

Universitatea Politehnică București
Facultatea de Automatică și Calculatoare

Sistem de rutare între VLAN-uri bazat pe LISA

Autor:

Radu Rendec

Coordonatori științifici:

Prof. dr. ing. Nicolae Țăpuș

Drd. ing. Octavian Purdilă

Iunie 2005

Cuprins

1. Introducere.....	3
2. Noțiuni teoretice.....	6
2.1. VLAN-uri.....	6
2.2. Porturi în trunchi.....	9
2.3. Comutare de pachete la nivel 3.....	12
2.4. Fluxul pachetelor în Linux. Socket Buffers.....	15
2.5. Recepția cadrelor în Linux. Softnet și NAPI.....	21
2.6. Trimiterea cadrelor în Linux.....	25
3. Arhitectură.....	27
4. Implementare.....	31
4.1. Modurile de acces al porturilor la VLAN-uri.....	32
4.2. Baza de date de VLAN-uri. Asignarea porturilor la VLAN-uri.....	33
4.3. Interfețele virtuale.....	34
4.4. Încapsularea 802.1q.....	37
4.5. Algoritmul de comutare la nivel 2.....	39
4.6. Optimizări pentru broadcast și multicast.....	42
4.7. Sincronizări.....	46
Sincronizarea VDB.....	46
Sincronizarea FDB.....	47
5. Performanțe și testare.....	51
6. Concluzii.....	52
7. Anexe.....	53
Anexa A □ Sincronizarea RCU.....	53
Anexa B □ Comutarea fără-copiere.....	57
Anexa C □ Algoritmul STP.....	59
Anexa D □ Porturi în trunchi și rutare între VLAN-uri cu Linux bridge și 8021q.....	61
Anexa E □ Echivalențe de termeni.....	62
8. Bibliografie.....	64

1. Introducere

O dată cu evoluția tehnicii de calcul și creșterea performanțelor stațiilor de lucru, cerințele utilizatorilor au devenit din ce în ce mai pretențioase. Aplicațiile software s-au dezvoltat pe măsură, ajungând să folosească resurse distribuite pentru a putea asigura accesul în timp real la un volum foarte mare de date. Au crescut de asemenea și necesitățile de comunicare, astfel încât conferințele video și prezentările multimedia bazate pe transport prin rețele de date sunt astăzi ceva obișnuit.

În aceste condiții, performanțele rețelelor de date trebuie să țină pasul și să asigure lărgimea de bandă și viteza de propagare necesare aplicațiilor software.

Numele proiectului vine de la *Linux Switch Appliance*¹. LISA își propune să ofere o soluție ieftină și eficientă pentru componentele cheie ale unei rețele de mici dimensiuni. Comutarea de pachete la nivelurile 2 și 3 ale modelului OSI² este un proces esențial pentru asigurarea performanțelor unei rețele de date de arie locală sau chiar metropolitană cu arhitectură Ethernet. LISA implementează comutarea de pachete atât la nivel 2 cât și la nivel 3, oferind totodată facilități de configurare, monitorizare și control ale procesului de comutare de pachete.

Proiectul este în întregime *open-source*, având în centru sistemul de operare Linux. Comutarea pachetelor este realizată în întregime în software. Poate pare inefficient (se știe că implementările hardware sunt în general mai eficiente), dar voi aduce argumente pertinente în favoarea acestei alegeri. Ținta principală nu vizează lărgimea de bandă și o latență scăzută (principalele avantaje ale unei implementări hardware), ci facilități foarte avansate (comparabile cu ale produselor de vârf de pe piață) la un preț foarte scăzut.

Fiind bazat pe kernelul Linux și folosind componenta de rețea a acestuia, LISA poate rula pe orice platformă suportată de Linux și poate folosi orice chip-uri de rețea pentru care există drivere în Linux, *fără a fi necesară adaptarea codului*.

Nu este nevoie de un hardware specializat. LISA a fost gândit în primul rând pentru a rula pe arhitecturi PC/x86. Folosind LISA, un simplu computer cu mai multe plăci de rețea poate fi transformat într-un comutator multistrat³ cu facilități foarte avansate, cum ar fi interogarea tabelii de comutare⁴, asignarea de adrese fizice⁵ statice ș.a. Un sistem PC bazat pe un microprocesor K6-2/300 MHz cu 6 chip-uri Realtek 8139 poate astfel oferi

1 O traducere aproximativă în limba română ar fi "Echipament dedicat de comutare folosind Linux".

2 OSI = Open Systems Interconnection

3 *Multilayer Switch* în limba engleză.

4 *Switching Table* în literatura de specialitate în limba engleză; acest concept este de multe ori folosit și sub numele de *Forwarding Database*.

5 Este vorba de adresele folosite la nivelul 2 al ierarhiei propuse de OSI; acestea mai pot fi numite *adrese de nivel 2*, sau, în limba engleză, *Layer 2 Addresses*. De cele mai multe ori însă, în limba engleză conceptul este denumit *MAC Address* (de la Medium Access Control), sau, pe scurt, doar *MAC* = variantă pe care o voi folosi și eu în restul materialului pentru simplitate.

performanțe satisfăcătoare pentru o rețea de dimensiuni mici. Este de notat că prețul unui astfel de sistem este comparabil cu al celor mai ieftine switch-uri⁶ configurabile⁷ de pe piață. Acestea din urmă însă au facilități net inferioare celor oferite de LISA și nu realizează comutare la nivel 3.

Acolo unde este nevoie de performanțe ridicate se poate folosi hardware mai pretențios, cum ar fi un microprocesor mai puternic, chipuri de rețea mai performante (de viteză mai mare sau care facilitează comutarea *zero-copy*⁸ a pachetelor). De asemenea, se pot folosi memorie dublu canal și/sau mai multe magistrale PCI pentru a micșora timpul de transfer al datelor.

Totuși, atunci când am inițiat proiectul, am avut în minte chiar realizarea unui produs compact, un sistem *single-board*⁹ bazat pe o arhitectură PC, dar din care să fie eliminat tot ceea ce nu este necesar pentru un switch (interfețe USB, audio, pentru dischetă, chip grafic etc.). Acest sistem ar putea avea chiar și un BIOS specializat – nu sunt necesare facilitățile unui sistem PC standard; în schimb, spațiul de pe memoria flash ar putea fi folosit pentru a implementa un mini sistem de operare care să permită recuperarea parolei de acces sau înlocuirea imaginii de kernel în cazul în care aceasta este distrusă prin eșuarea unui proces de actualizare.

Dezvoltarea unui astfel de sistem este destul de anevoioasă, deoarece implică utilizarea unor cunoștințe extrem de avansate de hardware, experiență în proiectarea sistemelor și documentație oferită de către producătorii de chip-uri doar pe baza unor acorduri speciale de confidențialitate.

În vederea unei utilizări cât mai ușoară pe un sistem dedicat, am creat o mini distribuție de Linux, care include toate componentele necesare. Distribuția de numai câțiva megabytes suportă momentan doar pornirea de pe un hard disk (sau o memorie statică astfel interfațată încât să emuleze un hard disk), însă ar putea fi modificată cu ușurință ca să suporte pornirea de pe un CD sau de pe o memorie de tip USB Stick.

Am instalat și folosit fără probleme distribuția astfel creată pe un sistem PC embedded de tip LE-564 produs de firma Commell Systems¹⁰. Sistemul conține o interfață Gigabit și trei interfețe Fast Ethernet produse de firma Intel și a fost echipat suplimentar cu o memorie dinamică SDRAM de 128 MB și o memorie non-volatilă de tip Compact Flash.

6 Termenul echivalent din limba română este *comutator*, însă, bazându-mă pe propria observație, pot afirma că termenul englezesc este mult mai des utilizat; prin urmare îmi voi permite să-l utilizez și eu pe parcursul acestui material.

7 În limba engleză *management switch* – un switch care poate fi administrat și la care procesul de comutare a pachetelor poate fi controlat prin configurare.

8 *Fără-copiere* în limba română – se referă la comutarea pachetelor fără a realiza vreo copie a conținutului lor între adrese diferite ale memoriei centrale; pentru mai multe detalii, vezi Anexa B.

9 Sistem de calcul în care toate componentele sunt montate pe aceeași placă.

10 Mai multe detalii despre acest sistem se pot obține de pe site-ul firmei producătoare, la adresa <http://www.commell-sys.com/Product/SBC/LE-564.htm>.

2. Noțiuni teoretice

Voi porni de la organizarea clasică a rețelelor bazate pe Ethernet, pentru a evidenția principalele ei dezavantaje. Voi arăta apoi ce îmbunătățiri au fost făcute pentru a elimina aceste dezavantaje, ajungând în cele din urmă la organizarea modernă a rețelelor.

Pe parcursul prezentării, voi introduce câteva concepte, arătând totodată cum au devenit necesare în mod natural.

2.1. VLAN-uri

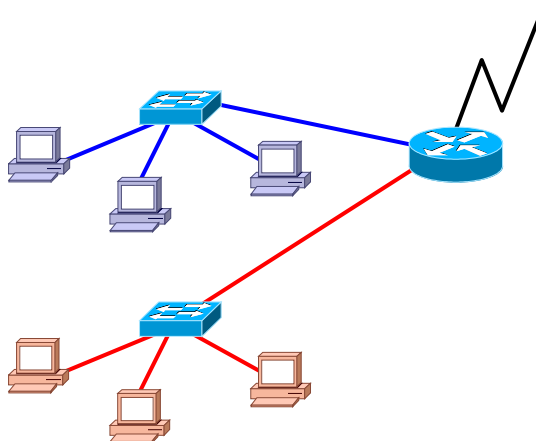


Figura 1 □ Organizarea clasică a unei rețele

Figura 1 ilustrează organizarea clasică a unei rețele de stații de calculatoare. Se pot observa două subrețele, figurate cu roșu și cu albastru. Fiecare subrețea conține un switch și trei stații de lucru. Cele două subrețele sunt interconectate prin intermediul unui router, care asigură politici de securitate pentru intercomunicare, precum și o legătură la internet.

Extinzând conceptul la nivelul unei mici firme, am putea avea mai multe subrețele, fiecare corespunzând unui anumit departament. Este evident că accesul la anumite resurse critice ale unui departament (baze de date, servere de stocare cu date confidențiale etc.) trebuie limitat pentru celelalte departamente. În aceste condiții, apartenența unei stații de lucru la o anumită subrețea devine un element cheie pentru securitate.

În rețelele mai mari cablarea se realizează de la început pentru toată clădirea. Astfel, mutarea unei stații într-o altă subrețea devine destul de complicată. În cazul cel mai fericit, cele două switch-uri se află în aceeași cameră de cablare¹¹ și problema poate fi rezolvată prin simpla mutare a cablului terminal¹² în switch-ul adecvat. Dar de cele mai multe ori switch-urile se află în camere de cablare diferite și atunci problema este foarte greu de

¹¹ În limba engleză *wiring room* sau *wiring closet*.

¹² În limba engleză *patch chord*.

rezolvat. Singura opțiune rămâne folosirea unor cabluri de intercomunicație între cele două camere de cablare, presupunând că acestea există. Dacă există, oricum sunt în număr limitat, deci soluția nu este scalabilă.

Astfel a apărut aproape natural ideea de rețea virtuală sau, pe scurt, VLAN¹³. Revenind la exemplul de mai sus, problema mutării unei stații într-o altă subrețea ar putea fi rezolvată foarte elegant dacă porturile unui switch ar putea fi cumva grupate, astfel încât pachetele să fie comutate doar în interiorul grupului. Un astfel de grup reprezintă ceea ce se numește VLAN, iar acum un switch fizic poate fi privit ca fiind alcătuit din mai multe switch-uri: fiecare dintre aceste switch-uri reprezintă un grup, adică un VLAN.

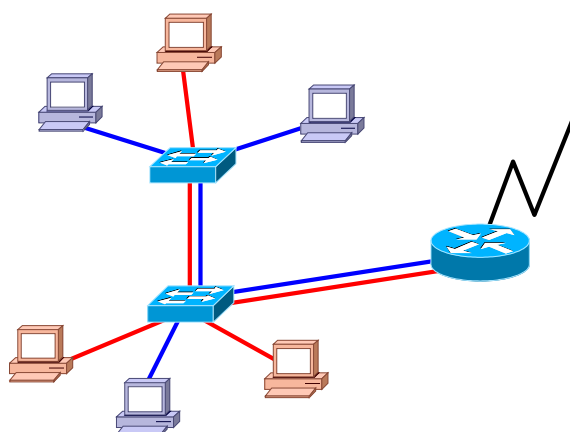


Figura 2 □ Organizarea unei rețele cu VLAN-uri

Figura 2 reprezintă aceeași rețea pe care am folosit-o mai devreme ca exemplu, dar acum reorganizată astfel încât să poată profita de avantajele aduse de VLAN-uri. Fiecărei subrețele îi corespunde acum un anumit VLAN și se observă că o stație poate fi mutată cu ușurință în altă subrețea prin simpla mutare a cablului într-un port din alt VLAN.

Dar dacă tot trebuie mutate cabluri, nu am câștigat foarte mult: în loc de VLAN-uri ar fi putut fi folosite mai multe switch-uri fizice și rezultatele ar fi fost aceleași.


Ideea de VLAN devine extrem de atractivă abia în contextul switch-urilor configurabile. În loc să mutăm un cablu, putem *configura* portul respectiv din switch să aparțină unui alt VLAN. Firește că în realitate primele switch-uri care au suportat VLAN-uri au fost și configurabile. Eu am separat aici cele două concepte doar pentru a pune și mai mult în evidență importanța facilității de configurare.

Înainte de a mai înainta în prezentarea evoluției rețelelor locale, se impun câteva observații cu privire la switch-urile cu suport de VLAN-uri. Prima observație (lucru destul de ușor de intuit de altfel) este că traficul de tip broadcast are loc doar în interiorul VLAN-urilor. Cu alte cuvinte, acum broadcast nu mai înseamnă "trimite către toate porturile", ci "trimite către toate porturile care fac parte din VLAN-ul pe care a venit pachetul".

Tabela de comutare și tot ceea ce este legat de ea suferă de asemenea modificări în cazul VLAN-urilor.

13 Prescurtarea provine din limba engleză și a apărut în două etape: *Local Area Network* (Rețea de arie locală) a fost prescurtat *LAN*; prescurtarea a fost atât de intensiv folosită încât aproape a căpătat valoare de substantiv. Astfel termenul de *rețea virtuală* a fost abreviat ca *VLAN*, de la *Virtual Lan*.

MAC	Port
000e.4c21.3b28	1
000e.6f21.d543	1
000e.835e.0a12	2
000e.d478.fe22	3



MAC	VLAN	Port
000e.4c21.3b28	4	1
000e.6f21.d543	5	1
000e.835e.0a12	4	2
000e.d478.fe22	5	3

Figura 3 □ Tabelă de comutare fără și cu VLAN

La un switch clasic tabela de comutare este construită memorând adresa MAC *sursă* a pachetelor împreună cu portul prin care au intrat. Atunci când trebuie transmis (comutat) un pachet se caută adresa MAC *destinație* în tabela de comutare. Dacă este găsită, pachetul este trimis pe portul asociat adresei; în caz contrar pachetul este trimis către toate porturile.

