```

struct net_switch_port {
    /* Legătura cu alte porturi */
    struct list_head lh;

    /* Interfața fizică asociată portului */
    struct net_device *dev;

    /* Switch-ul de care aparține */
    struct net_switch *sw;

    ...
};

```

O astfel de structură conține un pointer la interfața fizică asociată și un pointer la structura switch din care face parte. Acești pointeri au fost introduși pentru a putea obține ușor referințe la structurile menționate în cazul în care din contextul de apel nu se dispune decât de un pointer la o structură de tip port. Trebuie menționat că și în structura de date `net_device` asociată interfeței fizice există un pointer înapoi la port. Dacă acesta conține o valoare nenulă atunci cadrele care sosesc pe acea interfață trebuie tratate de către algoritmul de comutare și nu mai ajung să fie procesate de protocoalele de nivel superior. În caz contrar cadrele sunt ignorate de `hook-ul handle_switch` din `netif_receive_skb` și ajung mai departe la protocoalele de nivel superior pentru procesare.

Pe lângă câmpurile prezentate mai sus, un port mai încapsulează și alte informații folosite de algoritmul de comutare de pachete. Printre acestea se pot enumera:

- un bitmap de indicatori (sau *flags*) ce conține informații despre starea portului (activat/dezactivat), modul de funcționare al portului (access/trunk), viteza (10 Mbps, 100Mbps, 1000Mbps half-duplex sau full-duplex). Precizăm ca din punctul de vedere *LiSA*, un port poate avea două moduri de funcționare: access (portul face parte dintr-un VLAN) sau trunchi (prin interfața asociată portului cadrele circulă cu marcaj de VLAN). În acest ultim caz portul acceptă numai cadre cu marcaj de VLAN dintr-o mulțime de VLAN-uri permise. Setul de VLAN-uri permise este dat de un bitmap reținut în membrul `forbidden_vlans` descris mai jos.
- `vlan`: VLAN-ul din care face portul în cazul în care modul de funcționare este `access`.
- `forbidden_vlans`: bitmap-ul negat al VLAN-urilor permise pe acest port atunci când funcționează în mod trunchi.
- `desc`: identificator alfanumeric ce reprezintă descrierea portului.

4.1.3 Tabela de comutare (FDB)

Funcționalitatea unui switch la nivelul legătură de date constă în luarea deciziei de comutare a cadrului recepționat pe unul din porturile sale către zero, unul sau mai multe porturi. Decizia de comutare este luată de switch după ce analizează o structură internă de date numită tabelă de comutare. Prescurtarea *FDB* provine din termenul *Forwarding Database*, utilizat pentru a desemna această tabelă de comutare.

Având în vedere că pentru fiecare cadru sosit pe o interfață a switch-ului se analizează această tabelă, este foarte important ca accesul la aceasta să fie cât mai rapid posibil.

Din aceste motive, structura de date aleasă pentru implementarea tabelii de comutare a fost o tabelă de dispersie.

Tabela de dispersie a fost implementată ca un tablou de *SW_HASH_SIZE* structuri de tip *net_switch_bucket* ce conțin capetele unor *liste* înlănțuite de structuri de tip *net_switch_fdb_entry*. Acestea din urmă conțin informații precum adresa MAC învățată, portul pe care se poate ajunge către ea precum și o informație de tip (intrările pot fi: *dinamice*, atunci când sunt introduse prin procesul de învățare, sau *statice* atunci când sunt introduse printr-o comandă de configurare din spațiul utilizator).

Funcția de dispersie este calculată folosind o adresă de MAC. Atunci când se alege o funcție de dispersie este foarte importantă evitarea coliziunilor. Printr-o coliziune se înțelege obținerea aceleiași valori calculate a funcției de dispersie pentru două valori diferite de chei (sau parametri de intrare). Funcția de dispersie folosită pentru tabela de comutare în *LiSA* se bazează pe operații logice și de deplasare pe biți asupra octeților dintr-o adresă MAC. Adresele MAC constituie o sursă de entropie suficient de bună, deoarece suntem asigurați că acestea sunt unice pentru fiecare adaptor Ethernet.

În aceste condiții se poate afirma că o operație de căutare pe tabela de comutare are o complexitate de $O(1)$. În aproape toate cazurile elementul căutat se găsește chiar în primul element al listei de pe poziția respectivă din tablou. Pe un caz defavorabil, în care se presupune că au existat coliziuni în funcția de dispersie elementul căutat se va găsi la o distanță constantă față de începutul listei, deci și în acest caz se poate afirma că operația de căutare are complexitate $O(1)$.

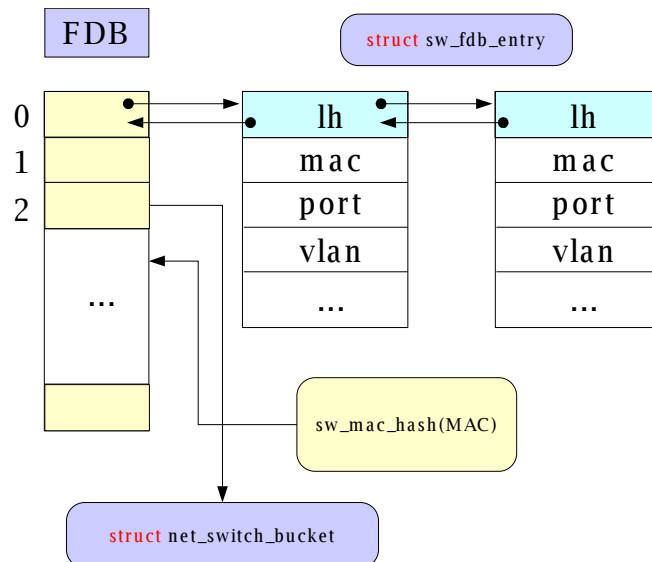


Figura 23 Structura tabelii de comutare

Fiecare element (bucket) din tabloul FDB conține un capăt de listă înlănțuită⁴². Sincronizarea la modificarea unei liste dintr-un bucket se face printr-un spinlock⁴³, acesta fiind și el membru în structura *net_switch_bucket*. Deoarece fiecare bucket dispune de spinlock-ul propriu, intrarea în zonă critică pe cazul de modificare a listei de intrări dintr-un bucket nu constituie o problemă de performanță pentru algoritmul de comutare, celelalte elemente ale tabloului fiind în continuare accesibile pe durata operației respective.

Structurile *sw_fdb_entry* sunt alocate dintr-un slab cache⁴⁴ deoarece operațiile de alocare și dealocare sunt destul de frecvente și s-a urmărit evitarea folosirii operațiilor costisitoare de alocare de memorie (cazul alocării și dealocării memoriei cu *kmalloc* și *kfree*). Tabloul FDB nu este alocat dinamic, așa că nici aici nu apare overhead generat de operații de alocare de memorie.

4.1.4 Operația de căutare și operațiile de modificare a listelor

Atunci când se lucrează cu liste trebuie avut în vedere că accesul la acestea se face concurrent. În concluzie trebuie folosite mecanisme de sincronizare specifice accesului concurrent la parcurgerea sau modificarea acestora.

În cazul tabelii de comutare este evident că majoritatea operațiilor asupra acesteia vor fi operații de căutare. În consecință, pentru maximizarea performanței, a trebuit să se folosească un mecanism ce favorizează viteza operațiilor de citire. Un astfel de mecanism

42 Toate listele înlănțuite folosite de LiSA sunt implementate cu listele generice din kernel. Pentru o descriere a acestora se recomandă consultarea fișierului *include/linux/list.h* din sursele de kernel.

43 Structură de sincronizare specifică kernel-ului linux. Pentru o descriere mai amănunțită a se consulta Anexa A.

44 Folosind alocatorul slab, există posibilitatea prealocării unor zone de memorie de lungime fixă. De obicei dimensiunea totală este multiplu al dimensiunii unei structuri specificate la construirea unui astfel de cache. Practic, atunci când se cere alocarea unei structuri se marchează zona corespunzătoare ca fiind ocupată, iar la dealocare se marchează din nou zona ca fiind free. Prealocarea de memorie și reciclarea zonelor eliberate elimină operațiile costisitoare de *kmalloc* și *kfree*. Cazul cel mai defavorabil este atunci când toate intrările din cache sunt ocupate, și se solicită o operație de alocare, moment în care se realocă zona de memorie asociată cache-ului la o dimensiune mai mare.

de sincronizare suportat în kernel-ul linux este algoritmul *RCU (Read Copy Update)*⁴⁵.

Pentru parcurgerea listelor generice din kernel există macro-uri predefinite, ce se bazează pe RCU și garantează siguranța accesului la listă fără să execute operații costisitoare de zăvorâre. Toate listele de elemente din tabela de comutare sunt sincronizate folosind algoritmul RCU. Astfel, operația de căutare în tabelă este extrem de simplă și eficientă:

```
/* Localizare bucket în O(1) */
struct net_switch_bucket *bucket =
    &sw->fdb[sw_mac_hash(skb->mac.raw)];
struct net_switch_fdb_entry *out;

if (fdb_lookup(bucket, skb->mac.raw, vlan, &out)
    { /* intrarea căutată se găsește în out*/ }

int fdb_lookup(struct net_switch_bucket *bucket,
    unsigned char *mac, int vlan,
    struct net_switch_fdb_entry **pentry) {
    struct net_switch_fdb_entry *entry;

    list_for_each_entry_rcu(entry, &bucket->entries, lh) {
        if (!memcmp(entry->mac, mac, ETH_ALEN) &&
            entry->vlan == vlan) {
            *pentry = entry;
            return 1;
        }
    }
    return 0;
}
```

În majoritatea cazurilor, dacă elementul căutat există, acesta se găsește în primul element din lista *bucket->entries*, iar în caz contrar lista va fi vidă și se va întoarce imediat valoarea 0 din funcție.

În cazul operațiilor de modificare a listelor, lucrurile nu mai sunt chiar atât de simple. La o operație de modificare trebuie să se localizeze mai întâi elementul din listă ce trebuie modificat, apoi să se realizeze efectiv modificarea. În cazul sincronizării listelor cu RCU, există o potențială problemă de consistență.

La o primă evaluare a problemei, dacă se ia exemplul operației de adăugare (sau învățare de adrese MAC, în cazul tabelii de comutare) algoritmul de adăugare mult simplificat ar putea fi descris în pseudocod astfel:

```
learn(mac, port, vlan) {
    bucket = fdb[hash(mac)];
    pentru fiecare element E din lista RCU {
        dacă (E.mac == mac și E.vlan == vlan) {
```

45 Vezi Anexa A.

```

        E.port = port;
        E.timestamp = now();
        return;
    }
}
bucket.lock();
E = new Element(mac, port, vlan);
add_tail(bucket.entries, E);
bucket.unlock();
}

```

Acest algoritm este incorect, deoarece există pericolul ca între ciclul de căutare și luarea lock-ului, pe cazul în care elementul nu este găsit, acesta să fie adăugat din altă parte. Mai există soluția în care se face lock pe bucket înainte de a începe căutarea elementului, însă aceasta este deficentă din punct de vedere al performanței. De exemplu, în cazul în care elementul există deja în listă bucket-ul este blocat inutil pentru ceilalți cititori.

Abordarea propusă de *LiSA* în astfel de cazuri este de natură tranzacțională. Prima parte a algoritmului de mai sus rămâne la fel, în sensul că nu se ia lock-ul pe cazul în care elementul se găsește deja în listă, însă în cazul în care nu este găsit și aceasta trebuie modificată, după luarea lock-ului se mai face încă o căutare pentru a verifica dacă nu cumva elementul a fost adăugat între timp.

Lock-ul asigură accesul exclusiv la listă, astfel că dacă din a doua căutare (care se execută cu lock-ul luat) rezultă că elementul nu există în listă, există certitudinea că nimeni nu l-a adăugat între timp și se poate efectua modificarea listei. Algoritm corect în pseudocod este:

```

learn(mac, port, vlan) {
    bucket = fdb[hash(mac)];
    pentru fiecare element E din lista RCU {
        dacă (E.mac == mac și E.vlan == vlan) {
            E.port = port;
            E.timestamp = now();
            return;
        }
    }
    bucket.lock();
    pentru fiecare element E din lista RCU {
        dacă (E.mac == mac și E.vlan == vlan) {
            E.port = port;
            E.timestamp = now();
            bucket.unlock();
            return;
        }
    }
    E = new Element(mac, port, vlan);
    add_tail(bucket.entries, E);
    bucket.unlock();
}

```

Toate celelalte operații care au ca rezultat modificarea unei liste din tabelă sunt realizate după algoritmi similari. Aceste măsuri de precauție au fost necesare deoarece listele pot fi modificate concurrent atât din context softirq cât și din context utilizator.

Factorii care contribuie la modificarea listelor din FDB sunt:

- operația de învățare adrese de MAC (fdb_learn).
- comenzile primite din spațiul utilizator (ioctl⁴⁶).
- expirarea cronometrelor asociate intrărilor dinamice din FDB.

4.1.5 Durata de viață a intrărilor din tabela de comutare

Pentru a reflecta cât mai bine realitatea, intrările *dinamice* din tabela de comutare au o durată de viață finită. Această durată este bineînțeles configurabilă și are valoarea implicită de 5 minute.

Procesul de expirare a intrărilor din tabelă este cunoscut sub numele de *Mac Aging* așa cum s-a precizat și în secțiunea 2.1.6 a lucrării.

Ca soluție de implementare a procesului de *Mac Aging* s-a ales folosirea timerelor⁴⁷ din kernelul Linux. Funcționarea acestora este similară cu cea a unui cronometru. O astfel de structură primește la inițializare trei parametri: o valoare întregă reprezentând timpul

⁴⁶ Mecanismul de configurare cu ioctl() va fi detaliat într-o secțiune următoare.

⁴⁷ structuri timer_list definite în include/linux/timer.h.

de expirare, o funcție de prelucrare ce va fi executată la expirarea timpului asociat și o adresă către o zonă de memorie de unde se vor lua parametrii pe care îi primește funcția respectivă la momentul apelului.

Rezolvarea oferită de *LiSA* este simplă și în același timp elegantă. Astfel, fiecare structură *net_switch_fdb_entry* are un membru de tip *timer_list* și un timestamp asociat. Atunci când se activează acest timer, parametrul pe care îl primește funcția handler este chiar adresa structurii *net_switch_fdb_entry*. De fiecare dată când se încearcă învățarea unei adrese MAC, dacă se constată că elementul există deja în tabelă se va efectua o actualizare a timestamp-ului din acesta. Astfel funcția handler asociată timer-ului de expirare nu va șterge intrarea asociată din listă decât în momentul în care diferența dintre timestamp-ul acesteia și momentul curent⁴⁸ va fi mai mare sau egală cu durata de viață menționată mai înainte. În caz contrar, tot din funcția handler, timer-ul se auto-reprogramează pentru o execuție ulterioară.

4.1.6 Algoritm de comutare la nivel legătură de date

După cum se preciza în secțiunea 4.1.1, legătura între sistemul de recepție de pachete din linux și modulul *LiSA* a fost realizată prin intermediul unui *hook*. Acest *hook* apelează funcția *sw_handle_frame*, care reprezintă punctul de intrare în algoritmul de comutare *LiSA*. Funcția *sw_handle_frame* primește ca parametri un pointer la portul de intrare al pachetului și un pointer la o structură de tip socket buffer.

Funcția *sw_handle_frame* nu face decât câteva verificări, după care cedează controlul algoritmului de comutare (funcția *sw_forward*).

Principalele operații executate înainte de apelul funcției *sw_forward* sunt:

- verificarea stării portului; dacă portul este dezactivat se dealocă socket buffer-ul și se iese imediat.
- completarea unei structuri ajutătoare ce conține două elemente: VLAN (se ia din antetul cadrului dacă acesta are marcaj 802.1q, altfel este egal cu VLAN-ul în care se găsește portul de intrare) și un flag care indică dacă pachetul are marcaj de VLAN sau nu.
- verificarea existenței vlan-ului în baza de date de VLAN-uri. În cazul în care acesta nu există se va elibera socket buffer-ul și se va ieși imediat.
- verificare ca în cazul în care cadrul sosește cu marcaj 802.1q portul de intrare să fie configurat în mod trunchi. In caz contrar se verifică dacă portul este configurat în mod acces. Nu se acceptă cadre cu marcaj pe porturi în mod acces sau cadre fără marcaj pe porturi configurate în mod trunchi.
- verificarea adresei MAC sursă pentru a se elimina unele anomalii: fie adresă nulă (toți biții zero), fie de broadcast (toți biții setați).
- actualizarea tabelii de comutare (se învață adresa MAC sursă a cadrului) și cedarea controlului algoritmului de comutare (*sw_forward*).

Structura algoritmului de comutare este destul de simplă. La început se tratează cazurile particulare: cadrul sosit poate fi destinat unei interfețe virtuale sau poate avea

⁴⁸ Prin moment curent se înțelege timpul kernel, măsurat în variabila *jiffies*. Similar prin actualizare timestamp se înțelege atribuirea valorii din variabila *jiffies* acestuia.

adresa MAC destinație de tip multicast.

Interfețele virtuale suportate de *LiSA* nu fac obiect de interes pentru această lucrare, așa că nu vor fi tratate în amănunt. Trebuie precizat însă că acestea au fost gândite în scopul realizării rutării de cadre între VLAN-uri.

Adresele de multicast constituie un caz particular deoarece sunt implementate ca adrese de MAC statice adăugate pe mai multe porturi sau VLAN-uri. Diferența constă în faptul că o aceeași adresă de MAC poate apărea pe mai multe porturi simultan, lucru ce nu se întâmplă în cazul adreselor de MAC obișnuite. Nu se va insista asupra algoritmului de comutare de cadre pentru acest tip de adrese, deoarece el este foarte asemănător cu cel de difuzare (sau flood) din cazul adreselor obișnuite.

La nivelul funcției *sw_forward* s-a făcut o separare logică între acțiunile executate pe diferite cazuri, acestea fiind tratate în funcții distincte. Astfel, scheletul algoritmului de comutare poate fi reprezentat într-o formă simplificată prin următorul pseudocod:

```
sw_forward(port_in, cadru, vlan) {
    /* Tratare cazuri VIF și multicast */
    port_out = caută (cadru.mac_dest, vlan) în FDB;
    dacă există port_out {
        dacă port_out==port_in
            drop(cadru);
        dacă mod_access(port_out) și vlan!=port_out.vlan
            drop(cadru);
        dacă mod_trunchi(port_out) și !permis(vlan)
            drop(cadru);
        __sw_forward(port_in, port_out, cadru, vlan);
    }
    altfel sw_flood(port_in, cadru, vlan);
}
```

După cum se poate observa din pseudocodul de mai sus, funcția *sw_flood* distinge două cazuri principale în cazul adreselor de MAC obișnuite.

Se caută obținerea portului de ieșire printr-o căutare în tabela de comutare. În cazul în care acesta este identificat, după ce în prealabil se fac câteva verificări preliminare, se apelează funcția ce realizează efectiv comutarea cadrului de pe portul de intrare pe portul de ieșire găsit. Verificările care se fac înainte de comutarea propriu-zisă sunt legate de filtrarea cadrelor destinate unei stații de pe același segment de rețea de pe care s-a recepționat cadrul, precum și filtrarea după VLAN⁴⁹.