Aceași adresă MAC nu poate fi asociată cu două porturi diferite. Într-adevăr, având în vedere că adresele din tabela de comutare sunt "învățate" la recepția pachetelor (considerând portul prin care au fost primite), nu are sens ca pachetele emise de aceeași placă de rețea a unei stații să ajungă în switch prin două porturi diferite¹⁴. În condițiile în care se primește un pachet și adresa lui sursă există deja în tabela de comutare, înregistrarea este *înlocuită* (de fapt se actualizează câmpul *port* al înregistrării).

La un switch care suportă VLAN-uri, așa cum am arătat, este esențial ca pachetele să nu fie comutate în afara VLAN-ului. Prin urmare, asocierea adreselor MAC cu porturile nu mai este suficientă. În plus, trebuie memorat VLAN-ul pe care a venit pachetul. Acum condiția de identificare a unei înregistrări corespunzătoare în tabela de comutare este să coincidă adresa MAC destinație și VLAN-ul.

Considerând exemplul din Figura 3, un pachet cu adresa MAC destinație 000e.d478.fe22 care sosește printr-un port configurat în VLAN-ul 4 va fi trimis pe toate porturile din VLAN-ul 4, deși în tabela de comutare există o înregistrare cu adresa MAC respectivă. Se întâmplă așa pentru că înregistrarea de care vorbeam (a 4-a din ilustrație) are asociat VLAN-ul 5. Pentru pachetul din VLAN-ul 4 este ca și când înregistrarea nu ar exista.

În condițiile asocierii adreselor MAC cu un port și un VLAN, tabela de comutare poate conține aceeași adresă MAC asociată cu VLAN-uri diferite. Acesta nu reprezintă un caz particular, ba chiar poate fi întâlnit destul de des în practică. Explicația fenomenului va fi expusă pe larg în secțiunea următoare.

2.2. Porturi în trunchi

Rețeaua din Figura 2 are un dezavantaj major: atât între cele două switch-uri cât și între al doilea switch și router există două legături fizice, câte una pentru fiecare VLAN. Aceasta

¹⁴ În realitate acest lucru este posibil, dar în două cazuri speciale. Primul caz îl constituie o așa-numită buclă de comutare (topologia rețelei nu mai este arbore, ci are un ciclu) și în acest caz *aceiași* pachet poate sosi pe două porturi diferite. Această problemă este tratată de către algoritmul *STP*, descris pe larg în Anexa C. Al doilea caz îl constituie *agregarea de legături* (Link Aggregation în limba engleză). Aceasta înseamnă folosirea mai multor legături fizice între aceleași două switch-uri, pentru a crește lărgimea de bandă între ele. La transmiterea pachetelor se folosește un algoritm (simplu round-robin sau politici mai avansate de distribuire) pentru a alege una dintre legăturile fizice. În aceste condiții, pachete emise de aceeași stație pot fi transmise pe legături fizice diferite și, prin urmare, pot sosi într-un switch prin porturi diferite. Totuși, atunci când se folosește agregarea de legături, adresele MAC nu sunt învățate pe porturile fizice, ci pe un port virtual asociat tuturor porturilor fizice implicate în agregarea de legături.

în condițiile în care în rețea există doar două VLAN-uri. Dacă însă numărul de VLAN-uri ar crește, ar atrage după sine creșterea numărului de legături fizice între echipamentele implicate. Acest lucru este inacceptabil, mai ales dacă rețeaua se întinde într-o clădire întreagă (sau chiar în mai multe clădiri alăturate) și conține un număr foarte mare de echipamente.

Pentru a combate acest neajuns, proiectanții au venit cu o soluție foarte elegantă.

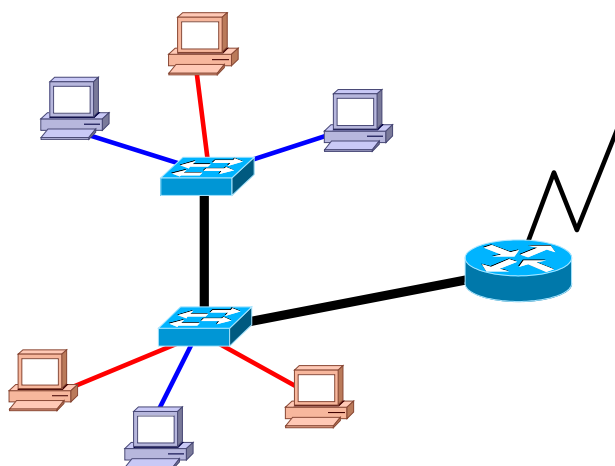


Figura 4 □ Organizarea unei rețele cu porturi în trunchi

Din moment ce oricum există cel puțin o legătură fizică între echipamente, aceasta poate fi folosită pentru a transmite pachetele pentru toate VLAN-urile implicate. Singura problemă o reprezintă modalitatea prin care se face identificarea VLAN-ului din care face parte pachetul.

Soluția a fost extinderea familiei de protocoale IEEE¹⁵ 802 cu încă un protocol, respectiv 802.1q. Acesta specifică modul în care se fac marcarea pachetelor la trimitere și respectiv identificarea VLAN-ului din care fac parte la recepție.

Src MAC (6 bytes)	Dst MAC (6 bytes)	Tip/Lungime (2 bytes)	Date (46-1500 bytes)	FCS (4 bytes)
----------------------	----------------------	--------------------------	-------------------------	------------------

Figura 5 □ Formatul cadrului Ethernet

Figura 5 ilustrează formatul unui cadru Ethernet. Nu au fost figurate câmpurile *Preamble* și *Delimiter de început a cadrului*, deoarece acestea sunt întotdeauna procesate în hardware (la recepție chip-urile nu depun aceste câmpuri în memoria centrală a sistemului, iar la trimitere le adaugă automat). Ele nu sunt niciodată accesibile din software și prin urmare nu prezintă interes.

Câmpul *Tip/Lungime* poate să conțină fie lungimea zonei de date utile (unitățile de date ale protocolului încapsulat¹⁶), fie tipul cadrului. În general, câmpul conține un tip atunci când este vorba de un protocol de nivel 2, cum ar fi ARP¹⁷. Atunci când cadrul încapsulează un

15 Institute of Electrical and Electronics Engineers.

16 În limba engleză *Protocol Data Unit* sau, prescurtat, *PDU*.

17 Address Resolution Protocol □ Protocol de rezoluție a adreselor; este protocolul prin care o stație poate identifica adresa fizică (de nivel 2) a unei alte stații atunci când îi cunoaște adresa de rețea (de nivel 3).

protocol de nivel mai înalt se presupune că acesta are fie pachete de lungime fixă, fie propriul câmp de lungime a datelor și, prin urmare, nu este necesară păstrarea acestei informații în cadrul de la nivelul 2.

Natura câmpului (tip sau lungime) poate fi determinată cu ușurință. Lungimea maximă a datelor unui cadru Ethernet este de 1500 bytes. Asignând valori superioare tuturor codurilor care identifică protocoale, nu apare riscul unei suprapunerii de codificare între tip și lungime.

Protocolul 802.1q folosește un artificiu remarcabil: cadrele Ethernet sunt încapsulate folosind propriul lor format¹⁸. Figura următoare ilustrează modul în care se realizează încapsularea.

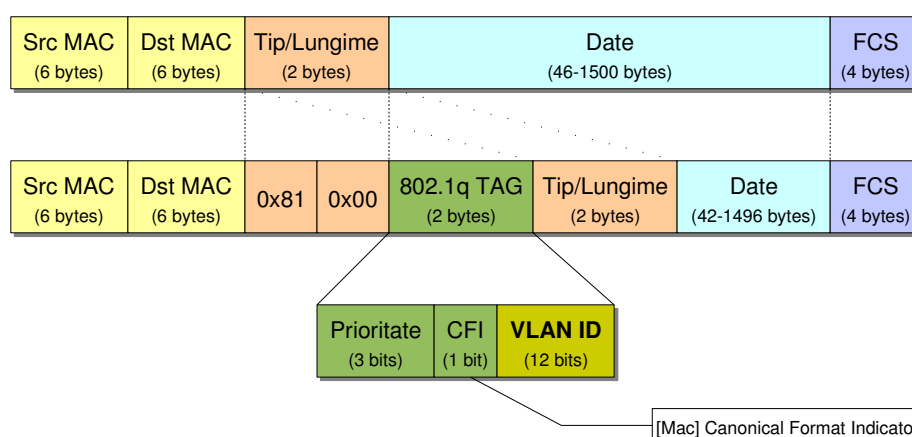


Figura 6 □ Încapsularea 802.1q

Se poate observa că tipul cadrelor 802.1q este 0x8100 și că sunt sacrificați 4 bytes de date pentru a insera informațiile specifice: marcajul 802.1q (2 bytes) și conținutul câmpului *Tip/Lungime* al cadrului original (2 bytes).

La o analiză atentă se poate constata că nu este vorba chiar de o încapsulare "Ethernet în Ethernet". Ca să fie așa, în câmpul de date ar fi trebuit să fie depus integral cadrul încapsulat, adică *inclusiv câmpurile de adrese MAC*. În practică acest lucru ar fi fost inutil: cadrul circulă între aceleași stații, chiar dacă marcăm VLAN-ul în care a fost generat. Prin urmare adresele rămân aceleași și se pot folosi câmpurile de adrese ale cadrului care încapsulează.

Pe scurt, ceea ce se încapsulează este un cadru Ethernet *fără* câmpurile de adrese. De aceea, în literatura de specialitate, atunci când se vorbește despre 802.1q se folosește în general termenul de *marcare (tagging, în limba engleză)* și nu cel de încapsulare.

Așa cum se poate observa și în Figura 6, marcajul 802.1q este format din 16 biți, împărțiți în trei câmpuri¹⁹:

- *Priority* (prioritate) □ specifică prioritatea utilizator, lăsând 8 valori posibile. Modul de

¹⁸ Această metodă este destul de des întâlnită în lumea protocoalelor de transmisie de date. De exemplu, este des folosită încapsularea pachetelor IP în alte pachete IP. Tunelele GRE încapsulează pachete IP în pachete UDP și astfel pot realiza, de exemplu, transportul de date între două rețele private (și de cele mai multe ori cu adrese IP nerutabile), folosind o legătură publică.

¹⁹ Descrierea câmpurilor, precum și a formatului cadrului 802.1q, au fost preluate din [Javvin01].

folosire a celor 3 biți de prioritate este specificat de standardul 802.1p.

- *CFI - Canonical Format Indicator* (indicator de format canonic) ¶ este întotdeauna pe 0 pentru switch-urile Ethernet. Se folosește pentru compatibilitate între rețelele Ethernet și rețelele Token Ring. Dacă un cadru primit pe un port Ethernet are CFI setat, atunci nu ar trebui să fie comutat, în formatul în care este, spre un port care nu folosește marcare.
- *VID - VLAN ID* ¶ identificatorul de VLAN ¶ este elementul cheie folosit în 802.1q. Are 12 biți și permite identificarea a 4096 (2^{12}) VLAN-uri. Dintre acestea, 0 este folosit pentru identificarea cadrelor prioritare, iar 4095 este rezervat. Rămân 4094 VLAN-uri efectiv utilizabile.

Revenind acum la Figura 4, se poate explica destul de clar ce sunt de fapt legăturile simbolizate cu negru. Este vorba de legături fizice pe care pachetele circulă marcate 802.1q. Desigur, echipamentele de la capetele unei astfel de legături trebuie să fie configurate special pentru a marca pachetele la trimitere și pentru a înlătura marcajul la primire.

Un port conectat cu o astfel de legătură fizică și configurat ca atare poartă denumirea de port *în trunchi*²⁰. Switch-urile mai complexe au facilități avansate pentru configurarea porturilor în trunchi. Ele permit construirea unei liste de VLAN-uri ale căror pachete să fie trimise (comutate) pe portul respectiv, precum și a VLAN-urilor permise pentru pachetele primite.

Deși conceptul de VLAN a fost prezentat destul de amănunțit, se impun câteva precizări suplimentare. Pornind de la exemplul cu mai multe switch-uri clasice, am arătat cum a apărut soluția VLAN-urilor, apoi am introdus în mod natural ideea apartenenței fiecărui port la un VLAN. În realitate există mai multe modalități de a desemna apartenența la VLAN-uri și, pentru ca expunerea să fie completă, le voi prezenta succint pe fiecare.

Apartenența la VLAN-uri *pe bază de port* este cea mai simplă variantă. Este practic ceea ce am prezentat pe larg până acum: fiecare port are asociat un VLAN și toate pachetele care intră în switch se consideră că fac parte din VLAN-ul asociat portului. Apartenența unei stații la un VLAN este practic determinată de configurația portului din switch la care este conectată stația. Această metodă este cel mai des folosită, tocmai de aceea nu am specificat de la început că mai există și altele. În plus, are marele avantaj că nu necesită eforturi suplimentare pentru procesare. La switch-urile bazate pe componente ASIC²¹ practic nu este nevoie de timp de procesor pentru procesarea VLAN-urilor, aceasta putând fi făcută integral în hardware.

Apartenența la VLAN-uri *pe bază de adresă MAC* presupune existența unei table de corespondență între adrese MAC și VLAN-uri. Identificarea VLAN-ului din care face parte un pachet primit se face căutând adresa MAC *sursă* a pachetului în tabela de corespondență. Avantajul ar fi că mutarea unei stații în alt port nu necesită modificări în configurația switch-ului. Dezavantajul este că tabela de corespondență (și căutarea prin ea)

20 Denumirea provine din limba engleză (*Trunk Port*). Este de notat că eu am folosit terminologia impusă de Cisco Systems. În literatura de specialitate termenul de *Trunking* este în general folosit cu sensul de "agregare de legături", în timp ce porturile configurate să folosească 802.1q sunt numite *Tagged Ports* (*porturi marcate*, sau *porturi cu marcare*). Cisco Systems utilizează chiar termenul de *Aggregation* pentru a denumi agregarea de legături.

21 *Application-Specific Integrated Circuit* (Circuit integrat specific aplicației) ¶ reprezintă un circuit integrat proiectat special pentru a implementa funcționalitatea unei anumite aplicații.

este destul de greu de implementat în hardware și de aceea, în general, se recurge la soluții hibride, ceea ce influențează în mod negativ performanța.

Apartenența la VLAN-uri *pe bază de protocol* se referă de fapt la adresa protocolului încapsulat (cel de nivel 3). Metoda este similară cu cea anterioară, cu diferența că pentru identificarea VLAN-ului se folosește adresa de rețea (adresa IP în cazul în care este vorba de rețele bazate pe familia TCP/IP) în loc de adresa MAC.

2.3. Comutare de pachete la nivel 3

Revenind la exemplul din Figura 4, se poate constata destul de ușor că are un dezavantaj major în ceea ce privește performanța: traficul între cele două VLAN-uri trece obligatoriu prin router (este singura cale de acces, din moment ce switch-urile asigură o izolare completă între VLAN-uri); singura legătură a rețelei cu routerul este cea figurată cu negru. Reamintesc că din punct de vedere logic acea legătură transportă mai multe VLAN-uri folosind marcarea 802.1q, în timp ce din punct de vedere fizic este o legătură similară²².