În cazul în care se constată că adresa destinație nu este cunoscută se aplică strategia cunoscută sub numele de *difuzare* (sau *flooding*). În acest caz switch-ul va trimite cadrul recepționat pe toate celelalte porturi din același VLAN.

LiSA nu tratează special cazul broadcast-urilor. Algoritmul se bazează pe faptul că adresa MAC destinație ff:ff:ff:ff:ff:ff nu se va găsi niciodată în tabela de comutare⁵⁰. Astfel în cazul unui broadcast se execută ramura în care adresa de MAC destinație nu a fost găsită

49 Nu se permite comutarea cadrelor între VLAN-uri la nivel de legătură de date și nici trecerea unui cadru cu marcaj de VLAN printr-un port în trunchi ce nu permite trecerea cadrelor marcate cu acel VLAN.

50 verificările din *sw_handle_frame*, fac imposibil acest lucru.

în FDB, obținându-se exact efectul dorit.

În prealabil, trebuie făcute o serie de precizări despre problemele care apar și precauțiile ce trebuie luate atunci când se lucrează cu obiecte de tip socket buffer.

În primul rând este important să se facă distincția între structura *sk_buff* și zona de date asociată (pachetul propriu-zis).

Referitor la zona de date asociate unui socket buffer trebuie menționat că aceasta poate fi modificată de algoritmul de difuzare, atunci când comutarea se face de pe un port configurat în mod acces pe un port configurat în mod trunchi sau invers (prin modificare se înțelege adăugare sau eliminare de marcaj VLAN 802.1q).

Structurile *sk_buff* sunt alocate dintr-un *slab cache*, iar după trimiterea datelor asociate pe interfața fizică acestea sunt colectate și marcate ca libere pentru a putea fi reutilizate la o solicitare ulterioară de alocare a unui nou pachet.

Pentru o funcționare corectă, algoritmul de difuzare trebuie să facă cel puțin o operație de *clonare*⁵¹ a socket buffer-ului atunci când trebuie să livreze cadrul pe mai mult de un port. Operația de clonare va alocă spațiu pentru o nouă structură *sk_buff*, va copia toți membrii structurii inițiale în cea nouă (inclusiv pointerii la zona de date utile, ce va rămâne partajată între cele două structuri) și va seta flag-ul *cloned* în ambele structuri. Clonarea este necesară atât datorită colectării structurilor *sk_buff* după trimiterea efectivă, cât și datorită faptului că la fiecare trimitere se va modifica valoarea membrului *dev* din structura *sk_buff* la adresa *net_device*-ului în a cărui coadă se înlănțuie cadrul pentru trimitere. Privită superficial, o operație de trimitere a unui cadru pe un anumit port se reduce la următoarele operații:

```
skb->dev = port->dev;  
dev_queue_xmit(skb);
```

Clonarea structurii *sk_buff* este totodată o măsură de protecție împotriva eliberării zonei de memorie utile asociate. Dealocarea acesteia nu are loc atât timp cât mai există clone ale pachetului.

Lucrurile se complică atunci când se dorește modificarea zonei de date, și aceasta din simplul motiv pentru că există posibilitatea ca alte zone din kernel să utilizeze date din acea zonă. Pentru a ilustra problema se va descrie sumar structura funcției *netif_receive_skb* (unde a fost introdus *hook*-ul pentru switch):

```
netif_receive_skb(skb) {  
    ...  
    generic_handler_hooks(skb);  
    ...  
    handle_switch(skb);  
    ...  
    protocol_handler_hooks(skb);  
}
```

După cum se poate observa mai sus, înainte de intrarea în funcția *hook* a switch-ului, socket buffer-ul este trecut prin *hook*-urile de "handleri generici". Mai exact, există

⁵¹ Se realizează utilizând funcția *skb_clone()*.

programe care își pot înregistra în kernel o funcție generică pentru anumite prelucrări, funcție ce va obține o clonă de pachet. Un exemplu de astfel de program este binecunoscutul program de captură de pachete, *tcpdump*⁵². Bineînțeles, în acest caz algoritmul de comutare din *LiSA* partajează zona de date asociată socket buffer-ului cu acestea.

În cazul în care la trimitere este necesară modificarea zonei de date, trebuie testat dacă pachetul mai este utilizat din altă parte. În caz afirmativ modificările trebuie executate pe o copie⁵³.

Ca fapt divers, handlerii de protocoale, ce apar în funcția *netif_receive_skb* după hook-ul *LiSA*, reprezintă funcțiile de prelucrare pentru protocoalele de nivel superior (de exemplu *ip_rcv*).

Problemele care apar la trimiterea unui cadru vor fi separate pe cele două cazuri principale:

- comutare 1 la 1 (cazul în care portul de ieșire este determinat din tabela de comutare și se apelează funcția *__sw_forward*).
- comutare 1 la N (algoritmul de difuzare, când portul de ieșire este nedeterminat; se apelează funcția *sw_flood*).

Cel mai simplu caz este acela de comutare 1 la 1. Aici se pot distinge două posibilități:

- portul de intrare și cel de ieșire sunt ambele configurate în modul acces sau trunchi. Aici nu se face absolut nici o modificare asupra cadrului și nu se execută nici o clonare sau copiere de socket buffer.
- portul de intrare și cel de ieșire sunt configurate diferit. În acest caz se face o copiere de socket buffer doar dacă socket buffer-ul de care se dispune este utilizat și în altă parte (i.e. într-un handler generic precum *tcpdump*). În caz contrar, zona de date nu este partajată, iar modificarea se poate face direct pe aceasta.

4.1.7 Algoritmul de difuzare. Optimizări.

Din nefericire, în cazul comutării 1 la N lucrurile nu mai sunt la fel de simple. Aici, pentru $N > 1$, în cazul cel mai favorabil, sunt necesare $N-1$ clonări. Datorită optimizărilor din algoritmul folosit de *LiSA* pe cazul cel mai defavorabil se fac $N-2$ clonări și o copiere.

Algoritmul de difuzare funcționează la nivel de VLAN. Practic, atunci când un cadru trebuie difuzat, el va fi trimis către toate porturile configurate în mod acces din VLAN-ul respectiv și către toate porturile configurate în mod trunchi care acceptă cadre cu marcaj 802.1q din acel VLAN. Datorită modului de organizare a structurii folosite pentru baza de date de VLAN-uri, se pot obține ușor două liste înlănțuite cu aceste porturi. Astfel, problema se reduce la parcurgerea acestor două liste și efectuarea modificărilor corespunzătoare asupra cadrelor înainte de trimitere.

După cum se arăta anterior, trebuie făcute $N-1$ operații de clonare pentru o listă de

52 Pe lângă *tcpdump* poate fi orice program ce folosește biblioteca *libpcap*.

53 Se folosește funcția *skb_copy()*.

N elemente. Cu toate acestea, în cazul în care lista conține un singur element nu este necesară nici o clonare.

Pentru evitarea acestei probleme, algoritmul de difuzare face o "post-procesare" a listei. Prin post-procesare a listei se înțelege amânarea prelucrării unui element de listă până la pasul următor din parcurgere. Astfel elementele 1 .. N-1 sunt prelucrate din ciclul de parcurgere al listei, ultimul element (desigur, în cazul în care lista a fost nevidă) urmând să fie prelucrat în afara ciclului. Avantajul acestei abordări este că evită o clonare inutilă în cazul unei liste de un singur element.

În cazul în care lista a fost vidă este de datoria algoritmului să elibereze structura *sk_buff* primită. Altfel, aceasta ar fi fost eliberată implicit după expedierea efectivă a cadrului din coada de trimitere a *net_device*-ului.

O reprezentare simplificată în pseudocod a algoritmului de post-procesare a unei liste este:

```
post_procesare(skb, lista) {
    prev = NULL;
    pentru fiecare element E din lista {
        dacă prev != NULL {
            skb2 = skb_clone(skb);
            trimite(skb, prev);
            skb = skb2;
        }
        prev = E;
    }
    dacă prev != NULL {
        /* Lista a avut cel puțin un element */
        trimite(skb, prev);
    }
    altfel {
        /* Lista a fost vidă */
        eliberează(skb);
    }
}
```

Totuși, algoritmul de difuzare are de prelucrat nu doar o listă ci două liste, cu mențiunea că la primul element din a doua listă trebuie să aplice și o modificare asupra antetului cadrului (o adăugare sau extragere de marcaj VLAN). Practic, parcurgerea celei de-a doua liste se face tot cu post-procesarea elementelor, numai că algoritmul trebuie modificat puțin pentru a face o copie de pachet înainte de aplicarea funcției de modificare a cadrului. Trebuie precizat că nu întotdeauna este nevoie de o copie. De exemplu, în cazul în care prima listă este vidă, o copie de cadru ar fi inutilă, desigur cu excepția cazului în care cadrul este partajat cu un handler generic. Soluția este modificarea algoritmului, astfel încât să se amâne procesarea primului element din prima listă până la prima iterație în a doua listă.

Algoritmul *sw_flood* tratează problema într-o manieră generică. Nu se face distincția între lista de porturi în mod acces și lista de porturi în mod trunchi. Algoritmul primește ca parametri cele două liste și o funcție de prelucrare a cadrului (de adăugare,

respectiv extragere marcaj) diferită în funcție de ordinea în care se pasează listele. Desigur, metoda optimă de parcurgere a listelor este diferită în funcție de modul portului pe care a sosit cadrul inițial. Se disting aici două cazuri:

- cadrul a sosit inițial pe un port în mod trunchi. În această situație funcția *sw_flood* apelează *__sw_flood(trunk_ports, non_trunk_ports, strip_vlan_tag)*⁵⁴.
- cadrul a sosit inițial pe un port în mod acces. Aici funcția *sw_flood* va apela *__sw_flood(non_trunk_ports, trunk_ports, add_vlan_tag)*.