Pentru ca un pachet să ajungă de la o stație din VLAN-ul "roșu" la o altă stație din cel "albastru", el va traversa legătura dinspre switch spre router, va fi rutat în VLAN-ul "albastru", apoi se va întoarce prin aceeași legătură fizică spre switch. Este destul de ușor de dedus că practic traficul între cele două VLAN-uri se dublează pe legătura dintre switch și router.

Materialele publicate de Cisco Systems (cum ar fi [Cisco01]) vorbesc despre regula "80-20", care afirmă că 80% din trafic rămâne în interiorul VLAN-ului, iar 20% este rutat în afară. Dacă atât legătura dintre switch și router cât și cele către stații sunt de 100Mbit, se poate calcula simplu numărul de stații pentru care (statistic vorbind) lărgimea de bandă este suficientă:



Prin urmare lărgimea de bandă este suficientă pentru doar două stații în fiecare VLAN. Dacă am avea legături fizice separate cu router-ul pentru fiecare VLAN (ca în exemplul din

Figura 2, termenul  ar dispărea și am putea avea 5 stații în fiecare VLAN. Dacă viteza

legăturii dintre switch și router ar fi de 10 ori mai mare decât cea a legăturilor dintre switch și stații, am putea avea 25 de stații în fiecare VLAN, chiar în cazul unei singure legături fizice care folosește 802.1q.

Concluzia este că scalabilitatea rețelei (performanțele acesteia în cazul creșterii) este puternic influențată de legătura prin care se realizează traficul între VLAN-uri.

Tot conform Cisco Systems, rețelele mari, la nivel de corporație, tind să își centralizeze resursele. Traficul devine destul de greu de menținut în interiorul VLAN-urilor și este foarte probabil ca regula de distribuție să ajungă "20-80", adică 20% din trafic rămâne în interiorul VLAN-ului, în timp ce 80% este rutat în afară. În aceste condiții, estimările făcute anterior vor fi deveni extrem de pesimiste, chiar în cazul folosirii unei legături de

²² Din considerente de performanță, în practică această legătură (denumită de multe ori *uplink*) se realizează pe interfețe de viteză superioară. Spre exemplu, dacă stațiile din rețea sunt conectate în porturi FastEthernet (100 Mbit), pentru legătura uplink se folosesc porturi Gigabit (1000Mbit).

viteză mai mare între switch și router.

Pentru că situații similare celei pe care am descris-o sunt foarte des întâlnite în practică, proiectanții de echipamente de rețea au căutat să îmbine funcționalitatea switch-ului cu cea a routerului într-un singur dispozitiv. În acest mod, penalizarea de performanță introdusă de legătura dintre switch și router este eliminată.

Rezultatul a fost *switch-ul la nivel 3 (Layer 3 Switch)*. Figura 7 ilustrează o rețea construită cu un astfel de dispozitiv.

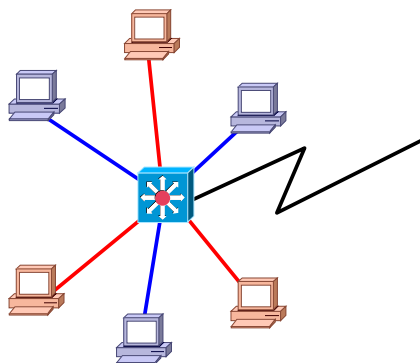


Figura 7 □ Rețea bazată pe comutare de pachete la nivel 3

Principiul de funcționare al comutării de pachete este destul de simplu. În fond, rutarea presupune în ultimă instanță tot o comutare de pachete, dar cu modificări ale adreselor sursă și destinație în antetul de nivel 2.

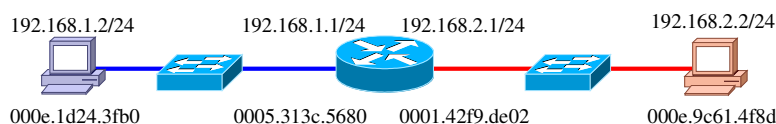


Figura 8 □ Rutarea pachetelor

Pentru ca rutarea pachetelor dintr-un VLAN în afara lui să fie posibilă, este necesară existența cel puțin a unei adrese de rețea asignată unui nod din VLAN. Acel nod trebuie să aibă o adresă MAC unică și să răspundă cererilor ARP. În cazul clasic (cel cu switch-uri și router separate) acel nod este chiar o interfață a routerului²³. Figura 8 ilustrează un exemplu foarte simplu de rutare. Dacă stația 192.168.1.2 dorește să trimită un pachet către 192.168.2.2, va emite pachetul având următoarea structură a adreselor:

	<i>Sursă</i>	<i>Destinație</i>
<i>Nivel 3</i>	192.168.1.2	192.168.2.2
<i>Nivel 2</i>	000e.1d24.3fb0	0005.313c.5680

După cum se poate observa, adresa MAC destinație este cea a interfeței routerului de pe același segment de rețea cu stația. Stația obține această adresă prin ARP după adresa de

²³ În cazul în care se folosește o singură interfață fizică și încapsulare 802.1q, există câte o interfață virtuală corespunzătoare fiecărui VLAN. Adresele de rețea ale routerului corespunzătoare fiecărui VLAN sunt asignate interfețelor virtuale.

rețea a interfeței routerului. Aceasta din urmă este configurată explicit pe stație și poartă numele de *Default Gateway*. La nivel 3, routerul va observa că pachetul nu îi este destinat lui, va lua o decizie de rutare și va trimite pachetul pe interfața corespunzătoare²⁴.

În cazul comutării la nivel 3, lucrurile sunt extrem de asemănătoare. Singura diferență este că nu mai există o interfață (fizică) a cărei adresă de rețea să poată fi folosită ca *Default Gateway*. Pentru ca procesul de rutare să rămână identic, switch-urile de nivel 3 au pentru fiecare VLAN câte o interfață *virtuală*. Aceasta are o adresă MAC proprie și i se pot asigura adrese de rețea.

Switch-ul de nivel 3 va răspunde la interogări ARP pentru adresele de rețea asignate interfețelor lui virtuale. Mai mult, atunci când switch-ul primește un cadru destinat adresei MAC a unei interfețe virtuale (și având adresa de rețea destinație diferită de a adreselor proprii), switch-ul va ști că nu este vorba de o simplă comutare (la nivel 2) și se va comporta ca atare. Are loc o decizie de rutare și se determină interfața "de ieșire".

Cum switch-ul de nivel 3 (în general) nu are asociate adrese de rețea decât pe interfețe virtuale, interfața "de ieșire" va fi una virtuală. Trimiterea unui pachet printr-o astfel de interfață presupune injectarea pachetului în VLAN-ul corespunzător interfeței, dar având ca adresă MAC sursă adresa interfeței virtuale. În continuare, se aplică algoritmul de comutare la nivel 2 pentru VLAN-ul în care a fost injectat pachetul, ca și când acesta ar fi fost primit printr-un port fizic din acel VLAN.

Prin urmare, procesul de comutare la nivel 3 presupune în esență modificarea adreselor MAC ale pachetului și aplicarea algoritmului de comutare la nivel 2. În plus, trebuie să fie cunoscută adresa MAC a stației destinație (sau a dispozitivului Next Hop atunci când ruta nu este direct conectată), ceea ce presupune un ARP înainte de injectarea pachetului în VLAN-ul destinație.

Decizia de rutare (care determină și modificarea adreselor MAC) și algoritmul de comutare la nivel 2 sunt delimitate din punct de vedere logic prin intermediul interfețelor virtuale. Aceasta duce la o înțelegere mai bună a procesului și la o configurare mai ușoară a echipamentelor. Totuși, switch-urile de nivel 3 performante implementează cea mai mare parte a acestui proces în hardware și atunci cele două subprocese se întrepătrund puternic.

2.4. Fluxul pachetelor în Linux. Socket Buffers

Implementarea sistemului de rețea în Linux a fost realizată astfel încât să fie independentă de protocol. Aici mă refer atât la protocoalele de nivel 2 (Ethernet, Token Ring etc.), cât și la cele nivel 3 și 4 (TCP/IP, IPX/SPX etc.). Se pot implementa cu ușurință protocoale noi, fără a fi necesare modificări majore ale codului deja existent.

Pentru a realiza abstractizarea necesară independenței de protocol, kernelul Linux folosește așa-numiții *socket buffers*. Din punct de vedere logic, un socket buffer este o entitate

²⁴ În cazul în care adresa de rețea destinație corespunde unei rute "direct conectată" (adică se poate ajunge direct la stația destinație printr-un broadcast la nivel 2), routerul însuși va face un ARP după adresa destinație din pachet pentru a determina adresa MAC a stației care trebuie să primească pachetul. Pachetul va ieși din router având la nivelul 2 ca adresă sursă cea a interfeței de ieșire a routerului, iar ca adresă destinație cea determinată prin ARP. În cazul în care ruta nu este direct conectată, în mod obligatoriu conține un așa-numit *Next Hop*, care reprezintă o adresă de rețea (a unui alt router) direct accesibilă. În mod similar unei stații, routerul va trimite pachetul către Next Hop, urmând ca acolo să se ia o nouă decizie de rutare.

formată din două părți:

- *Datele pachetului*: Reprezintă datele utile, ceea ce se transmite efectiv prin rețea.
- *Datele de control*: Pe parcursul procesării unui pachet de către kernelul Linux sunt necesare informații suplimentare, care nu are rost să fie stocate în interiorul pachetului. La nivelul codului, această componentă este o structură de date de tip *sk_buff*.

Cele două componente, deși puternic legate din punct de vedere logic, sunt păstrate separat la nivelul memoriei. O structură *sk_buff* conține informații necesare diferitelor straturi ale sistemului de rețea Linux, cum ar fi lungimea sau tipul pachetului la anumite niveluri sau pointeri în interiorul datelor, care au o semnificație specială. Aceste informații sunt folosite pentru interfațarea între diferitele protocoale, împreună cu cele transmise ca parametri la apelurile de funcții.

De multe ori atunci când spunem *socket buffer* ne referim la structura *sk_buff* împreună cu zona de date asociată. Există foarte multe argumente pentru ca datele utile ale pachetului să fie păstrate într-o zonă de memorie complet separată de structura *sk_buff*²⁵.

Structura *sk_buff* are lungime fixă (indiferent de pachetul pe care îl reprezintă) și este mică în comparație cu datele utile ale pachetului. În plus, dinamica structurilor *sk_buff* este extrem de mare. Toate acestea fac să aibă sens alocarea acestor structuri dintr-un cache²⁶.

Prin contrast, dimensiunea datelor utile nu este cunoscută dinainte și este variabilă. În plus, este suficient de mare încât folosirea unui cache (cu elemente de dimensiune maxim posibilă) să însemne risipă de memorie. De aceea, memoria pentru datele utile ale pachetelor este alocată cu apeluri normale *kmalloc()*.

Principalul avantaj pe care îl aduce Linux față de alte sisteme de operare este că datele pachetului nu sunt copiate pe măsură ce acesta traversează stiva de protocoale. Atunci când un proces scrie date într-un socket, acesta creează un socket buffer și alocă memorie pentru datele utile ale pachetului. Spațiul de memorie alocat este mai mare decât dimensiunea efectivă a datelor. Astfel, pe măsură ce pachetul coboară în stiva de protocoale, acestea îi pot adăuga informații specifice *fără a fi necesare realocări sau copieri* ale datelor deja existente. Similar, atunci când o interfață de rețea primește un cadru, *driverul* alocă un socket buffer, în care depune datele din cadru. Pe măsură ce acesta urcă în stiva de protocoale, acestea modifică pointerii către începutul (și, după caz, sfârșitul) datelor utile, conținuți în structura *sk_buff*. Prin urmare, nu sunt necesare copieri de date sau realocări de memorie.

Așa cum am arătat, un socket buffer este folosit pe întreaga durată de viață a unui pachet în kernelul Linux. Pentru ca datele unui pachet să ajungă de la procesul care le generează

25 Unul dintre cele mai pertinente argumente este cel că memoria pentru datele utile trebuie să fie alocată într-un segment capabil DMA pentru a putea realiza eficient transferurile cu hardware-ul interfeței de rețea. Pentru mai multe detalii, vezi Anexa B.

26 Este vorba de funcția de cache a alocatorului *slab*, care permite alocarea extrem de eficientă a unor zone de memorie de lungime fixă și relativ mică. În realitate este prealocată o zonă de memorie mai mare, capabilă să conțină mai multe elemente de cache. Atunci când se cere un nou element de cache, memoria este deja alocată și este suficient să se marcheze că spațiul corespunzător noului element este folosit. La eliberarea unui element cache nu are loc dealocare de memorie, ci doar marcarea spațiului corespunzător ca fiind liber, acesta putând fi imediat reutilizat pentru alocarea unui nou element. Astfel este eliminat overhead-ul algoritmilor de alocare și eliberare a memoriei kernel. Zona de memorie efectiv alocată este ajustată automat în funcție de numărul elementelor din cache.

până la hardware-ul care le transmite efectiv, sunt necesare doar două copieri ale datelor: una atunci când datele tranzitează din spațiul de memorie utilizator în spațiul de memorie kernel și încă una atunci când pachetul este pasat hardware-ului.

Am afirmat deja că alocarea unui spațiu de memorie mai mare decât datele utile ale pachetului este esențială pentru a asigura un număr minim de copieri ale datelor. Voi arăta în continuare cum este gestionat acest spațiu de către structura `sk_buff`.

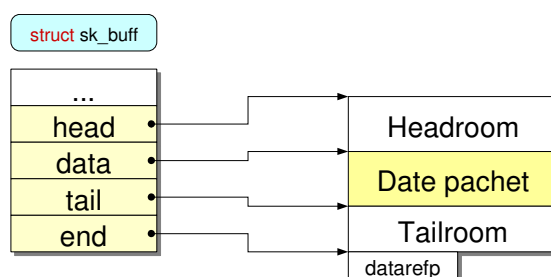


Figura 9 □ Structura unui Socket Buffer

Figura 9 ilustrează organizarea zonei de date utile ale pachetului, împreună cu câmpurile relevante din structura `sk_buff`. Pointerii `head` și `end` delimitează zona de memorie efectiv alocată pentru datele pachetului²⁷. Pointerii `data` și `tail` delimitează datele utile din interiorul zonei de memorie alocate. În spațiul de memorie alocat pentru pachet există două zone nefolosite, care au denumiri speciale. *Headroom* reprezintă spațiul nefolosit (și disponibil) dinaintea datelor utile. Acesta este delimitat de pointerii `head` și `data`. *Tailroom* reprezintă spațiul nefolosit (și disponibil) de după datele utile. Pe măsură ce un pachet coboară în stiva de protocoale, acestea consumă din *headroom* și *tailroom* pentru a-și adăuga informațiile specifice.