În concluzie, algoritmul de *flooding* face un număr minim de copii (zero în cazul cel mai favorabil, una în cazul cel mai defavorabil) și un număr minim de clonări (0 în cazul cel mai favorabil când numărul de elemente de procesat este mai mic sau egal cu 1 și N-2 în cazul cel mai defavorabil).

4.1.8 Interfațarea spațiu utilizator - kernel

Deoarece configurația *activă*⁵⁵ a switch-ului se găsește în totalitate în spațiul kernel, a fost nevoie de implementarea unei componente care să permită primirea și executarea unei serii de comenzi de configurare. Această componentă a fost proiectată astfel încât să fie flexibilă și scalabilă: adăugarea de funcționalitate nouă și deci implicit de noi comenzi să nu necesite modificări masive în codul deja existent și / sau recompilarea unei mari părți din kernel.

Pentru aceasta, protocolul de comunicație cu spațiul utilizator a fost realizat printr-o metodă clasică, bazată pe interfața socket BSD și apelul de sistem *ioctl()*.

Subsistemul de rețea din kernel-ul Linux poate comunica cu spațiul utilizator printr-un *socket raw din familia PF_PACKET*⁵⁶. Familia de protocoale *PF_PACKET* a fost introdusă în kernelele Linux mai noi de versiunea 2.0 și permite unei aplicații să trimită și să primească pachete direct, evitând procesarea de către stiva de protocoale (procesare TCP/IP sau IP/UDP). În concluzie, orice pachet primit prin socket va fi pasat direct către kernel și orice pachet trimis de către kernel va fi pasat direct aplicației.

Trimiterea efectivă de comenzi către kernel a fost implementată folosind apelul de sistem *ioctl*. Din perspectiva aplicației din spațiul utilizator, pentru trimiterea unei comenzi se va apela funcția *ioctl* căreia i se vor pasa trei argumente:

- un descriptor de fișier (obținut ca rezultat al apelului funcției *socket*)
- un cod de operație
- un argument ce poate fi de tip primitiv sau pointer către zona de memorie ce conține parametrii suplimentari.

Funcția din kernel de interpretare a comenzilor primite prin *ioctl*, se numește *sock_ioctl* și se găsește în fișierul *net/socket.c*. Unul dintre parametrii funcției este o

54 Funcția *sw_flood* face doar distincția între aceste două cazuri și apelează corespunzător *__sw_flood()*, care este de fapt implementarea efectivă a algoritmului.

55 Prin configurație activă se înțelege tot ceea ce ține de funcția de comutare (configurația VLAN-urilor, porturilor, adrese IP, adrese MAC statice și de multicast)

56 Socket-ul va fi creat cu un apel *socket(PF_PACKET, SOCK_RAW, 0)*.

valoare întreagă ce reprezintă codul comenzii ce trebuie tratată. Pentru a putea folosi această interfață în *LiSA* s-a pus problema introducerii de coduri de comenzi⁵⁷ specifice operațiilor de configurare necesare switch-ului. Deoarece acestea sunt relativ multe și introducerea fiecărei operații noi ar fi necesitat recompilarea întregului cod de rețea din kernel, soluția introducerii mai multor coduri de operații nu putea fi luată în considerație.

Soluția adoptată de *LiSA* pentru această problemă este mai simplă decât ar părea la prima vedere. În locul unui număr mare de operații s-a convenit să se introducă o singură operație (definită în *sockios.h* prin codul de operație *SIOCSWCFG*) și un alt hook care să apeleze o funcție din modulul switch ce se ocupă de tratarea comenzilor primite prin *ioctl*. Convenția a fost ca la un apel *ioctl* să fie transmis prin al treilea parametru un pointer la o structură (*net_switch_ioctl_arg*) ce are ca membru un întreg ce reprezintă subcomanda.

Acest mod de abordare a determinat apariția unor probleme suplimentare. Numărul comenzilor suportate fiind suficient de mare, fiecare dintre acestea necesitând argumente diferite, apărea riscul creării unei structuri trimise ca argument la *ioctl* de dimensiune prea mare.

Soluția a venit de la sine folosind un mecanism specific limbajului C: pentru părțile variabile ale structurii trimise ca parametru la *ioctl* s-a folosit construcția *union*. Structura folosită ca argument pentru comenzile *ioctl* are următoarea formă:

```
struct net_switch_ioctl_arg {
    unsigned char cmd;
    char *if_name;
    int vlan;
    union {
        ...
        /* Parametrii opționali */
        ...
    } ext;
};
```

O altă problemă legată de comunicația dintre spațiul utilizator și kernel este transmiterea parametrilor. În cazul în care ultimul parametru cu care se apelează *ioctl* este de tip primitiv, acesta se poate utiliza direct din kernel. Dacă, în schimb, acesta este pointer trebuie aplicate metode speciale pentru a îl utiliza. Este cunoscut că spațiul de adresă al unui proces nu poate fi accesat direct din spațiul kernel. În general accesarea acestuia este total descurajată, însă există cazuri în care acest lucru este necesar. În plus, pointerul respectiv este furnizat de către un program din spațiul utilizator, care poate fi incorect sau rău intenționat. În consecință, primirea precum și trimiterea de parametri de tip pointer între spațiul utilizator și kernel, trebuie făcut întotdeauna prin intermediul unor funcții speciale oferite de kernel, care pe lângă funcționalitatea de bază efectuează operații de verificare suplimentare. Dintre aceste funcții se pot menționa: *put_user*, *get_user*, *copy_to_user*, *copy_from_user*, *strncpy_from_user* etc.

La apelul acestor funcții trebuie să se verifice întotdeauna rezultatul întors de acestea. Un stil de programare sigur impune o utilizare de genul:

⁵⁷ Codurile de operații interpretate de funcția *sock_ioctl* sunt definite în fișierul *include/linux/sockios.h*

```
if (copy_from_user(buffer, ptr, size))
    return -EFAULT;
/* Aici se poate face prelucrarea buffer-ului obținut */
```

Funcția apelată din hook-ul de ioctl se numește *sw_deviceless_ioctl* și tratează toate comenzile de configurare implementate în modulul LMS. Printre comenzile implementate se pot enumera: adăugarea/eliminarea unei interfețe fizice în/din switch, adăugarea sau ștergerea unui VLAN, adăugarea sau ștergerea unei adrese de MAC statice etc.

4.2 Implementarea aplicației de configurare (CLI)

Modulul CLI a fost realizat din nevoia de a avea o interfață standard și ușor de utilizat, pentru gestionarea configurației modulului LMS. Configurarea acestuia ar fi putut fi realizată și prin intermediul unui utilitar mai simplu, rulabil din linia de comandă, însă aceasta ar fi impus restricții asupra obiectivelor inițiale ale proiectului, mai exact ar fi necesitat ca sistemul pe care se instalează *LiSA* să ruleze un demon ssh și să existe obligatoriu instalat un program de shell și toate utilitarele necesare asociate. Cum obiectivul inițial al proiectului a fost realizarea unei platforme de aplicații orientată spre sistemele dedicate, acest lucru ar fi constituit un inconvenient.

Deoarece s-a urmărit un mod standard de configurare și o interfață familiară pentru utilizator, pentru aplicația CLI a fost ales modelul Cisco IOS.

Pentru simplificarea implementării unor componente precum istoricul comenzilor, completare automată, asociere de acțiuni pe apăsarea unor taste sau combinații de taste a fost folosită biblioteca *readline*⁵⁸.

4.2.1 Organizarea comenzilor

O comandă CLI este formată dintr-o succesiune de cuvinte delimitate de un număr variabil de spații albe (caractere blank sau tab). Un cuvânt dintr-o comandă poate avea unul dintre următoarele formate:

- *format strict*. Pentru a fi admis de către analizor, cuvântul trebuie să apară în forma sa completă sau într-o formă prescurtată, cu restricția ca linia de comandă să nu fie ambiguă⁵⁹.
- *format variabil*. Cuvântul trebuie să se încadreze într-un tipar. Pentru aceasta, analizorul va folosi o funcție de validare pentru a stabili dacă îl poate accepta sau nu. Exemple de astfel de cuvinte cu format variabil sunt adresele de MAC sau adresele de IP, care pot fi validate folosind expresii regulate.

Cuvintele sunt organizate din punct de vedere logic sub forma unui arbore pe mai multe nivele. Trebuie menționat că există mai multe moduri de operare și în consecință mai

⁵⁸ <http://cnswww.cns.cwru.edu/php/chet/readline/readline.html>

⁵⁹ Ambiguitatea presupune existența mai multor posibilități de execuție pentru aceeași linie de comandă

mulți arbori de comenzi. Structura arborilor variază în funcție de nivelul de privilegiu al utilizatorului.

Implementarea practică a sistemului de comenzi a fost realizată folosind arbori în care nodurile sunt structuri de tip comandă (*sw_command_t*). Într-un nod se rețin informații despre cuvântul respectiv: un identificator ce reprezintă forma afișabilă a cuvântului, nivelul de privilegiu minim necesar pentru a accesa nodul, funcția de validare (valoare nenulă când cuvântul are formă variabilă), funcția de tratare a comenzii (valoare nenulă pentru nodurile executabile), starea nodului (codare pe biți a mai multor indicatori), informații pentru funcția de help interactiv și un pointer la următorul subarbore (nenul dacă nodul nu este terminal).

4.2.2 Analizorul de Comenzi

În acest caz s-a optat pentru scrierea unui analizor simplu, excluzându-se alternativa unui analizor sintactic generat automat cu ajutorul unor programe dedicate precum *lex*, *bison* sau *yacc*. Folosirea unor astfel de instrumente nu a fost necesară datorită particularităților situației descrise: majoritatea cuvintelor din comenzi au formă fixă iar succesiunea lor poate fi determinată foarte simplu prin parcurgerea în adâncime a arborelui de comenzi.

Practic funcționalitatea de bază a analizorului este împărțirea liniei de comandă în cuvinte și apelarea unei funcții ce primește ca parametru cuvântul curent și calculează o stare curentă. Pentru flexibilitate același analizor este folosit atât de mecanismele de completare automată și help interactiv, cât și de mecanismul de execuție. Astfel, funcția de împărțire a liniei de comandă în cuvinte primește ca parametru o funcție folosită pentru calcularea stării curente. În cazurile de completare automată sau help, analizorul primește ca parametru o funcție care completează o structură de tip *sw_completion_state_t*, pe baza cuvântului curent analizat și a stării anterioare. Similar, în cazul execuției, acesta primește ca parametru o funcție ce completează o structură de tip *sw_execution_state_t*, pe baza stării anterioare și a cuvântului analizat curent.

Funcțiile de calcul a stării curente obțin din structura determinată la pasul anterior un pointer la nodul curent din arbore și fac o parcurgere a fiilor acestuia încercând să selecteze următorul nod, dacă acesta există. Decizia de selectare a unui nod depinde de factori precum numărul de noduri compatibile cu cuvântul analizat și atributele din structurile asociate acestora.

A acțiunile executate de componentele apelante sunt determinate pe baza rezultatelor produse de analizor. De exemplu, în cazul mecanismului de execuție comenzi, dacă nu au fost întâlnite erori, se va prelua lista de argumente necesare apelului din structura corespunzătoare stării de execuție și se va apela funcția de tratare a comenzii.

5 Studiul performanțelor și testare

5.1 Descrierea Platformei de testare

Pentru măsurarea performanțelor realizate de algoritmul de comutare *LiSA* s-a

utilizat un sistem de test bazat pe o placă de bază PC Embedded din seria LE-564 dotată cu 3 porturi LAN Intel PRO/100+ și un port Intel PRO/1000 GbE Gigabit Ethernet. Pe lângă acestea, s-au mai adăugat în sistem 128 MegaBytes memorie RAM și un CompactFlash Disk Drive cu capacitate de stocare de 128 MegaBytes.

Pe acest sistem s-a instalat o distribuție de Linux minimală, conținând un kernel linux 2.6.10 compilat cu suport pentru modulul LMS și o serie de aplicații și biblioteci necesare funcționării aplicațiilor din pachetul *LiSA-CLI*, totalizând în jur de 12 MegaBytes de spațiu pe disc.

5.2 Indicatori de performanță

În scopul măsurării performanțelor sistemului de test s-au ales ca puncte de referință următorii indicatori:

- *Rata Pachetelor Comutate.* Acest indicator este obținut prin raportarea numărului de pachete pe secundă transmise de switch pe un port de ieșire la numărul de pachete pe secundă primite pe un port de intrare. Prin măsurarea acestuia pentru un set de valori de intrare experimentale s-a urmărit observarea ratei critice de comutare, ce reprezintă punctul în care sistemul înregistrează pierderi semnificative de pachete .
- *Repartiția Utilizării CPU.* Foarte important în evaluarea performanțelor sistemului este măsurarea distribuției utilizării procesorului între întreruperile hardware generate de plăcile de rețea și întreruperile software (softirq). Prin măsurarea acestui indicator, pentru aceleași date de intrare ca și în cazul *ratei de pachete comutate*, s-a urmărit compararea performanțelor obținute cu cele ale algoritmului NAPI, descris în secțiunea 2.2.6.
- *Rata de transfer.* S-a considerat necesară măsurarea performanțelor obținute în câteva cazuri de trafic intens prin switch și s-a urmărit obținerea de rezultate experimentale atât în cazul traficului unidirecțional, cât și al celui bidirecțional.

5.3 Rata de comutare a pachetelor

Pentru a putea determina experimental valorile acestui indicator a fost realizată următoarea configurație:

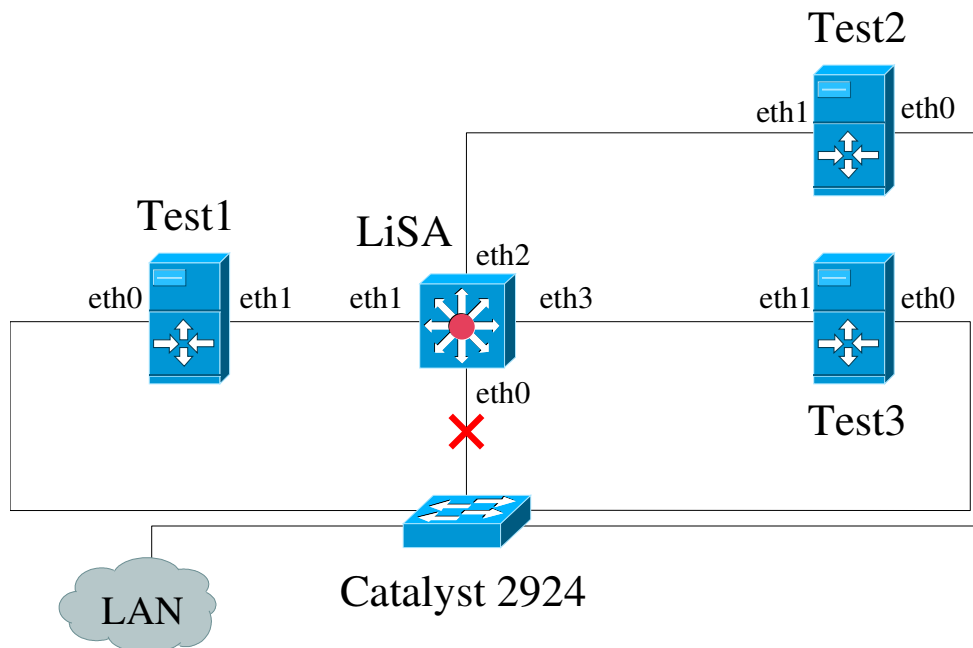


Figura 24 Configurația Experimentală

Așa cum se poate observa în Figura 24, sistemul *LiSA* a fost folosit pentru a comuta pachete între 3 servere⁶⁰ de test. Fiecare dintre cele 3 servere a fost conectat atât în switch-ul *LiSA*, cât și într-un switch Cisco Catalyst 2924, pentru a putea fi accesate de pe rețeaua locală.

În această configurație, server-ul *Test1* joacă rolul de generator de pachete, iar serverele *Test2* și *Test3* au fost folosite pentru măsurarea numărului de pachete comutate de către switch-ul *LiSA*. Practic, pe *Test1* a fost instalat un kernel 2.6.12 cu suport pentru modulul de generare de pachete *pktgen* iar pe celelalte două servere s-a folosit utilitarul *tcpdump* pentru măsurarea numărului de pachete primite. *Test1* a fost configurat să trimită 1.000.000 de pachete de dimensiune 64 bytes iar variația numărului de pachete pe secundă generate de acesta a fost obținută alegând o serie de valori pentru *delay*-ul între operațiile de trimitere a pachetelor.

Valoarea temporizării necesare (t_d) pentru obținerea unui anumit număr de PPS⁶¹ generate, poate fi calculată atunci când se cunoaște viteza de transfer a interfeței de ieșire (v_{trans}) și dimensiunea pachetului (len), astfel:



Deoarece în general valoarea celui de-al doilea termen este neglijabilă pe lângă cea a primului, se poate considera că temporizarea (sau *delay*-ul) între pachete este invers proporțională cu nr_{PPS} . Astfel, pentru generarea setului de date de intrare, s-a pornit cu o valoare inițială pentru nr_{PPS} , aceasta fiind decrementată la fiecare măsurare cu o valoare constantă.

Pe un server destinație (2 și/sau 3) se poate măsura numărul total de pachete

⁶⁰ Configurația hardware a serverelor: Dual Xeon 2.8 Ghz, 2 GigaBytes de memorie RAM, chipset Intel 6300ESB și două interfețe de rețea Broadcom Tigon 3.

⁶¹ Pachete pe secundă.

primite. Numărul de PPS de ieșire poate fi calculat astfel:



De asemenea, au fost considerate două situații relevante pentru măsurarea indicatorilor:

- *Unicast*: switch-ul are configurat static pe unul dintre porturi adresa de MAC a unuia dintre servere, iar generatorul de pachete, trimite cadre cu adresă MAC destinație egală cu a aceluia. Aici s-a urmărit observarea performanțelor pentru cazul cel mai avantajos (comutare 1 la 1).
- *Broadcast*: Generatorul de pachete trimite cadre cu adresă MAC destinație de broadcast, iar switch-ul difuzează cadrul pe celelalte două porturi. În această situație s-a urmărit observarea efectului introdus de procesarea suplimentară asociată algoritmului de difuzare (comutare 1 la N).