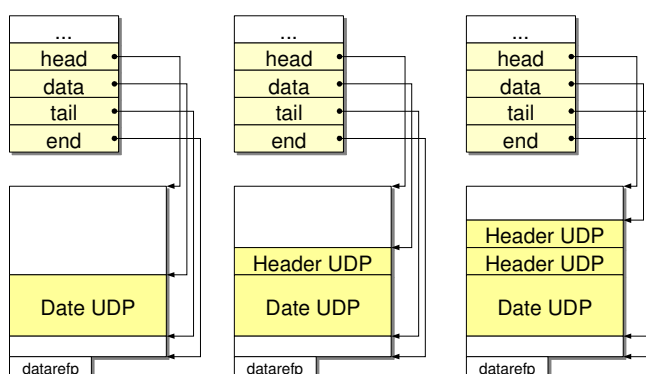


Figura 10 □ Încapsularea unui pachet UDP

Figura 10 ilustrează modul în care are loc încapsularea datelor unui pachet UDP pe măsură ce aceasta coboară prin stiva de protocoale. Se poate observa că practic are loc micșorarea *headroom*-ului, pe măsură ce antetele sunt adăugate înaintea datelor utile.

²⁷ În realitate, la alocare se rezervă un plus de memorie după `end`, de dimensiunea unui cuvânt mașină. Din punct de vedere al datelor pachetului, acest spațiu de memorie este inexistent, de aceea pointerul `end` este poziționat înaintea lui.

Împărțirea socket buffer-elor în date ale pachetului și date de control mai are încă un avantaj pe lângă cele prezentate până acum: aceleași date ale pachetului pot fi folosite în paralel de mai multe protocoale (atâta timp cât nici unul dintre ele nu trebuie să modifice datele utile). Deși la prima vedere acesta nu reprezintă un avantaj major, folosirea în comun a datelor utile este o facilitate esențială pentru o serie de optimizări. Spre exemplu, atunci când se folosește un program de captură a pachetelor (cum ar fi tcpdump) pachetele primite pot fi procesate atât de programul de captură cât și de stiva TCP/IP fără a face o copiere suplimentară a datelor. Un alt exemplu este chiar LISA, în cazul trimiterii aceluiași pachet pe mai multe porturi de ieșire (broadcast sau multicast). Acest caz este prezentat pe larg în secțiunea dedicată implementării.

Partajarea datelor nu este însă atât de simplă. Nu trebuie pierdut din vedere faptul că memoria ocupată de socket buffer trebuie eliberată (fie că este vorba de structura `sk_buff` alocată dintr-un cache sau de datele utile care sunt alocate direct), iar lucrurile se complică atunci când există fire de execuție diferite care au acces la pachet. Eliberarea memoriei în acest caz poate duce la curse critice. Dacă un fir de execuție eliberează socket buffer-ul, zona respectivă de memorie ar putea fi refolosită (și prin urmare suprascrisă) înainte ca celelalte fire de execuție să fi terminat de procesat datele. Având în vedere dinamica pachetelor în kernel, nici nu se poate vorbi de mecanisme de sincronizare.

Soluția a fost foarte simplă, respectiv introducerea unui contor de utilizare. Un socket buffer nu este niciodată dealocat explicit, în schimb este apelată funcția `dev_kfree_skb()`, care realizează corect eliberarea socket buffer-ului. Această funcție decrementează contorul de utilizare și, în cazul în care aceasta ajunge la zero, eliberează efectiv memoria utilizată de structură²⁸.

Linux implementează o arhitectură modulară și extrem de flexibilă pentru procesarea pachetelor primite. Prototipul rutinelor de tratare este fix, în schimb acestea pot fi adăugate dinamic în kernel. Există o listă globală de rutine de tratare (care vor fi apelate indiferent de tipul pachetului) și liste separate, indexate după protocolul de nivel 3 (acestea permit înregistrarea unei rutine de tratare specifice unui protocol de nivel 3). Întreaga logică de procesare a pachetelor primite este implementată în funcția `netif_receive_skb()`. Aceasta parcurge întâi listele de rutine de tratare generice, apoi lista corespunzătoare protocolului de nivel 3 specificat în pachetul procesat. Ori de câte ori funcția `netif_receive_skb()` apelează o rutină de tratare (pasându-i o referință spre structura `sk_buff` a pachetului procesat), această funcție incrementează explicit contorul de utilizare al structurii `sk_buff`²⁹.

Există totuși situații în care partajarea datelor nu este suficientă atunci când mai multe fire de execuție procesează același pachet. Așa se întâmplă în cazul în care unul dintre firele de execuție trebuie să modifice în vreun fel socket bufferul. Este de observat că modificarea socket buffer-ului poate însemna modificări în oricare dintre cele două componente ale sale. Prin urmare, se disting două cazuri:

- Se fac modificări numai în datele de control, iar datele pachetului sunt doar citite. În acest caz se pot copia doar datele de control și zona de date ale pachetului poate fi în continuare partajată. Operația de copiere a datelor de control (și partajare a datelor

28 Subliniez din nou că memoria respectivă nu este *dealocată*, ci doar *eliberată*, deoarece face parte dintr-un cache.

29 În Linux 2.6.10 incrementarea are loc în funcția `deliver_skb()`. Aceasta este o funcție inline care incrementează contorul de referință (`skb->users`) și apelează o rutină de tratare a pachetului.

pachetului) se numește *clonare*. Funcția Linux care implementează această operație este `skb_clone()`. Aceasta alocă din cache spațiu pentru o nouă structură `sk_buff` și copiază majoritatea câmpurilor din structura originală. Contorul de utilizare al clonei este inițializat la 1, în schimb este setat un alt câmp (`skb->cloned`) pentru a indica faptul că mai există o altă structură `sk_buff` care partajează aceleași date ale pachetului (și, prin urmare, acestea nu pot fi modificate, deși contorul de utilizare din noua structură `sk_buff` are valoarea 1).

- Se fac modificări în zona datelor pachetului. Acest caz presupune copierea datelor pachetului. Cum acestea nu pot exista (sau nu au sens) de sine stătătoare, este necesară și copierea structurii `sk_buff`. Operația se numește *copierea* socket buffer-ului și este implementată în funcția `skb_copy()`. Aceasta alocă o nouă structură `sk_buff` și copiază majoritatea câmpurilor din cea originală. În plus, este alocat spațiu de memorie pentru datele pachetului și conținutul pachetului original este copiat.

Evident, clonarea unui socket buffer este mult mai eficientă decât copierea. Am arătat deja motivele pentru care alocarea unei structuri `sk_buff` este mult mai rapidă decât alocarea memoriei pentru datele utile. În plus, copierea socket bufferului, pe lângă alocarea memoriei pentru datele pachetului, presupune și copierea acestora, mult mai mari în comparație cu dimensiunea structurii `sk_buff`.

Deși este mai eficientă, clonarea ridică o problemă suplimentară: cea a dealocării memoriei pentru datele utile ale pachetului. În cazul în care un socket buffer a fost clonat, contorul de utilizare din interiorul structurii nu mai este relevant, deoarece pot exista alte structuri `sk_buff` care partajează aceleași date. Prin urmare, este necesar un contor suplimentar pentru a număra referințele către o zonă de date pachet. Cum nu există o listă a referințelor către aceeași zonă de date pachet, singura soluție este de a păstra contorul chiar în interiorul zonei de date pachet. Este vorba de câmpul *datarefp*, care apare în Figura 9, dar pe care am omis în mod intenționat să-l menționez la momentul respectiv. La clonarea unui socket buffer, acesta este automat incrementat. Atunci când are loc eliberarea unei structuri `sk_buff` (se cere eliberarea socket bufferului și contorul de utilizare al structurii ajunge la zero), contorul de referințe din zona de date pachet este și el decrementat. Abia atunci când și aceasta ajunge la zero este dealocată efectiv memoria pentru zona de date ale pachetului.

Înainte de a încheia subsecțiunea dedicată socket buffer-elor, mă simt dator să mai fac încă o observație. La prima vedere ar putea să pară că operația de clonare nu are sens: dacă nu se modifică datele, se poate partaja inclusiv structura `sk_buff`, iar dacă se modifică, oricum este necesară o copie. În realitate există însă multe situații în care clonarea este utilă, iar una dintre ele este chiar comutarea pachetelor în cazul în care *același* pachet trebuie trimis pe mai multe interfețe *de același tip*³⁰.

Atunci când este apelată funcția `netif_receive_skb()`, antetul de nivel 2 al pachetului a fost

30 "Același tip" se referă la protocolul de nivel 2. În cea mai mare parte a cazurilor, atunci când se vorbește de comutare de pachete aceasta se realizează între interfețe de același tip, prin urmare coincide și protocolul de nivel 2. Există totuși și cazuri în care tipul interfețelor poate fi diferit (de exemplu comutare între Ethernet și Token Ring). Proiectul LISA a fost conceput și testat doar pentru interfețe de rețea Ethernet.

deja eliminat³¹ (headroom-ul a fost crescut și pointerul data indică spre PDU³² de nivel 2). Este important de reținut că, deși pointerul data este crescut, antetul de nivel 2 rămâne intact. Atunci când un socket buffer este depus pentru trimitere în coada unui dispozitiv de rețea, câmpul data trebuie să indice spre începutul cadrului (inclusiv headerele). Prin urmare, atâta timp cât interfața de ieșire este de același tip cu cea de intrare, nu mai este necesară reconstruirea antetului de nivel 2, deoarece este identic cu cel original (iar antetul nu a fost modificat în headroom). Este suficientă coborârea pointerului data cu o valoare egală cu dimensiunea antetului.

Atunci când același pachet trebuie trimis pe mai multe interfețe de ieșire, pointerul data trebuie coborât înainte de trimiterea pe fiecare dintre interfețe. Modificarea pointerului data înseamnă de fapt modificarea structurii `sk_buff`, fără însă a modifica datele utile ale pachetului. Prin urmare clonarea este operația care trebuie efectuată în această situație pentru a asigura consistența datelor.

2.5. Recepția cadrelor în Linux. Softnet și NAPI

Realizarea unei implementări eficiente și corecte a comutării de pachete a presupus studiul aprofundat al codului de recepție a cadrelor în Linux. Înțelegerea fenomenelor care se petrec la recepția pachetelor a fost esențială pentru a implementa corect sincronizarea și pentru a lucra corect cu socket buffer-ele primite. De asemenea, studiul stivei de recepție s-a dovedit extrem de util pentru implementarea interfețelor virtuale³³.

Implementarea recepției de pachete în Linux a cunoscut modificări majore în fiecare versiune nouă a Linux kernel. Prezentarea întregii istorii depășește obiectul acestui material, așa că mă voi limita la o prezentare succintă a penultimei versiuni, discutând avantajele pe care le aduce față de aceasta versiunea cea mai recentă (din Linux 2.6). Cei interesați de toate variantele de implementare și de o discuție a problemelor pe care le-au avut pot găsi mai multe amănunte în [Salim01].

Principalul avantaj pe care l-a adus implementarea recepției de pachete din Linux 2.4 (denumită *Softnet*) a fost procesarea pachetelor primite în context softirq³⁴. Având în vedere că în sistemele SMP poate rula *același* softirq *simultan* pe mai multe procesoare, este posibilă procesarea simultană a unui număr de pachete egal cu numărul de procesoare din sistem.

Pentru a putea amâna procesarea pachetelor până la rularea efectivă a softirq, este necesară o coadă în care pachetele să fie depuse de către rutina de tratare a întreruperii hardware³⁵.

31 În structura `sk_buff` există totuși un câmp (un pointer), `mac.raw`, care indică în interiorul zonei de date ale pachetului, către locația adreselor de nivel 2. Acest câmp este setat înainte de a intra în `netif_receive_skb()`, iar la momentul intrării în `netif_receive_skb()` indică undeva în headroom, înapoi față de pointerul data. Setarea câmpului `mac.raw` (și a altor câmpuri) și urcarea pointerului data sunt responsabilitatea *driver-ului* interfeței de rețea.

32 Protocol Data Unit. Reprezintă datele utile la nivelul unui protocol, respectiv ceea ce protocolul încapsulează în propriul pachet adăugându-și informațiile specifice. De exemplu, pentru cadrul Ethernet (ilustrat în Figura 5), PDU reprezintă câmpul "Date".

33 Este vorba de interfețele virtuale folosite pentru a face posibilă rutarea inter-VLAN. Mai multe detalii pot fi găsite în secțiunea dedicată implementării.

34 Prescurtarea vine de la *Software Interrupt Request* (cerere de întrerupere software). Acesta este un context de execuție invocat de către scheduler și care poate fi *planificat* folosind apelul `cpu_raise_softirq()`. Mai multe detalii pot fi găsite în [Wehrle01], subsecțiunea 2.2.3.

35 Regulile de proiectare a driverelor pentru dispozitive impun ca rutinele de tratare a întreruperilor

Ulterior ele vor fi extrase și procesate de codul din `softirq`.

Pentru exemplificarea stivei de apeluri la recepția de pachete în Softnet, am ales driverul `8139too`³⁶. Tabelul următor prezintă apeluri care au loc de la apariția întreruperii hardware până la procesarea efectivă a pachetului. Prima coloană trebuie interpretată astfel: funcțiile care apar cu nivel de indent mai mare sunt apelate din funcția de deasupra cu nivel de indent mai mic.

<i>Funcție</i>	<i>Observații</i>
<code>rtl8139_interrupt</code>	Rutina de tratare a întreruperii hardware
<code>rtl8139_rx_interrupt</code>	Rutina de tratare a întreruperii hardware în cazul recepției de pachete ³⁷
<code>dev_alloc_skb</code>	Alocarea unui socket buffer
<code>eth_type_trans</code>	Pregătirea socket buffer-ului ³⁸
<code>netif_rx</code>	Adaugă socket bufferul în coada de recepție a procesorului curent.
<code>cpu_raise_softirq</code>	Planifică pentru execuție <code>softirq</code> -ul <code>NET_RX_SOFTIRQ</code>
<code>net_rx_action</code>	Rutina de tratare pentru <code>softirq</code> -ul <code>NET_RX_SOFTIRQ</code>
<code>(struct packet_type).func</code>	Rutină generică de tratare a pachetelor ³⁹
<code>(struct packet_type).func</code>	Rutină de tratare a pachetelor specifică protocolului

Abordarea din Softnet a rezolvat majoritatea problemelor pe care le aveau implementările anterioare. A rămas totuși o problemă destul de mare: în cazul în care interfața de rețea primește un număr foarte mare de pachete⁴⁰, procesorul își petrece tot timpul tratând întreruperile generate de aceasta. Procesele utilizator nu mai ajung să fie planificate pentru execuție și sistemul se comportă ca și când ar fi blocat.

Câteva chip-uri se adresează acestei probleme suportând adaptarea dinamică a ratei de întreruperi prin feedback negativ: atunci când rata de pachete crește, nu se mai generează întreruperi pentru fiecare pachet în parte, ci o dată la câteva pachete (care sunt preluate în aceeași întrerupere). Această soluție a adus o îmbunătățire a performanțelor, dar, din păcate, este specifică hardware-ului folosit și deci nu este general implementabilă.

hardware să fie cât mai "scurte". Acestea trebuie să facă doar operațiile strict necesare, amânând pe cât posibil operațiile mari consumatoare de timp. Acestea din urmă pot fi planificate pentru execuție într-un alt context, cum ar fi un *tasklet* sau un *softirq*.

- 36 Acesta este driverul folosit pentru familia de chip-uri Realtek 8139 (variantele A,B,C și D). Ele sunt niște componente ieftine și destul de performante, ceea ce explică de ce sunt întâlnite în majoritatea interfețelor de rețea de pe piață.
- 37 În hardware se generează o întrerupere pentru toate evenimentele: s-a recepționat un pachet întreg, s-a terminat trimiterea unui pachet etc. Driverul, la apariția unei întreruperi, trebuie să interogheze un registru de stare din chip pentru a determina tipul evenimentului.
- 38 Aici are loc actualizarea câmpului corespunzător tipului de cadru și a câmpului *mac.raw*. Practic, socket bufferul este pregătit pentru a putea fi pasat rutinei de tratare a pachetelor primite. Mai multe detalii se pot găsi în subsecțiunea anterioară, dedicată socket buffer-elor.
- 39 Este vorba de sistemul de înregistrare dinamică a funcțiilor de tratare a pachetelor, despre care am vorbit mai pe larg în subsecțiunea anterioară.
- 40 Conform [Salim01], un sistem Pentium II folosit ca router atinge punctul de colaps la o intrare de 60000 pachete/secundă. La acest punct procesorul este utilizat 100% și nici un pachet nu mai iese din sistem.

Problema a fost soluționată complet o dată cu apariția implementării *NAPI*⁴¹. Ideea care stă la baza *NAPI* este dezactivarea întreruperilor hardware și interogarea chip-ului (device polling) în condițiile unei rate mari a pachetelor de intrare.

Deși interogarea este un cuvânt urât pentru dezvoltatorii de drivere dispozitiv, *NAPI* se bazează pe o observație pertinentă: interogarea este inefficientă doar atunci când în marea majoritate a cazurilor rezultatul ei este negativ (nu există evenimente care să fie tratate). Interogarea devine însă extrem de eficientă atunci când evenimentele apar extrem de des. Într-adevăr, atunci când pachetele sosesc în număr foarte mare, există *întotdeauna* pachete de procesat (se întâmplă chiar să fie mai multe pachete disponibile).

NAPI funcționează pe baza următorului mecanism:

- La apariția primei întreruperi hardware care semnalează primirea unui pachet, acesta este adăugat în coada de procesare și este dezactivată generarea de întreruperi la primirea unui pachet⁴². Interfața care a generat pachetul este adăugată pe o *listă de interogare* a procesorului curent și este programat *softirq*-ul corespunzător recepției de pachete.
- Rutina de tratare a *softirq* pentru recepția de pachete parcurge lista de interogare și apelează metoda *poll* a driverului. Aceasta interoghează hardware-ul și, în cazul în care există pachete primite și neprocesate, apelează rutina de tratare a pachetelor primite pentru fiecare dintre aceste pachete. Cum contextul *softirq* nu poate fi întrerupt pentru planificarea proceselor utilizator, există o limită maximă a numărului de pachete care pot fi procesate. După atingerea acestei limite, metoda *poll* este obligată să returneze controlul. Această limită asigură că procesorul este alocat și proceselor utilizator în condițiile în care pachetele sosesc atât de des încât bucla de interogare găsește mereu un pachet disponibil.
- În condițiile în care metoda *poll* nu mai găsește pachete neprocesate, scoate interfața din lista de interogare și reactivează generarea întreruperilor hardware.

Mecanismul descris reprezintă o modalitate de reglare automată a ratei întreruperilor hardware și are la bază un feedback negativ. În condițiile în care rata pachetelor este mică, metoda *poll* reactivează întotdeauna întreruperile și procesorul este liber până la sosirea unui nou pachet. În cazul în care rata pachetelor este mare, întreruperile rămân permanent dezactivate, iar limita de pachete procesate asigură alocarea procesorului și pentru procesele utilizator. Mai mult, în condițiile în care rata pachetelor este atât de mare încât acestea nu apucă să fie extrase în totalitate de metoda *poll*, hardware-ul poate ignora pachete fără a deranja procesorul⁴³.

41 Prescurtarea vine de la *New API* (New Application Programming Interface = Noua Interfață pentru Programarea Aplicațiilor). Implementarea este disponibilă în Linux 2.6 și chiar în Linux 2.4 începând cu versiunea 2.4.20.

42 Nu toate chip-urile suportă dezactivarea întreruperilor doar pentru un anumit tip de evenimente. În aceste condiții folosirea *NAPI* devine mai dificilă, dar nu imposibilă. Pentru mai multe detalii vezi .

43 Într-adevăr, majoritatea chip-urilor folosesc zone tampon circulare pentru recepția pachetelor. În condițiile în care sosesc foarte multe pachete și întreruperile sunt dezactivate, se poate întâmpla ca zona tampon să fie umplută și să înceapă să fie suprascrise pachete neprocesate până ca procesorul să ajungă să facă vreo interogare. În condițiile unei rate prea mari a pachetelor, o parte din acestea trebuie oricum ignorate, dar avantajul pe care îl aduce *NAPI* este că ignorarea pachetelor are loc de la sine, fără vreo intervenție a procesorului.

Tabelul următor prezintă apeluri care au loc de la apariția întreruperii hardware până la procesarea efectivă a pachetului în condițiile utilizării NAPI. Pentru exemplificare, am ales de asemenea driverul 8139too. Interpretarea tabelului trebuie făcută similar cu a aceluia care descrie stiva de apeluri Softnet.

<i>Funcție</i>	<i>Observații</i>
<code>rtl8139_interrupt</code>	Funcția de tratare a întreruperilor hardware
<code>netif_rx_schedule_prep</code>	"Prima jumătate" a apelului <code>netif_rx_schedule()</code> ⁴⁴
<code>__netif_rx_schedule</code>	A doua jumătate a apelului <code>netif_rx_schedule()</code> .
<code>list_add_tail</code>	Adaugă interfața în lista de interogare.
<code>__raise_softirq_irqoff</code>	Planifică apelarea <code>softirq NET_RX_SOFTIRQ</code> .
<code>net_rx_action</code>	Rutina de tratare pentru <code>softirq-ul NET_RX_SOFTIRQ</code>
<code>dev->poll</code>	Metoda <code>poll()</code> a driverului; în acest caz, <i>rtl8139_poll</i> .
<code>rtl8139_rx</code>	Rutina driverului de tratare a pachetelor primite.
<code>dev_alloc_skb</code>	Alocarea unui socket buffer.
<code>eth_type_trans</code>	Pregătirea socket buffer-ului.
<code>netif_receive_skb</code>	Rutina NAPI de tratare a pachetelor primite.
<code>deliver_skb</code>	Incrementează numărul de utilizări și apelează o rutină de tratare.
<code>(struct packet_type).func</code>	Rutină generică de tratare a pachetelor
<code>deliver_skb</code>	Incrementează numărul de utilizări și apelează o rutină de tratare.
<code>(struct packet_type).func</code>	Rutină de tratare a pachetelor specifică protocolului

La Softnet, rutina Linux de tratare a pachetelor primite este chiar rutina de tratare a `softirq-ului NET_RX_SOFTIRQ`. După cum se poate observa, la NAPI există o rutină separată de tratare, care este apelată explicit de către metoda `poll` a driverului.

Implementarea NAPI asigură compatibilitate perfectă cu driverele Softnet. Un driver proiectat pentru Softnet nu necesită nici un fel de modificări pentru a funcționa într-un kernel care folosește NAPI. Într-un astfel de kernel există în continuare coada de pachete per-procesor (numită *backlog*), astfel încât driverele Softnet pot în continuare să depună pachete în backlog apelând funcția `netif_rx()`. În plus, există un dispozitiv (de rețea) virtual (numit *backlog device*) care simulează funcționarea unui driver NAPI:

- Atunci când driverul Softnet apelează `netif_rx()` pentru a depune pachetul în coadă, dispozitivul backlog este planificat pentru interogare, întocmai ca un dispozitiv hardware care a generat o întrerupere de primire de pachet.
- Metoda `poll()` a dispozitivului backlog extrage pachete din coada de pachete

⁴⁴ Funcția `netif_rx_schedule()` este folosită de către rutinele de tratare a întreruperilor hardware din drivere pentru a planifica apelarea metodei `poll()` de către sistemul de rețea Linux. Această funcție este alcătuită din două subfuncții: `netif_rx_schedule_prep()` și `__netif_rx_schedule()`. Prima testează dacă interfața este pornită, iar a doua realizează planificarea efectivă. Pentru mai multe detalii vezi [Kuznetsov01].

per-procesor în loc să interogheze hardware-ul și să preia de la el pachetele (cum face metoda poll a unui driver NAPI real).

2.6. Trimiterea cadrelor în Linux

Deși în cazul comutării de pachete sau al rutării numărul de pachete care părăsesc sistemul este comparabil cu al celor care intră, trimiterea pachetelor nu ridică probleme de performanță atât de mari ca primirea. Dacă la primirea pachetului majoritatea hardware-ului de rețea generează implicit câte o întrerupere pentru fiecare pachet primit, pentru trimitere se folosește o zonă tampon circulară (ring buffer) care poate stoca mai multe pachete care trebuie trimise⁴⁵. Generarea întreruperii pentru semnalizarea terminării trimiterii se face abia după ce zona tampon a fost golită. Astfel, o întrerupere apare pentru mai multe pachete și rata întreruperilor este mult mai mică decât în cazul recepției⁴⁶.

Cum trimiterea de pachete nu este critică din punct de vedere al performanței, atunci când s-a trecut de la Softnet la NAPI partea de trimitere a rămas nemodificată.

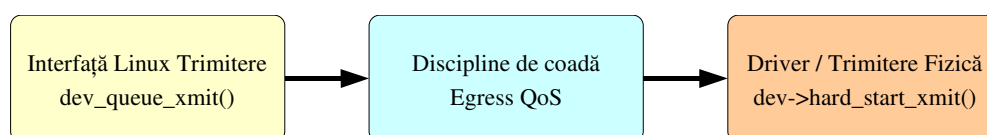


Figura 11 □ Schema generală a trimiterii de pachete

Figura 11 ilustrează schema generală a trimiterii de pachete în Linux. După ce pachetul a ajuns la nivelul cel mai coborât al stivei de protocoale, acesta nu este trimis direct⁴⁷, în schimb este depus în coada de trimitere corespunzătoare interfeței prin care trebuie trimis, folosind funcția `dev_queue_xmit()`. Cozile de trimitere fac parte din componenta *Network Scheduler* a Linux kernel. Aceasta se ocupă de planificarea trimiterii pachetelor, implementând diverse politici. Arhitectura cozilor de trimitere este extrem de complexă, asigurând flexibilitate și modularitate. Politicile de trimitere a pachetelor pot merge de la un simplu FIFO până la limitarea/garantarea traficului pe bază de anumite proprietăți ale pachetelor. Mai multe detalii despre acestea se pot găsi în [Wehrle01], în capitolul 18.

Planificarea trimiterii pachetelor are la bază tot un softirq, respectiv `NET_TX_SOFTIRQ`.

45 Chip-uri mai performante, cum ar fi RTL8139C+, permit stocarea în ring buffer a unor pointeri spre datele utile în loc de datele utile în sine. Această abordare oferă două avantaje majore: fiind vorba doar de pointeri spre datele utile, bufferul poate păstra mult mai multe pachete; în plus, același pachet nu trebuie să se găsească într-o zonă continuă de memorie. Diferite părți ale lui (cum ar fi antetul de nivel 2 și PDU de nivel 2) se pot găsi în zone de memorie separate. În buffer se depun doar pointerii spre cele două zone, iar "alipirea" lor se face direct în hardware la trimitere. Această facilitate poartă numele de Scatter-Gather I/O. Implementarea de rețea din Linux (în particular socket buffer-ele) suportă Scatter-Gather I/O și cooperează cu driverul pentru a optimiza asamblarea pachetelor (în cazul în care acesta suportă, la rândul lui, Scatter-Gather I/O).

46 Din considerente de lărgime de bandă limitată, rata pachetelor este invers proporțională cu dimensiunea lor. Prin urmare, atunci când rata pachetelor este mare, pachetele sunt mici și încap în număr mare în bufferele de trimitere. Cum întreruperea se generează doar la golirea bufferului, rezultă că rata întreruperilor depinde foarte puțin de rata pachetelor.

47 În cazul particular în care interfața nu definește metode pentru gestionarea cozii (metoda `dev->enqueue()` este NULL), pachetul este trimis imediat folosind metoda de trimitere fizică a driverului, `dev->hard_start_xmit()`. În general, acesta este cazul interfețelor virtuale, cum ar fi loopback sau tunelele. În LISA, este și cazul interfețelor virtuale care fac posibilă rutarea inter-VLAN.

Acesta permite procesarea cozilor asincron față de adăugarea pachetelor în coadă.

3. Arhitectură

Arhitectura LISA este destul de complexă și acoperă atât spațiul kernel cât și cel utilizator. O vedere "de la 1000 metri" asupra arhitecturii LISA este prezentată în figura următoare.

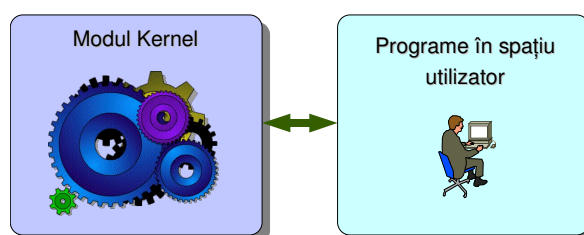


Figura 12 □ Arhitectura generală a LISA

Modulul kernel implementează tot mecanismul de comutare a pachetelor. Programele în spațiul utilizator permit configurarea, monitorizarea și controlul procesului de comutare.

Partea de spațiu utilizator a fost astfel proiectată încât să fie posibile mai multe sesiuni de configurare simultane. Practic un număr nelimitat de utilizatori se pot conecta pentru a face modificări în configurație sau pentru a monitoriza diverși parametri ai procesului de comutare. În aceste condiții însă, apar probleme de sincronizare și de consistență a datelor.

Proiectarea LISA s-a bazat pe câteva reguli simple:

- Toate opțiunile de configurare care influențează în vreun fel procesul de comutare sunt păstrate în kernel.
- Toate opțiunile de configurare care nu sunt legate de procesul de comutare sunt păstrate în spațiu utilizator într-un spațiu de memorie partajată. Accesul la memoria partajată este exclusiv și este controlat de un semafor.
- Toate configurările componentei kernel se fac prin intermediul unor apeluri `ioctl()`.
- Toate interogările de configurație și de stare se fac prin apeluri `ioctl()`. Interogările de stare (cum ar fi listarea tabeli de comutare) nu se fac prin alte metode (cum ar fi citirea dintr-un *char device* sau dintr-un fișier virtual din `/proc`) deoarece acestea nu suportă pasarea de opțiuni din spațiu utilizator (cum ar fi listarea doar a înregistrărilor din tabela de comutare corespunzătoare unui anumit port) și ridică probleme serioase de sincronizare⁴⁸.