În continuare se vor prezenta datele obținute experimental, pe ambele situații considerate:

21000	43006	1000000	48.4	47.2	0.0000	0	43006
20000	45000	1000000	50.1	47.2	0.0000	0	45000
19000	47054	1000000	47.2	51.1	0.0000	0	47054
18000	49076	1000000	45.4	53.5	0.0000	0	49076
17000	51767	1000000	44.2	55.6	0.0000	0	51767
16000	54390	1000000	41.9	57.8	0.0000	0	54390
15000	57523	1000000	40.7	59.1	0.0000	0	57523
14000	60853	1000000	38.4	61.2	0.0000	0	60853
13000	64770	999967	36.3	63.5	0.0033	33	64768
12000	69138	999997	34.6	65.2	0.0003	3	69138
11000	74176	1000000	31.5	68.5	0.0000	0	74176
10000	79975	1000000	28.3	71.5	0.0000	0	79975
9000	86679	999975	25.3	74.5	0.0025	25	86677
8000	95723	999318	20.3	79.7	0.0682	682	95658
7000	106182	999042	13.8	86.2	0.0958	958	106080
6000	118784	955570	0.6	99.0	4.4430	44430	113506
5000	134144	826955	9.3	90.7	17.3045	173045	110931
4000	142068	766293	8.9	90.7	23.3707	233707	108866

Figura 25 Valori Experimentale în cazul de Unicast

21000	43009	1000000	12.5	87.5	0.0000	0	43009
20000	45002	999995	10.2	89.6	0.0005	5	45002
19000	47055	1000000	9.0	91.0	0.0000	0	47055
18000	49061	999987	7.0	92.7	0.0013	13	49060
17000	51781	999594	2.5	97.5	0.0406	406	51760
16000	54393	994947	0.3	99.7	0.5053	5053	54118
15000	57514	973148	0.2	99.6	2.6852	26852	55970
14000	60851	908248	3.3	96.6	9.1752	91752	55268
13000	64770	836896	4.6	95.1	16.3104	163104	54206
12000	69148	781228	3.9	95.8	21.8772	218772	54020
11000	74179	726160	3.6	96.0	27.3840	273840	53866
10000	79958	674529	3.4	96.4	32.5471	325471	53934
9000	86672	624782	3.8	95.5	37.5218	375218	54151
8000	95667	565889	3.3	96.2	43.4111	434111	54137
7000	106193	509206	3.7	95.6	49.0794	490794	54074
6000	118793	455121	3.6	96.2	54.4879	544879	54065

Figura 26 Valori Experimentale în cazul de Broadcast

Pe baza acestor date experimentale, s-au obținut următoarele grafice:

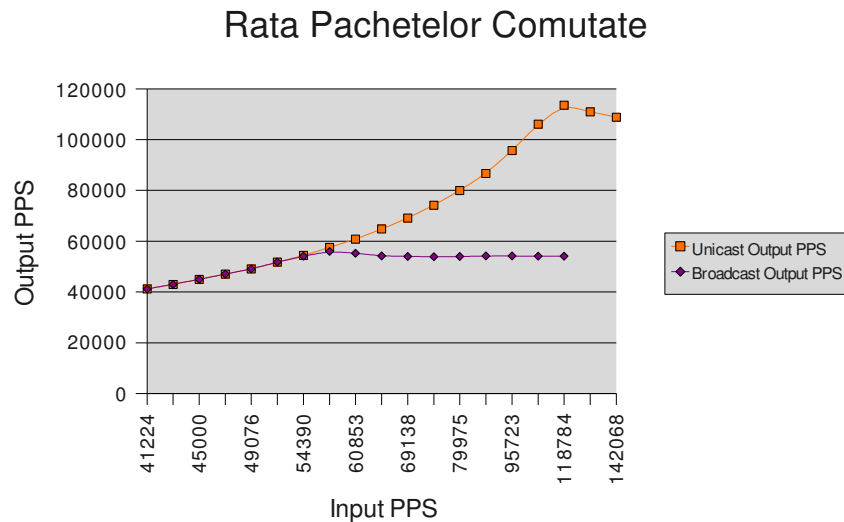


Figura 27 Rata Pachetelor Comutate

Repartiția utilizării CPU (Unicast)

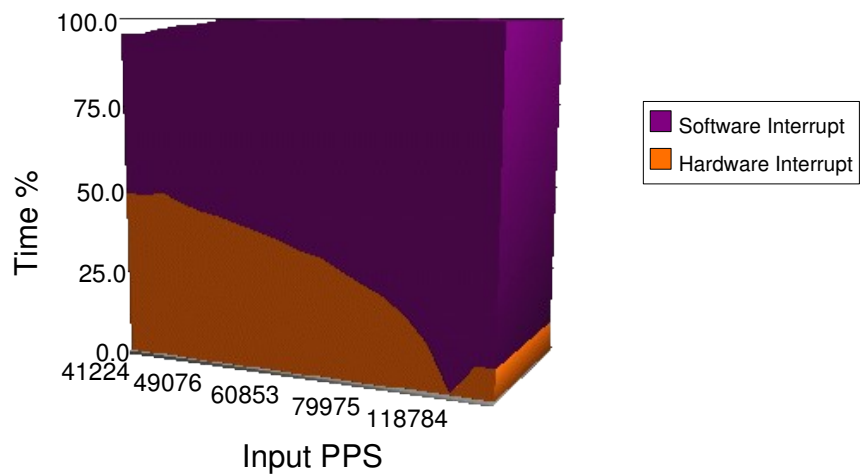


Figura 28 Repartiția Utilizării CPU în cazul de Unicast

Repartiția Utilizării CPU (Broadcast)

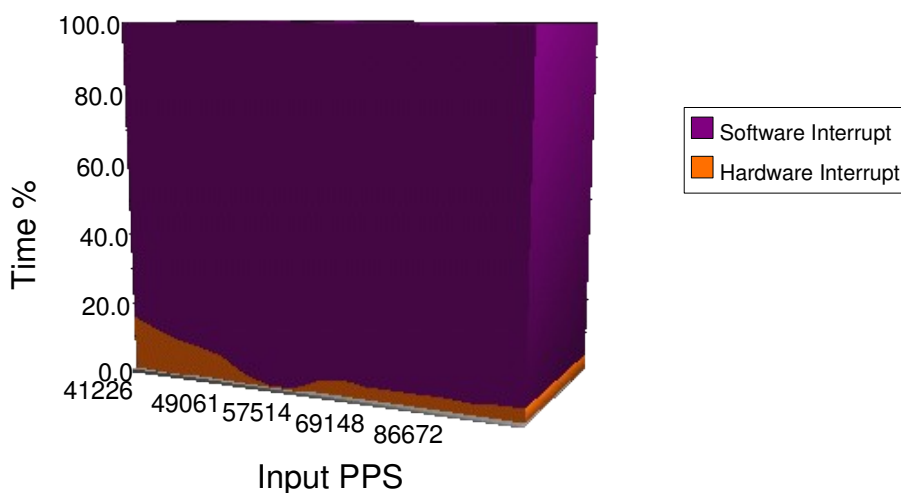


Figura 29 Repartiția Utilizării CPU in cazul de Broadcast

5.4 Rata de transfer

Pentru măsurarea ratei de transfer au fost considerate două situații:

- *transfer asimetric bidirecțional*: serverul **Test1** a fost conectat în switch pe interfața Gigabit, iar celelalte două servere și o stație de lucru pe celelalte trei interfețe. Au fost măsurate ratele de transfer de intrare și ieșire pe toate cele patru sisteme. A fost simulate transferuri simultane de la **Test1** către **Test2**, **Test3** și **workstation** și de la fiecare dintre **Test2**, **Test3** și **workstation** către **Test1**.
- *transfer asimetric unidirecțional*: folosind aceeași configurație a rețelei ca în testul anterior s-au măsurat ratele de transfer de intrare și ieșire pentru transferuri simultane de la **Test2**, **Test3** și **workstation** către **Test1**.

În ambele situații, pentru a evita ca rezultatele să fie influențate vitezele de citire / scriere corespunzătoare discurilor celor patru sisteme, s-a simulat un mediu *client-server* folosind utilitarul *nc*⁶², indirectând */dev/zero* în *stdin*-ul acestuia pentru servere și redirectând *stdout*-ul său către */dev/null* în cazul clienților.

⁶² Prescurtare de la netcat.

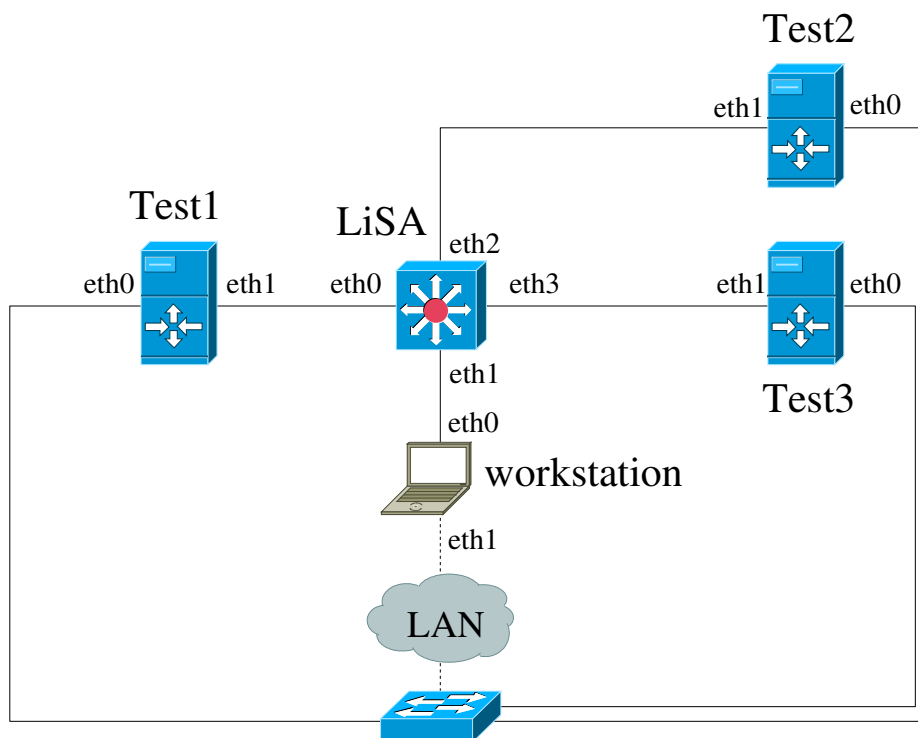


Figura 30 Configurație Rețea la măsurarea ratei de transfer

Rezultatele măsurate în cele două cazuri considerate au fost:

Mașina	Transfer Bidirecțional		Transfer Unidirecțional	
	IN Rate (kbps)	OUT Rate (kbps)	IN Rate (kbps)	OUT Rate(kbps)
Test1	148292.7	69794.5	199590	4429.7
Test2	20368.4	50664.6	1501.5	68383.1
Test3	18581.5	50727.7	1505.1	68441.4
workstation	23098.1	49293.9	1414.2	63242.6

Figura 31 Rezultate Măsurare Rate de Transfer

6 Concluzii

Analizând rezultatele obținute la măsurarea *ratei pachetelor comutate* (Figura 27), putem observa că, până la un anumit punct, numărul de PPS obținute la ieșirea switch-ului variază liniar cu numărul de PPS de la intrare, atât în cazul comutării 1 la 1 (*unicast*), cât și în cazul comutării 1 la N (*broadcast*).