48 Tabela de comutare este extrem de dinamică și puternic legată de procesul de comutare. Prin urmare zăvorărea ei pentru a asigura consistența datelor la listarea tabeli din spațiul utilizator este exclusă. Singura soluție rămâne sincronizarea RCU. Citirea din `/proc` este limitată la dimensiunea unei pagini (și nu se poate ști dinainte dacă aceasta este suficientă pentru tot ce se va lista). Citirea dintr-un caracter device nu este atomică (în sensul că buffer-ul alocat din spațiu utilizator pentru citire poate fi insuficient și atunci este nevoie de unul sau mai multe apeluri suplimentare `read()` pentru a transfera datele complet). Mai multe detalii despre modul în care este sincronizată listarea tabeli de comutare se pot găsi în

Toate aceste reguli au dus la o implementare simplă și ușor de dezvoltat. Sincronizarea sesiunilor de configurare este asigurată prin intermediul apelurilor `ioctl()` la nivelul componentei kernel⁴⁹ și prin intermediul semaforului la nivelul componentei utilizator.

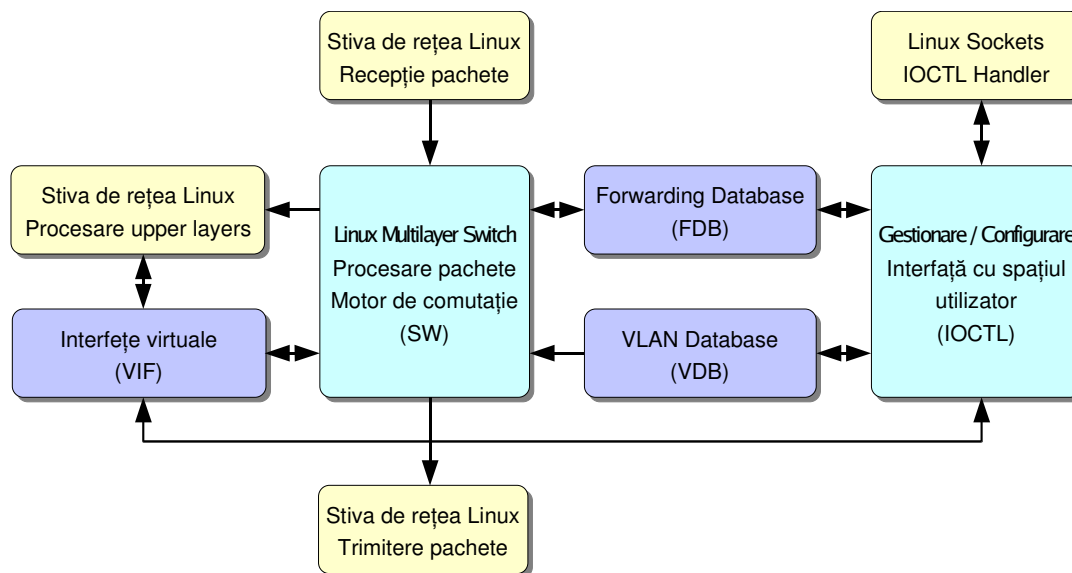


Figura 13 □ Arhitectura componente kernel a LISA

Figura 13 ilustrează arhitectura componente kernel a LISA. Se pot observa componentele de bază, precum și modul în care acestea sunt conectate între ele și cu alte componente deja existente ale Linux kernel. Principalele componente ale modulului kernel LISA sunt următoarele:

- SW □ reprezintă motorul de comutare a pachetelor, inima întregului proiect LISA.
- FDB □ reprezintă tabela de comutare, necesară funcționării oricărui switch. Pe lângă tabela de comutare în sine, componenta FDB înglobează toate funcțiile necesare manipulării acesteia (adăugare și ștergere de înregistrări, expirarea înregistrărilor dinamice, listarea din spațiul utilizator) împreună cu toată logica de sincronizare care asigură funcționarea corectă.
- VDB □ reprezintă baza de date de VLAN-uri, esențială pentru realizarea comutării în contextul VLAN-urilor.
- VIF □ reprezintă interfețele virtuale, necesare pentru a putea implementa rutarea inter-VLAN folosind stiva TCP/IP deja existentă în Linux.
- IOCTL □ reprezintă rutina de tratare a apelului `ioctl()`, împreună cu toate rutinele necesare pentru a implementa efectiv comenzile primite prin `ioctl()`.

Funcționarea acestor componente este descrisă pe larg în secțiunea următoare, dedicată

secțiunea dedicată implementării.

49 Pentru simplitate LISA folosește un singur `ioctl()`, de tip socket, respectiv `SIOCSWCFG`. În principiu "ioctl de tip socket" presupune efectuarea apelului folosind ca *file descriptor* un socket. În Linux există o rutină specială de tratare a apelurilor `ioctl()` pentru socket. Printre altele, la intrarea în această rutină este zăvorât un semafor kernel care asigură sincronizarea între mai multe apeluri `ioctl()`. Se consideră că apelurile `ioctl` nu se realizează intensiv și prin urmare serializarea lor folosind un semafor nu ridică probleme de performanță.

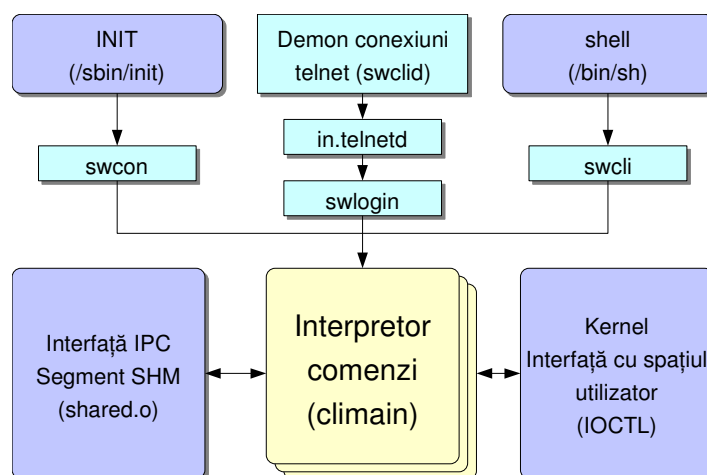


Figura 14 □ Arhitectura componentei utilizator a LISA

implementării.

Figura 14 prezintă arhitectura componentei utilizator în cazul unui sistem LISA dedicat⁵⁰. Componenta principală este interpretorul de comenzi. Acesta are forma unei biblioteci partajate (*shared library*) și înglobează toată funcționalitatea necesară operării cu un switch:

- *Shell-ul* □ reprezintă interfața în linia de comandă și se bazează pe bibliotecă *readline*. Are facilități avansate, cum ar fi completarea automată a comenzilor, istoria comenzilor și funcții help încorporate.
- *Parser-ul* □ realizează validarea și interpretarea comenzilor introduse de către utilizator. Funcționează în strânsă legătură cu shell-ul, pentru ca acesta să poată realiza completarea automată comenzilor și funcțiile help.
- *Executorul de comenzi* □ reprezintă componenta cea mai vastă a bibliotecii, care execută efectiv comenzile utilizatorului. Executorul de comenzi este în esență o colecție foarte mare de funcții care pregătesc și rulează apeluri `ioctl()` către componenta IOCTL a modulului de kernel.

După cum am afirmat deja, interpretorul de comenzi este de fapt o bibliotecă partajată, prin urmare nu este direct invocabil de către utilizator. Există însă mai multe interfețe pentru el, sub forma unor fișiere executabile care realizează câteva funcții suplimentare înainte de a pasa controlul către rutina principală a interpretorului de comenzi. În continuare voi prezenta pe scurt toate aceste interfețe:

- *swcon* □ este o interfață concepută pentru accesul de pe o consolă, cum ar fi portul serial al sistemului sau chiar o consolă virtuală pentru cazul în care sistemul dedicat LISA are placă video și tastatură. Această interfață a fost proiectată sub forma unui program de tip login, special pentru a putea fi invocată de către *init* prin intermediul unui program de tip *getty*. Particularitatea interfeței o constituie invocarea interpretorului de comenzi direct în nivelul minim de privilegii, fără a face în prealabil o autentificare.
- *swlogin* □ este o interfață proiectată tot sub forma unui program de tip login și destinată

⁵⁰ Prin "sistem LISA dedicat" mă refer la o mașină dedicată comutării de pachete și care rulează mini-distribuția de Linux inclusă în proiectul LISA.

invocării de către demonul *in.telnetd*⁵¹. Particularitatea acestei interfețe o reprezintă autentificarea completă (inclusiv verificarea parolei de acces pentru nivelul minim de privilegii) înainte de a pasa controlul către rutina principală a interpretorului de comenzi.

- *swcli* □ reprezintă cea mai simplă interfață pentru interpretorul de comenzi. Controlul este pasat direct către acesta și se pornește din nivelul maxim de privilegii. Interfața este destinată invocării dintr-un shell Linux standard și a fost concepută pentru sistemele nededicate și pentru testare și depanare. Menționez că această interfață nu implică riscuri de securitate din moment ce interpretorul de comenzi are oricum nevoie de privilegii de root pentru a putea funcționa.

51 Demonul *in.telnetd* (inclus în majoritatea distribuțiilor de Linux) este folosit pentru implementarea protocolului telnet. Demonul asigură crearea unui terminal virtual (un *pty* □ pentru mai multe detalii vezi [Lawyer01]), controlul terminalului virtual și interpretarea secvențelor escape specifice protocolului telnet.

4. Implementare

Trebuie să menționez încă de la început că în Linux exista deja suport pentru comutare de pachete (modulul *bridge*) și suport pentru marcarea 802.1q a pachetelor (modulul *8021q*). Folosind funcționalitatea oferită de cele două module am reușit să testez cu succes în condiții de laborator toată funcționalitatea pe care am descris-o în partea teoretică a lucrării: comutare de pachete, porturi în trunchi și chiar rutare între VLAN-uri. Mai multe detalii se pot găsi în Anexa D.

Deși posibilă, implementarea funcționalității unui switch de nivel 3 cu modulele standard din Linux este greoaie și ineficientă. În aceste condiții, implementarea suportului de VLAN-uri și a rutării inter-VLAN în LISA este pe deplin justificată prin proiectarea unitară a întregului cod, simplitate și eficiență.

Principalele dezavantaje pe care le are soluția *bridge* + *8021q* sunt următoarele:

- La porturile în trunchi este necesară crearea unui *net_device* virtual pentru fiecare VLAN.
- Este necesară crearea unui *bridge* separat pentru fiecare VLAN.
- Pentru a realiza comutare de pachete între două porturi în trunchi pentru n VLAN-uri, sunt necesare $3n$ *net_device*-uri (n *net_device*-uri virtuale *8021q* pentru fiecare dintre cele 2 porturi și n *net_device*-uri virtuale pentru fiecare *bridge*).
- Pentru comutarea la nivel 2 a pachetelor între două porturi în trunchi se face o eliminare a marcajului 802.1q, apoi o adăugare a acestuia, deși cele două operații nu sunt necesare din moment ce pachetele circulă marcate pe ambele porturi.
- La traficul de tip broadcast/multicast care implică și porturi în trunchi se fac mai multe modificări (adăugări sau eliminări) de marcaj 802.1q, deși pachetele rezultate sunt identice. O dată cu acestea are loc copierea întregului pachet, din moment ce modificarea de marcaj presupune copierea datelor asociate *sk_buff*-ului în condițiile în care acesta are mai multe clone.
- Modulul *bridge* nu suportă adrese MAC statice.
- Modulul *bridge* nu suportă în mod explicit multicast (traficul de tip multicast este tratat implicit ca trafic de broadcast din moment ce nu sunt suportate adrese MAC statice iar adresele MAC de multicast nu pot fi niciodată "învățate" pentru că nu sunt folosite ca adrese sursă).

Algoritmii folosiți de către LISA elimină toate aceste neajunsuri. Scopul proiectului a fost de a realiza o implementare cât mai eficientă folosind pe cât posibil ceea ce există deja în Linux și oferind în același timp un mod de configurare facil și cât mai asemănător cu Cisco IOS.

După cum reiese și din secțiunea care descrie arhitectura LISA, am folosit din componenta

de rețea a Linux tot ceea ce rămâne nemodificat în contextul VLAN-urilor, al porturilor în trunchi și al comutării multistrat:

- Driver-ele adaptoarelor de rețea, pentru accesarea generalizată și independentă de dispozitiv a funcțiilor de recepție și trimitere de cadre. Interfațarea cu acestea se face prin intermediul unui *hook* în rutina de tratare a cadrelor primite în cazul recepționării și prin intermediul cozilor de trimitere (generice) ale unui *net_device* în cazul trimiterii.
- Stiva TCP/IP, pentru a implementa comunicația cu mașina și rutarea. Interfațarea cu stiva TCP/IP se face prin intermediul unor *net_device*-uri virtuale. Spre deosebire de soluția *bridge + 8021q*, în LISA este suficient un singur *net_device* pentru fiecare VLAN, iar acesta este necesar doar în cazul în care se dorește rutarea în/din VLAN-ul respectiv.
- Abstractizarea *sk_buff* a pachetelor și funcționalitatea oferită de aceasta (clonare, copiere, realocare, contorizarea referințelor la zona de date ale pachetului etc.).

4.1. Modurile de acces al porturilor la VLAN-uri

În LISA prin *port* se înțelege în principiu o interfață fizică de rețea. Intern, porturile sunt reprezentate prin structuri de date (numite *net_switch_port*). Pe lângă un pointer spre structura de tip *net_device* care reprezintă interfața fizică, structura *net_switch_port* conține o serie de date de configurație și stare specifice funcționalității de comutare. Prezintă în continuare această structură, păstrând doar câmpurile cele mai importante.

```
struct net_switch_port {
    struct net_device *dev;
    unsigned int flags;
    int vlan;
    unsigned char *forbidden_vlans;
    ...
};
```

Spre deosebire de majoritatea switch-urilor, care privesc asocierea port-VLAN din perspectiva VLAN-urilor (adică asocierea se face atunci când se configurează VLAN-urile și unui VLAN *i* se asociază porturi), LISA se bazează pe abordarea din Cisco IOS. Prin urmare, asocierea se face din perspectiva porturilor și unui port îi sunt asociate unul sau mai multe VLAN-uri.

LISA distinge două moduri de funcționare a unui port:

- Modul *acces* (*access mode*). Portul primește și trimite doar pachete fără marcaj. Se configurează așa-numitul "VLAN de acces". Toate pachetele primite de către port se consideră că aparțin VLAN-ului de acces (și implicit adresele MAC sunt învățate pe acest VLAN) și portul este implicat în procesul de comutare atunci când este primit un pachet pe VLAN-ul de acces prin alt port.
- Modul *trunchi* (*trunk mode*). Portul primește și trimite doar pachete cu marcaj 802.1q. Se configurează o listă de "VLAN-uri permise" (implicit toate VLAN-urile sunt permise). Pachetele primite pe un VLAN care nu se află în listă sunt ignorate. Portul este implicat în procesul de comutare pentru toate pachetele primite pe unul dintre VLAN-urile din listă.