De la o anumită valoare a numărului de PPS, în ambele cazuri se poate observa apariția pierderilor de pachete la ieșire. Aceasta nu este un lucru anormal, deoarece datorită numărului foarte mare de pachete sosite pe secundă, sistemul nu reușește să le proceseze la timp. Din acest punct devine evidentă și intervenția algoritmului NAPI . Sistemul procesează pachetele primite în măsura în care reușește să facă acest lucru. Se poate observa că după atingerea maximului (*rata critică de comutare*), valoarea ratei de comutare începe să se stabilizeze în jurul unei constante, formând o treaptă. Aceasta este o

consecință a scalabilității algoritmului NAPI.

Se poate observa că în cazul de *unicast*, overhead-ul introdus de comutare este aproape inexistent, *rata critică de comutare* (atinsă pentru un număr de aproximativ 110.000 de PPS la intrare) fiind stabilită de puterea de procesare a sistemului.

Performanțele obținute în cazul de *broadcast* se deteriorează față de primul caz, valoarea *rății critice de comutare* fiind atinsă mai devreme. Scăderea performanțelor este normală, deoarece sistemul trebuie să facă procesări în plus pentru fiecare pachet primit: se însumează latențele introduse de clonările de socket buffere și procesările cozilor de ieșire de pe interfețe.

Se observă din repartitia utilizării procesorului (Figura 28 și Figura 29) că în cazul testului de broadcast, algoritmul NAPI intervine mai devreme cauzând o reducere a numărului de întreruperi hardware și o creștere a timpului petrecut de CPU în întreruperi software (*softirq*).

În cazul testului de măsurare a ratei de transfer, se poate observa că atât în cazul de transfer bidirecțional cât și în celălalt caz, se obține aproximativ același *throughput*. Însușind ratele de transfer de intrare și de ieșire obținem o valoare de aproximativ 200 Mbps.

O explicație a acestei limitări, observabile în ambele cazuri, este că lărgimea totală de bandă este limitată de lărgimea de bandă a magistralei PCI. Practic, dacă se ia în considerare faptul că fiecare pachet este transferat de două ori prin magistrală, se poate estima o valoare a lărgimii de bandă pentru aceasta, de aproximativ 400 Mbps.

Performanțele realizate de switch la toate testele sunt destul de bune. Trebuie avut în vedere că în teste s-au considerat cazuri extreme de utilizare. Dimensiunile foarte mici pentru pachete trimise, precum și numărul mare de PPS cu care s-a testat sistemul, demonstrează o comportare apreciabil de bună a sa, chiar în sisteme de producție.

Pe lângă testele efectuate, sistemul pe care s-a instalat *LiSA*, a fost folosit într-o rețea reală, funcționând timp de două săptămâni fără a se înregistra vreă problemă și fără ca acesta să necesite reinițializarea sistemului de operare. Trebuie menționat că în toată această perioadă, sistemul a rulat un kernel compilat cu simboluri de debugging, în vederea depanării oricărei probleme ce ar fi putut apărea.

Testele de performanță, precum și testarea utilizării într-un mediu real, dovedesc obținerea unei platforme stabile și scalabile, aceasta având avantajul major, spre deosebire de alte produse de pe piață, de a putea fi realizată cu un minim de costuri de producție. Deși soluția nu este proiectată pentru a fi implementată în rețele de dimensiuni mari, aceasta poate fi utilizată cu succes în rețele mici și medii.

Desigur, mai există o mulțime de facilități încă neimplementate în acest proiect, printre care, una dintre cele mai notabile este suportul pentru algoritmul Spanning Tree Protocol⁶³. Mai există și alte idei ce urmează a fi implementate, printre care facilități avansate de rutare și filtrare de pachete, suport pentru transferul bazei de date de VLAN-uri prin protocolul VTP ș.a.

Cu toate că numărul de îmbunătățiri ce pot fi aduse proiectului este destul de mare, avantajul major al platformei open-source pe care a fost dezvoltat inițial acesta, poate permite oricui să contribuie nu numai cu cod sursă, dar și cu idei noi. Acesta este deja înregistrat pe site-ul de proiecte open-source <http://freshmeat.net>, iar codul sursă și documentația pot fi descărcate de pe pagina de web dedicată proiectului: <http://lisa.ines.ro>.

63 Prescurtat STP. Scopul său principal este acela de evitare a buclelor de comutare atunci când există legături redundante între mai multe switch-uri.

7 Anexă: Primitive și algoritmi de sincronizare în Linux Kernel

7.1 Zăvorâre și Operații atomice

În kernel-ul Linux se pot executa simultan mai multe activități ce se pot întrerupe una pe cealaltă. În cazul sistemelor multiprocesor, mai multe activități se pot desfășura în paralel. De aceea, este extrem de important pentru stabilitatea sistemului ca aceste operații să poată rula în paralel fără a produce efecte nedorite.

Atât timp cât activitățile din kernel operează independent nu poate apărea nici o problemă, însă atunci când mai multe activități accesează structuri de date partajate, pot apărea efecte nedorite, chiar și în sistemele uni-procesor.

Ca exemplu pentru o astfel de problemă, în cele două figuri de mai jos se pot observa efectele nedorite create de două operații A și B care încearcă să adauge un element într-o coadă. După prima instrucțiune executată de A, aceasta este întreruptă de execuția operației B:

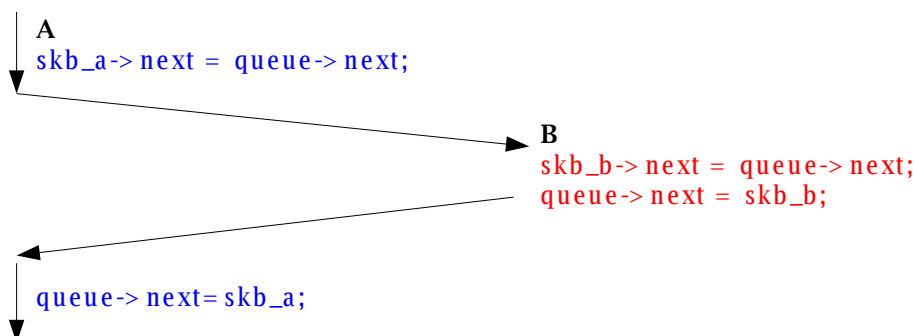


Figura 32 B întrerupe pe A în regiunea critică

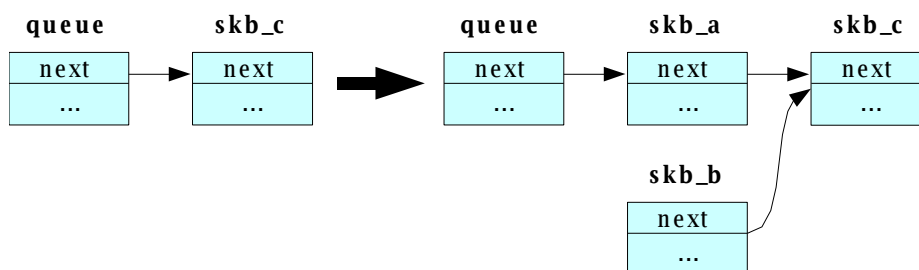


Figura 33 Rezultatul operațiilor neprotejate A și B

Pentru a evita astfel de rezultate nedorite, aceste operații trebuie să fie *atomice*. Aceasta înseamnă că o operație compusă din mai mulți pași trebuie executată ca o operație

indivizibilă. Nici o altă instanță nu poate modifica structura de date concurrent cu o operație atomică.

În linux există mai multe mecanisme de execuție atomică a operațiilor. Acestea diferă după modul în care se așteaptă intrarea într-o regiune critică potențial ocupată. În continuare vor fi prezentate sumar câteva dintre aceste mecanisme.

7.1.1 Operații pe biți

Operațiile atomice pe biți constituie punctul de pornire în implementarea altor mecanisme de zăvorâre precum *spinlock-uri* și *semafoare*. Protecția regiunilor critice este realizată cu zăvoare, care uzual sunt implementate prin variabile ce rețin starea curentă a zăvorului (memorează câte activități sunt în regiunea critică).

Aceasta înseamnă că, înainte de intrarea în regiune critică, trebuie mai întâi citită starea variabilei zăvor și apoi modificată corespunzător. Într-un procesor aceasta se realizează prin două comenzi ce trebuie executate una după cealaltă. Oricum, poate apărea o situație precum cea descrisă în figura 33: prima operație este întreruptă de o a doua care va schimba starea variabilei. Din acest motiv, pentru a putea suporta regiunile critice, apare necesitatea operațiilor atomice de setare și testare, implementate ca o singură instrucțiune mașină.

În linux există atât operații de setare și testare la nivel de bit, cât și operații de setare și testare la nivelul unei valori întregi (Ex: `set_bit`, `clear_bit`, `atomic_set`, `atomic_read`, `atomic_inc`, `atomic_dec` etc.).

7.1.2 Spinlock

Spinlock-urile mai sunt cunoscute și sub denumirea de *busy wait locks*, datorită modului în care funcționează. Când începe o regiune critică și zăvorul (în acest caz spinlock) a fost deja luat, procesorul așteaptă activ până când acesta este eliberat. Aceasta înseamnă că procesorul testează în buclă variabila de zăvorâre până când aceasta este eliberată de activitatea care a obținut inițial zăvorul.