Modul în care este configurat un port este determinat de un bit al câmpului *flags* al

structurii `net_switch_port`. Bitul setat indică funcționarea în mod trunchi. Constanta simbolică (masca de biți) care identifică bitul respectiv se numește `SW_PFL_TRUNK`.

4.2. Baza de date de VLAN-uri. Asignarea porturilor la VLAN-uri

Dintre cele 4094 de VLAN-uri utilizabile (conform 802.1q), în general, se folosesc destul de puține în cadrul aceleiași rețele. În plus, pentru administratorul de rețea ar fi util să poată asigna câte un nume (cât mai semnificativ) fiecărui VLAN pe care îl folosește, pentru o identificare cât mai ușoară.

O bază de date de VLAN-uri, în principiu, conține lista VLAN-urilor utilizate și opțiunile de configurare specifice fiecărui VLAN, cum ar fi numele acestuia, intervalul de învechire a adreselor MAC învățate (acolo unde este suportat) etc.

În plus față de acestea, LISA păstrează tot în baza de date de VLAN-uri (pe care în continuare, pentru simplitate, o voi denumi pe scurt *VDB*⁵²) lista porturilor care au acces la fiecare VLAN (fie că sunt configurate în mod acces sau trunchi). Deși această informație este redundantă (din moment ce apare oricum în configurația specifică fiecărui port) este extrem de utilă în cazul traficului de tip broadcast, după cum voi arăta în continuare. În plus, în LISA nu are loc niciodată comutare de pachete pe un VLAN care nu există în VDB. Cu alte cuvinte, dacă există două porturi configurate în trunchi și ambele conțin de exemplu VLAN-ul 10 în lista de VLAN-uri permise, pachetele sosite pe unul dintre cele două porturi prin VLAN-ul 10 nu sunt comutate spre celălalt fără ca VLAN-ul 10 să existe în VDB.

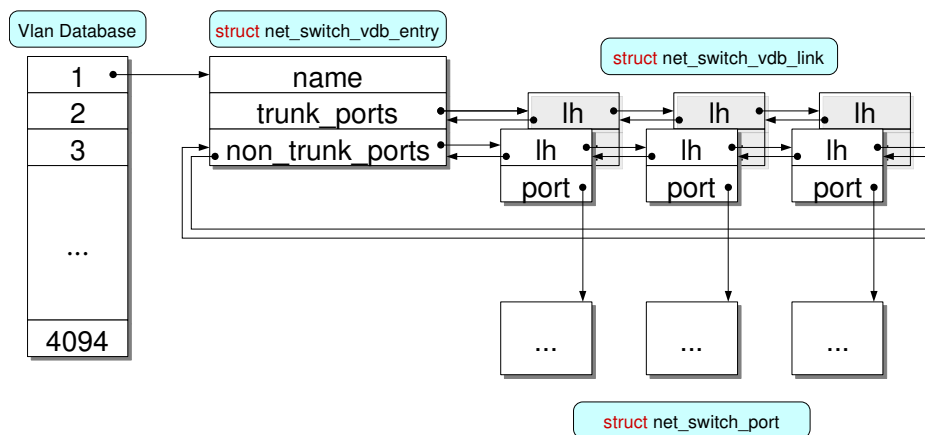


Figura 15 □ Arhitectura VDB

Arhitectura VDB a fost astfel concepută încât să optimizeze cât mai mult operațiile critice în timp. Profitând de faptul că nu există decât 4096 numere de VLAN, am sacrificat 16KB de memorie în favoarea performanței. După cum se poate vedea și în Figura 15, scheletul VDB îl reprezintă un vector de pointeri. Acesta permite localizarea unui VLAN (sau testul existenței acestuia în VDB) în $O(1)$. VLAN-urilor care nu există le corespunde un pointer NULL în acest vector.

Structurile corespunzătoare fiecărui VLAN (de tip `net_switch_vdb_entry`) sunt alocate dinamic la adăugarea unui nou VLAN în VDB și, respectiv, eliberate la ștergerea unui

52 Prescurtarea vine de la denumirea în limba engleză, respectiv VLAN DataBase.

VLAN. Fiecare astfel de structură conține (pe lângă descrierea VLAN-ului) două capete de liste: cea a porturilor în mod acces și cea a porturilor în mod trunchi care aparțin VLAN-ului respectiv. Aceste liste sunt esențiale pentru optimizarea broadcast-ului, după cum voi arăta în secțiunile următoare.

Cele două liste conțin structuri de tip `net_switch_vdb_link`, care la rândul lor conțin pointeri spre structurile corespunzătoare porturilor. Este de reținut că structurile care descriu porturile nu pot fi înlănțuite direct pentru că același port poate fi înlănțuit în listele corespunzătoare mai multor VLAN-uri. Prin urmare, în structura `net_switch_port` ar fi fost nevoie de câte un `list_head` pentru fiecare VLAN din care face parte portul. Numărul acestor VLAN-uri nu poate fi cunoscut dinainte (este dinamic) și folosirea unui vector cu câte un `list_head` pentru toate cele 4096 VLAN-uri posibile nu se justifică.

Operațiile asupra VDB sunt realizate doar în context proces, prin intermediul unui apel `ioctl()` pe un socket. Cum apelurile `ioctl()` pe socket sunt mutual exclusive (chiar din rutina kernel de tratare a apelurilor `ioctl()` pentru un socket), nu apar probleme de sincronizare între procese. Totuși, la ștergerea unui VLAN, pot să apară probleme de sincronizare între procesul care îl șterge și mecanismul de comutare de pachete (care rulează în context `softirq`). Mai multe detalii voi prezenta în secțiunea dedicată sincronizărilor.

4.3. Interfețele virtuale

Așa cum am arătat și în secțiunea teoretică, modul cel mai simplu de a implementa comutarea de pachete la nivel 3 (rutarea între VLAN-uri) este de a folosi câte o interfață virtuală pentru fiecare VLAN. Astfel se poate face o separare logică între procesul de comutare la nivel 2 și procesul de rutare.

Linux dispune deja de o stivă TCP/IP și de facilități foarte avansate de rutare, cum ar fi filtrarea pachetelor și rutarea după sursă (*source routing* sau *policy routing*). Pentru ca LISA să poată profita din plin de aceste facilități, este necesară o interfațare cu procesul de comutare de pachete (la nivel 2). Aceasta se realizează prin intermediul unei interfețe virtuale, pe care o voi denumi pe scurt *VIF*⁵³. Din punctul de vedere al stivei TCP/IP și al procesului de rutare, o VIF este un `net_device` de tip ethernet, întocmai ca o interfață de rețea fizică. Acestea i se pot asigna adrese de rețea, iar stiva TCP/IP împreună cu toate facilitățile oferite pot fi folosite în mod natural, fără a fi necesară vreo modificare în codul sursă sau vreo configurare specială. Din punctul de vedere al procesului de comutare la nivel 2, o VIF este un port, asemănător porturilor fizice.

Mecanismul de funcționare este extrem de simplu: atunci când procesul de comutare la nivel 2 *trimite* un pachet pe portul virtual, `net_device`-ul virtual *primește* pachetul; atunci când se *trimite* un pachet spre `net_device` (când politica de coadă extrage pachetul din coadă și apelează rutina de trimitere "hardware"), pachetul este injectat în procesul de comutare, invocându-se algoritmul corespunzător ca și când pachetul ar fi fost *primit* prin portul virtual asociat VIF.

Se impune o singură observație cu privire la ceea ce am afirmat: adresele MAC ale VIF sunt recunoscute automat de către codul de comutare (fără căutare în tabela de comutare) la primirea unui cadru destinat unei VIF și nu sunt învățate (nu sunt adăugate în tabela de comutare) la trimiterea unui cadru de către o VIF. Aceasta este o optimizare care scutește

⁵³ Prescurtarea vine de la *Virtual InterFace*.

trimiterea primului cadru destinat VIF către toate porturile în vederea învățării adresei MAC (vezi și pentru mai multe detalii).

Mecanismul descris profită din plin de avantajele oferite de abstractizarea `sk_buff`. Recepția unui cadru pe VIF se face extrem de simplu, fără nici un fel de simulare a întreruperilor hardware. Atunci când un `sk_buff` trebuie trimis către un port virtual, codul de comutare la nivel 2 pur și simplu se incrementează contoarele de recepție și se apelează funcția `netif_receive_skb()`⁵⁴, întocmai cum se întâmplă în driverul unei interfețe de rețea fizice atunci când hardware-ul a primit un cadru complet pe care l-a depus în memoria centrală. Această abordare este justificată având în vedere următoarele:

- Funcția `netif_receive_skb()` este reentrantă: singurul mecanism de sincronizare pe care îl folosește este RCU, sub forma `rcu_read_lock()`. Se știe că un al doilea apel `rcu_read_lock()` (fără ca primul să fi fost deblocat) nu are nici un efect.
- Dacă nu ar fi existat codul de comutare, iar adresa (adresele) de rețea de pe VIF ar fi fost asignate direct interfeței fizice, procesarea pachetului ar fi avut loc tot în contextul apelului `netif_receive_skb()` inițiat din driverul interfeței fizice. Prin urmare, apelul ar fi avut durată comparabilă.
- Un `net_device` care corespunde unei VIF nu poate fi adăugat niciodată ca port în switch. Hook-ul din `netif_receive_skb()` apelează codul de comutare de pachete doar dacă `net_device`-ul prin care a sosit pachetul este un port în switch. Prin urmare, nu există riscul reinvocării codului de comutare și intrării într-o buclă infinită.

Stiva de apeluri în cazul recepției de pachete pe VIF are următoarea structură:

<i>Funcție</i>	<i>Descriere</i>
<code>netif_receive_skb()</code>	Rutina Linux de procesare a cadrelor primite.
<code>sw_vif_rx()</code>	Rutina LISA de procesare a cadrelor primite pe o VIF.
<code>sw_vif_forward()</code>	Rutina LISA de comutare a unui cadru spre o VIF.
<code>sw_forward()</code>	Rutina LISA de comutare a cadrelor.
<code>sw_handle_frame()</code>	Rutina LISA de procesare a cadrelor primite.
<code>netif_receive_skb()</code>	Rutina Linux de procesare a cadrelor primite.
<code>driver.poll()</code> ⁵⁵	Metoda <code>poll()</code> a driverului.

Trimiterea unui pachet printr-o VIF este implementată la fel de simplu. Metoda `hard_start_xmit()` a driver-ului VIF actualizează contoarele de trimitere și apelează rutina de comutare de pachete (la nivel 2). Această abordare este justificată având în vedere următoarele:

- Codul de comutare la nivel 2 nu folosește pentru sincronizare decât RCU, sub forma `rcu_read_lock()`. Se știe că un al doilea apel `rcu_read_lock()` (fără ca primul să fi fost deblocat) nu are nici un efect.

⁵⁴ Desigur, pointerul către `net_device`-ul prin care a venit `sk_buff`-ul este actualizat corespunzător înainte de apel (astfel încât să indice spre `net_device`-ul asociat VIF).

⁵⁵ Este vorba de metoda `poll()` a driverului pentru interfața fizică prin care sosește cadrul. În cazul în care driverul folosește Softnet, este vorba de metoda `poll()` corespunzătoare dispozitivului backlog.

- Apelul codului de comutare are loc tot în context softirq, ca și în cazul recepției de cadre. Este drept că diferă softirq-ul (NET_TX_SOFTIRQ, spre deosebire de NET_RX_SOFTIRQ folosit pentru recepție), dar acest lucru nu ridică probleme.
- Dacă nu ar fi existat codul de comutare și cadrul ar fi fost trimis direct către un dispozitiv fizic, apelul făcut de coada de pachete Linux ar fi presupus depunerea cadrului în coada proprie a driverului. Pentru VIF, apelul presupune apelarea codului de comutare și redepunerea cadrului în coada de pachete Linux⁵⁶. Prin urmare, durata apelului este comparabilă în cele două cazuri, astfel că implementarea VIF nu aduce penalizări de performanță.
- Comutarea la nivel 2 se face întotdeauna în interiorul VLAN-ului, într-un VLAN nu poate exista decât cel mult un port virtual (corespunzător unei VIF), iar un cadru nu este niciodată comutat înapoi spre portul pe care a sosit. Prin urmare, nu există riscul reinvocării codului de comutare și intrării într-o buclă infinită.

Stiva de apeluri în cazul trimerii de pachete pe VIF are următoarea structură:

<i>Funcție</i>	<i>Descriere</i>
dev_queue_xmit()	Rutina Linux pentru adăugarea unui pachet în coada de trimitere.
sw_skb_xmit()	Rutina LISA inline de trimitere efectivă a unui cadru pe un port.
__sw_forward()	Rutina LISA de trimitere a cadrelor unicast ⁵⁷ .
sw_forward()	Rutina LISA de comutare a cadrelor.
driver.hard_start_xmit()	Rutina VIF de trimitere "fizică" a unui cadru.
qdisc_restart()	Rutina Linux pentru repornirea unei cozi de trimitere.
qdisc_run()	Rutina Linux (inline) pentru rularea unei cozi de trimitere.
net_tx_action()	Rutina Linux de tratare softirq pentru trimitere pachete.

4.4. Încapsularea 802.1q

Așa cum am arătat în secțiunea teoretică, încapsularea 802.1q presupune adăugarea sau eliminarea unui marcaj în antetul de nivel 2 al pachetului. Înainte de a putea trece efectiv la descrierea modalității prin care se realizează adăugarea și eliminarea marcajului, sunt necesare câteva precizări cu privire la socket buffers (sk_buff).

Având în vedere că antetul de nivel 2 se găsește în zona de date utile asociată sk_buff, implementarea standardului 802.1q presupune modificarea conținutului pachetului. Zona de date utile poate fi partajată între mai multe structuri sk_buff. În aceste condiții modificarea conținutului pachetului (care presupune *mutarea* de date în memorie) ar face ca celelalte structuri sk_buff să "vadă" date alterate la adresele indicate de pointerii lor. De altfel, regulile care asigură utilizarea corectă a sk_buff interzic modificarea în orice fel a zonei de date atunci când aceasta este partajată între mai multe structuri sk_buff.

Codul care realizează efectiv adăugarea sau eliminarea marcajului presupune că toate verificările au fost deja efectuate și modificarea pachetului se poate face în siguranță.

⁵⁶ Firește, redepunerea în coadă are loc după modificarea net_device-ului destinație. Acesta este cel coresponzător interfeței fizice asociate portului prin care va ieși cadrul.

⁵⁷ Spre deosebire de sw_forward(), această rutină este apelată atunci când adresa MAC destinație a fost găsită în tabela de comutare și cadrul va fi cu siguranță trimis spre un singur port.

Într-adevăr, aceste verificări (și acțiunile necesare în cazul în care rezultatul este negativ) au loc anterior, în codul de comutare a pachetelor. Mai multe detalii se pot găsi într-o subsecțiune ulterioară, care prezintă optimizările pentru broadcast.

Privind Figura 6, se poate observa că atât adăugarea cât și eliminarea marcajului 802.1q se pot face cu ușurință prin mutare de date în memorie. Cum antetul pachetului este în general mult mai mic față de date, este mai eficientă deplasarea datelor dinaintea marcajului, respectiv adresele MAC sursă și destinație.