Deși la prima vedere aceasta irosește timpul de procesor "fără sens", se poate dovedi mai eficient să se testeze în continuu valoarea variabilei zăvor pentru o perioadă foarte scurtă, intrând apoi în regiunea critică, decât să se cedeze controlul planificatorului și să se acorde timpul de calcul altei activități, deoarece în ultimul caz există pericolul scurgerii unei perioade de timp mult mai mari între momentul în care zăvorul este eliberat și momentul în care activitatea ce așteaptă reușește să intre în regiunea critică. În plus, comutarea controlului către altă activitate, cauzată de apelarea planificatorului, poate dura mai mult decât o simplă așteptare în ciclul de *busy wait*. Din aceste motive, se poate deduce că *spinlock-ul* este cea mai bună modalitate de zăvorâre pentru regiuni critice de dimensiuni mici (timp de zăvorâre foarte reduși).

În kernel-ul linux, spinlock-urile sunt implementate prin variabile de tip *spinlock_t*, definite în fișierul *include/linux/spinlock.h*.

7.1.3 RW Lock

Spinlock-urile reprezintă mecanisme simple și utile pentru protejarea operațiilor paralele pe structuri partajate. Cu toate acestea, ele prezintă dezavantajul încetinirii activităților, deoarece acestea trebuie să aștepte *activ* pentru eliberarea zăvoarelor. În anumite situații *busy-waiting*-ul nu este necesar. De exemplu, există structuri care sunt citite frecvent, dar care sunt modificate foarte rar. Un exemplu clasic din linux îl constituie lista *net_device*-urilor înregistrate, *dev_base*. Aceasta este foarte rar modificată, în schimb constituie subiectul multor accese de citire.

La accesarea unei astfel de structuri, este absolut necesară folosirea unui spinlock, însă activitățile de citire nu trebuie întârziate atunci când nu există nici o activitate de scriere asupra acesteia. Pentru rezolvarea acestei probleme au fost proiectate *spinlock-urile read-write* sau *RW lock-urile*. Acestea permit mai multor activități de citire să acceseze simultan regiunea critică, atât timp cât nu există nici o operație de scriere. O dată ce o operație de scriere obține zăvorul, nici o operație de citire nu trebuie să se găsească în regiunea critică sau să intre în aceasta, până când zăvorul nu este eliberat.

În linux, *read-write spinlock-urile* sunt implementate prin structura *rwlock_t*.

7.1.4 Semafoare

Există și un mod de a evita așteptarea activă până când o regiune critică poate fi accesată. În loc de a aștepta eliberarea acesteia, activitatea cedează procesorul apelând planificatorul. Aceasta înseamnă că timpul de procesor poate fi folosit în beneficiul altor activități. Acest concept este cunoscut sub numele de *semafor* sau *mutex*.

Kernel-ul linux oferă suport pentru *semafoare*. Cu toate acestea, ele sunt foarte rar folosite în codul de networking.

7.2 Algoritmul Read-Copy-Update (RCU)

Ideea de bază din spatele algoritmului RCU (read-copy-update) este separarea operațiilor distructive în două părți, prima dintre ele având rolul de a împiedica observarea distrugerii unui element din alte activități, iar a doua realizând efectiv distrugerea elementului. Există o anumită perioadă, numită și perioadă de grație, care trebuie să expire între execuția celor două părți, perioadă ce trebuie să fie suficient de mare astfel încât toate activitățile care accesau pentru citire elementul ce trebuie șters să nu mai dețină nici o referință la acesta. De exemplu, o ștergere dintr-o listă protejată prin RCU va extrage mai întâi elementul din listă, va aștepta să expire perioada de grație, după care va elibera memoria rezervată acestuia.

Avantajul abordării RCU este că cititorii RCU nu trebuie să obțină nici un zăvor, nu trebuie să execute operații atomice, să scrie într-un segment de memorie partajată sau să execute bariere de memorie⁶⁴. Faptul că aceste operații sunt destul de costisitoare chiar și pe procesoarele moderne este ceea ce determină performanța superioară a algoritmilor RCU în cazurile în care se efectuează citiri intensive. Faptul că cititorii RCU nu sunt obligați să obțină zăvoare poate simplifica în mare măsură codul de evitare a *dead-lock*-urilor.

Întrebarea evidentă este: cum poate un writer să își dea seama când a expirat perioada de grație dacă nici un reader nu semnalează în nici un fel atunci când termină?

La fel ca și în cazul *spinlock*-urilor, cititorii RCU nu au voie să se blocheze, să comute la execuția mod utilizator sau să intre în ciclul *idle*. Astfel, de îndată ce se observă că procesorul trece prin oricare dintre aceste trei stări, se știe că acesta a ieșit din orice regiune critică de citire. Astfel, dacă se elimină un element dintr-o listă înlănțuită și se așteaptă până când toate procesoarele au schimbat contextul, au trecut prin execuția în mod utilizator sau au trecut prin ciclul *idle*, elementul poate fi dealocat fără probleme.

7.2.1 Manipularea listelor înlănțuite folosind RCU

Una dintre cele mai bune aplicații pentru algoritmul RCU este protejarea listelor înlănțuite (implementate cu structuri *list_head*) asupra cărora se efectuează predominant operații de citire. Unul dintre cele mai mari avantaje ale acestei abordări este acela că toate barierele de memorie necesare sunt incluse în macro-urile de lucru cu liste definite în *list.h*.

Practic, pentru a executa un ciclu de citire pe o listă înlănțuită protejată cu RCU, trebuie parcurși următorii pași:

- înainte de începerea iterației trebuie să se apeleze macro-ul *rcu_read_lock*. Acesta apelează *preempt_disable* care incrementează contorul de preemptivitate al firului de execuție curent, prevenind întreruperea acestuia.
- parcurgerea listei trebuie făcută cu primitivele de parcurgere cu sufix *_rcu()*, care asigură barierele de memorie necesare pe cazul de citire în cazul procesoarelor DEC

⁶⁴ Valabil pe orice arhitectură, cu excepția procesoarelor Alpha.

Alpha.

- La sfârșitul iterației trebuie apelat macro-ul *rcu_read_unlock*, care apelează *preempt_enable*, decrementând contorul de preemptivitate al firului de execuție curent.

În cazul operațiilor de scriere, abordarea e un pic diferită. Pentru exemplificare se va considera cazul ștergerii unei mulțimi de elemente dintr-o listă înlănțuită protejată cu RCU. Pentru aceasta este nevoie de o listă suplimentară în care vor fi adăugate elementele ce trebuie șterse. Algoritmul de ștergere va fi realizat în doi pași:

- parcurgerea listei la fel ca în cazul de citire, iar la întâlnirea unui element ce trebuie șters acesta va fi scos din listă (se va apela *list_del_rcu*) și va fi adăugat în lista adițională.
- la terminarea parcurgerii listei, toate elementele ce trebuie șterse se găsesc în lista adițională însă memoria asociată acestora nu a fost încă eliberată. Pentru a se asigura consistența, trebuie ca toți cititorii să "vadă" modificările din lista inițială. Pentru aceasta trebuie apelată funcția *synchronize_kernel*, care asigură așteptarea perioadei de grație până când se termină toate regiunile critice de citire RCU. După aceasta, parcurgând lista adițională, se poate face eliberarea elementelor din aceasta fără nici un risc. Este eliminat orice risc, deoarece atunci când se apelează *synchronize_kernel* pe unul dintre procesoare, în timp ce alte procesoare se găsesc în regiune critică de citire RCU, este garantat că apelul *synchronize_kernel* se blochează până când toate celelalte procesoare ies din regiunea critică.

Mai există o metodă de realizare a operațiilor de scriere într-o listă protejată cu RCU, ce diferă de prima metodă prezentată anterior prin faptul că nu folosește o listă suplimentară. Aceasta constă în apelarea funcției *call_rcu* pentru fiecare element ce trebuie modificat. Funcția *call_rcu* primește ca parametri un pointer la o structură *rcu_head* și un pointer la o funcție care execută efectiv operația de scriere. Practic, *call_rcu* planifică execuția funcției primite ca parametru după expirarea perioadei de grație, ideea fiind asemănătoare cu cea din cazul *synchronize_kernel*. La apel, funcția va fi apelată cu parametrul de tip *rcu_head **. În practică, structurile ce constituie elemente ale listei înlănțuite vor avea un membru de tip *rcu_head*, astfel că la apelul lui *call_rcu* se va pasa un pointer la acesta. Din funcția programată pentru execuție după expirarea perioadei de grație se va putea obține înapoi o referință la structura inițială (elementul ce trebuie șters) prin folosirea macro-ului *container_of*.

Bibliografie

- Klaus Wehrle, Frank Pahlke, Hartmut Ritter, Daniel Muller, Marc Bechler - **The Linux Network Architecture**, Editura Prentice Hall, 2005.
- Jonathan Corbet, Alessandro Rubini, Greg Kroah-Hartman - **Linux Device Drivers, 3rd edition**, Editura O'Reilly and Associates, 2005.
- Charles E. Spurgeon - **Ethernet: The Definitive Guide**, Editura O'Reilly and Associates, 2000.
- Octavian Purdilă - **Curs de Sisteme de Operare**.

- *Alan Cox - Network Buffers and Memory Management*, Linux Journal ediția 29, Septembrie 1996.
- *Alexey Kuznetsov, Robert Olsson, Jamal Hadi Salim - Beyond Softnet*, Usenix Paper, 2004.
- *Rich Seifert - Issues in LAN Switching and Migration from a Shared LAN Environment*, Networks and Communications Consulting, Technical Report, November 1995.
- *Vincent Guffens - Path of a packet in the Linux Kernel*, 2003.
- *Paul E. McKenney - Exploiting Deferred Destruction: An Analysis of Read-Copy-Update Techniques*, Carnegie Mellon University, 2004.
- *Cisco Systems - The Cisco Certified Network Associate Curriculum*.