La adăugarea marcajului cei 12 bytes corespunzători adreselor sunt deplasați cu 4 *înapoi*, lăsând astfel 4 bytes liberi între adresa sursă și câmpul "tip/lungime". Cei 4 bytes sunt completați ulterior cu marcajul 802.1q. Mutarea datelor se face cu un simplu apel `memmove()`. Adresa antetului de nivel 2 (câmpul `mac.raw` din `sk_buff`) este actualizată explicit, apoi este folosit apelul `skb_push` pentru a actualiza corespunzător restul câmpurilor (și a marca astfel micșorarea zonei `headroom`).

La eliminarea marcajului cei 12 bytes corespunzători adreselor sunt deplasați cu 4 *înainte*, suprascriind astfel cei 4 bytes ai marcajului 802.1q. Mutarea datelor se face cu un simplu apel `memmove()`. Adresa antetului de nivel 2 (câmpul `mac.raw` din `sk_buff`) este actualizată explicit, apoi este folosit apelul `skb_pull` pentru a actualiza corespunzător restul câmpurilor (și a marca astfel mărirea zonei `headroom`).

Un caz particular îl reprezintă cadrele generate chiar de sistemul pe care rulează LISA, atunci când acestea trebuie comutate către un port configurat în trunchi. Cum porturile virtuale corespunzătoare VIF sunt întotdeauna configurate în mod acces, cadrele generate nu vor avea marcaj 802.1q, iar acesta va trebui adăugat în vederea comutării spre un port în trunchi.

Testele pe care le-am efectuat au arătat că stiva TCP/IP Linux lasă minimum de `headroom` necesar atunci când generează cadre, respectiv 14 bytes⁵⁸ adică spațiul necesar adăugării antetului Ethernet. Funcțiile de alocare specifice `sk_buff` fac aliniere la dimensiunea cuvântului mașină. Pe platformele x86 pe care am testat, cuvântul este de 32 biți, prin urmare `headroom`-ul după aliniere este de 16 bytes. Dintre aceștia, 14 sunt folosiți pentru antetul Ethernet, rămânând liberi doar 2. Prin urmare, în `headroom` nu este spațiu suficient pentru adăugarea marcajului 802.1q.

Codul de adăugare a marcajului 802.1q face o verificare suplimentară pentru a asigura o comportare corectă în situația în care `headroom`-ul nu este suficient de mare. În această situație, `sk_buff` este modificat în vederea măririi `headroom`-ului folosind apelul `pskb_expand_head()`. Acesta lasă nemodificată adresa *structurii* `sk_buff`, în schimb realocă *zona de date utile* și actualizează corespunzător câmpurile din structura `sk_buff` (pointerii spre adrese din interiorul datelor utile). Trebuie precizat că *realocarea* zonei de date utile se face în siguranță, din moment ce structura `sk_buff` primită de rutina de adăugare a marcajului 802.1q era oricum exclusivă în vederea *modificării conținutului* zonei de date utile.

Este de asemenea de reținut că problema `headroom`-ului insuficient apare doar în cazul cadrelor *generate* de mașina care rulează LISA, nu și în cazul cadrelor *rutate* (subliniez că rutarea inter-VLAN presupune trecerea cadrului prin două VIF, una corespunzătoare

⁵⁸ Cei 14 bytes sunt distribuiți astfel: 6 bytes pentru MAC destinație, 6 bytes pentru MAC sursă, 2 bytes pentru câmpul *tip/lungime*.

VLAN-ului sursă și cealaltă corespunzătoare VLAN-ului destinație). Din fericire, driverele interfețelor fizice asigură headroom suficient, iar `sk_buff` nu este modificat (nu este realocat) în timpul trecerii prin diversele straturi ale stivei de rețea. Prin urmare, rutarea inter-VLAN se poate face fără realocări și copieri de date și astfel nu apar probleme de performanță.

În vederea optimizării, codul care asigură exclusivitatea structurii `sk_buff`, înainte de a o pasa spre rutina de adăugare a marcajului 802.1q, mărește headroom-ul cu 4 bytes în cazul în care este necesară o copie a zonei de date utile. Practic sunt sacrificați 4 bytes de memorie pentru a asigura ca headroom-ul să fie suficient dacă s-ar adăuga ulterior un marcaj 802.1q și nu se face o *a doua copie* în rutina de adăugare a marcajului. Mai multe detalii se pot găsi într-o subsecțiune ulterioară, care descrie optimizările pentru broadcast.

4.5. Algoritm de comutare la nivel 2

Am introdus această subsecțiune pentru că implementarea VLAN-urilor presupune modificări ale algoritmului clasic de comutare la nivel 2. În esență, este vorba despre verificări suplimentare, dar scopul meu este de a face o prezentare sumară a întregului algoritm, insistând pe aspectele specifice comutării în contextul VLAN-urilor.

Modificările cele mai importante apar la tabela de comutare și la codul pentru tratarea broadcast-urilor. Tabela de comutare trebuie să conțină în plus VLAN-ul pe care a fost învățată fiecare adresă MAC. Căutarea în tabelă se face de asemenea ținând cont de VLAN. Aceeași adresă MAC nu poate fi învățată pe două porturi diferite, dar acest lucru este valabil doar în cadrul aceluiași VLAN (nu există nici o restricție în cazul în care VLAN-ul diferă).

Codul de comutare la nivel 2 este distribuit în câteva rutine, împărțind astfel din punct de vedere logic funcționalitatea procesului de comutare. Voi descrie succint cele mai importante dintre aceste rutine.

Funcția `sw_handle_frame()` reprezintă punctul de intrare în LISA și este apelată de hook-ul din rutina Linux de tratare a cadrelor primite. Funcția face câteva verificări, apoi pasează controlul către `sw_forward()`. Sunt parcurși următorii pași:

- Se verifică existența în VDB a VLAN-ului prin care a venit cadrul.
- Se verifică faptul că modul de configurare a portului de intrare (acces sau trunchi) coincide cu tipul cadrului (fără sau cu marcaj 802.1q)⁵⁹.
- Pentru porturile în trunchi se verifică dacă VLAN-ul specificat în marcajul 802.1q este permis pe portul respectiv.
- Se verifică adresa MAC sursă. Aceasta nu trebuie să fie nulă (toți biții 0) sau broadcast (toți biții 1)⁶⁰.

59 Deși în implementarea actuală din LISA condiția este strictă, în practică există posibilitatea primirii de cadre fără marcaj 802.1q pe porturi configurate în trunchi. Se consideră că aceste cadre fac parte dintr-un VLAN implicit, care se configurează global sau pentru fiecare port în parte. Versiuni ulterioare de LISA ar putea să implementeze această facilitate și, implicit, toate opțiunile de configurare necesare.

60 Verificarea este extrem de importantă pentru cazul broadcast. Algoritm de comutare LISA nu tratează special cazurile cadrelor cu adresă *destinație* broadcast, ci se bazează pe faptul că un cadru a cărui adresă sursă nu se găsește în tabela de comutare este oricum trimis pe toate porturile și că adresa de broadcast nu va apărea niciodată în tabela de comutare. Cum adresa *sursă* a cadrelor este cea învățată, este important

- Este învățată adresa sursă a cadrului, folosind `fdb_learn()`.

Funcția `sw_forward()` reprezintă logica procesului de comutare. Acțiunile care trebuie întreprinse în diverse cazuri sunt implementate în funcții separate. Funcția parcurge următorii pași (fiecare punct reprezintă o ramură a algoritmului și presupune un caz de terminare a execuției):

- Dacă adresa MAC destinație este de tip multicast, se apelează `__sw_multicast()`. În cazul în care pachetul nu a fost comutat pe nici un port, se apelează `__sw_flood()`.
- Adresa MAC destinație este căutată în tabela de comutare folosind `fdb_lookup()`. Dacă este găsită, pachetul este comutat spre portul specificat în tabela de comutare, nu înainte însă de a verifica dacă sunt îndeplinite următoarele condiții:
 - Portul de ieșire este diferit de cel de intrare.
 - Dacă portul de ieșire este în mod acces, VLAN-ul cadrului coincide cu cel în care este configurat portul.
 - Dacă portul de ieșire este în mod trunchi, VLAN-ul cadrului se găsește printre VLAN-urile permise pe portul de ieșire.
- Cadrul este trimis spre toate porturile din VLAN-ul din care face parte.

Funcția `__sw_flood()` este folosită pentru a trimite un cadru către toate porturile dintr-un VLAN. O optimizare evidentă este folosirea listelor de porturi din VDB (despre care am vorbit în subsecțiunea dedicată VDB). Astfel nu sunt necesare căutări sau testări pentru a determina porturile de ieșire. O altă problemă este cea a efectuării unui număr minim de clonări/copieri de `sk_buff` în vederea asigurării unei copii exclusive atunci când este necesară modificarea datelor utile. Optimizările implementate în acest sens sunt descrise pe larg în subsecțiunea următoare.

Funcția `__sw_multicast()` este folosită pentru a trimite un cadru către toate porturile care au asociată o adresă MAC multicast în tabela de comutare⁶¹. Algoritmul este foarte asemănător cu cel din `__sw_flood()`. Diferența principală o constituie modul în care este obținută lista porturilor de ieșire. Dacă în cazul `__sw_flood()` lista exista în formă explicită în VDB, aici este necesară o parcurgere și o filtrare. Eficiența algoritmului se bazează pe faptul că tabela de comutare este organizată sub forma unei tabele de dispersie (hash) după adresa MAC și că astfel toate înregistrările care conțin o anumită adresă MAC se găsesc în același bucket. Prin urmare, este suficientă parcurgerea listei de înregistrări corespunzătoare unui singur bucket. Mai mult, puterea de dispersie a funcției de hash asigură că în bucket-ul în care se face căutarea nu există prea multe înregistrări irelevante (în cazul uzual, în care nu există coliziuni de distribuție și aceeași adresă MAC multicast nu apare în mai multe VLAN-uri, în bucket-ul în care se face căutarea există *doar*

ca aceasta să nu fie broadcast pentru a asigura funcționarea corectă.

61 Reamintesc faptul că adresele MAC multicast nu sunt niciodată învățate (învățarea se face după adresele sursă ale cadrelor, iar o adresă MAC multicast nu ar trebui să apară niciodată ca adresă sursă), ci sunt adăugate static în tabela de comutare. Există însă protocoale (cum ar fi IGMP) prin care o stație poate informa routerele că dorește să primească traficul destinat unui anumit grup multicast. Un switch inteligent poate limita traficul de multicast doar la porturile corespunzătoare prin interpretarea cadrelor IGMP și adăugarea automată de adrese MAC multicast statice în tabela de comutare. O astfel de funcționalitate este denumită *IGMP Snooping*. Momentan LISA nu implementează IGMP Snooping. Mai multe detalii pot fi găsite în [Cisco02].

înregistrări relevante).

Pentru a asigura funcționarea corectă, în funcția `__sw_multicast()` sunt necesare verificări suplimentare față de `__sw_flood()` atunci când este parcursă lista. Aceste verificări reprezintă criteriile de filtrare despre care vorbeam. Ordinea testelor a fost aleasă cu atenție, astfel încât algoritmul să fie cât mai eficient în cazul în care în bucket există și înregistrări irelevante. Lista testelor (în ordinea în care se efectuează) este următoarea:

- VLAN-ul înregistrării coincide cu al cadrului;
- portul din înregistrare nu este același cu portul de intrare;
- adresa MAC din înregistrare coincide cu adresa destinație a cadrului;
- dacă portul din înregistrare este configurat în mod acces, VLAN-ul cadrului coincide cu cel în care este configurat portul;
- dacă portul din înregistrare este configurat în mod trunchi, VLAN-ul cadrului se găsește printre VLAN-urile permise ale portului.

Funcția `fdb_lookup()` realizează căutarea unei înregistrări după adresa MAC în tabela de comutare. Am arătat deja că tabela de comutare este un hash; `fdb_lookup()` nu face altceva decât să aplice funcția de dispersie și să caute liniar înregistrarea în bucket-ul obținut.

Funcția `fdb_learn()` implementează învățarea adreselor MAC (adăugarea dinamică în tabela de comutare). Modificarea tabelii de comutare ridică probleme de sincronizare atât cu procesul de comutare cât și cu interfața cu spațiul utilizator. Acestea sunt descrise pe larg în subsecțiunea dedicată sincronizărilor, iar aici mă voi limita la a descrie logica învățării de adrese.

Fiecare înregistrare dinamică are asociate timpul ultimei actualizări și un cronometru. La adăugarea înregistrărilor cronometrul este inițializat la intervalul de învechire a adreselor MAC⁶², iar la expirarea cronometrului înregistrarea este automat ștearsă din tabela de comutare.

Funcția `fdb_learn()` este apelată la fiecare cadru primit. În majoritatea cazurilor înregistrarea există deja. În aceste condiții, înregistrarea este doar adusă la zi: timpul ultimei actualizări este setat la timpul curent⁶³.

Există două cazuri speciale, care reprezintă excepții la ceea ce am descris până acum:

- Există deja o înregistrare statică având aceeași adresă și același VLAN. În acest caz algoritmul se termină imediat, fără a face modificări în tabela de comutare.
- Există deja o înregistrare dinamică având aceeași adresă și același VLAN, dar un port diferit. Această situație poate fi replicată destul de ușor în practică, de exemplu dacă o stație este mutată fizic dintr-un port în altul. În acest caz, pe lângă actualizările descrise anterior, este modificat și portul din înregistrare. Motivul este foarte simplu: în condiții normale, pachetele generate de aceeași stație nu pot sosi în switch prin porturi diferite

62 Acest interval este o opțiune configurabilă global sau la nivel de VLAN. Momentan LISA implementează doar configurarea globală a opțiunii.

63 Cronometrul nu este resetat la intervalul de învechire. În schimb, la expirarea cronometrului, acesta verifică dacă de la ultima actualizare a înregistrării a trecut un interval mai mare decât cel de învechire înainte de a șterge înregistrarea. În cazul în care timpul care a trecut este mai mic, cronometrul se rearmează singur, calculând timpul rămas până la învechirea (expirarea) înregistrării.

(excepțiile au fost deja discutate în secțiunea teoretică).

4.6. Optimizări pentru broadcast și multicast

În cazul transmiterii aceluiași cadru către mai multe porturi (fie că este vorba de broadcast sau multicast) apar probleme de performanță legate de marcajul 802.1q și de faptul că acesta presupune modificarea pachetului, care la rândul ei, presupune existența unei copii exclusive.

Chiar în cazul în care pachetul nu este modificat, este necesară *clonarea* `sk_buff` pentru fiecare port prin care este trimis. Aceasta pentru că, o dată adăugat în coada dispozitivului hardware, `sk_buff` va fi eliberat după trimitere. Clonarea presupune doar copierea structurii `sk_buff` și partajarea zonei de date utile între cele două clone. În cazul în care nu s-ar face clonarea, ar apărea o cursă critică: după ce primul dispozitiv termină de trimis, pachetul este eliberat și atât zona de memorie asociată structurii `sk_buff` cât și cea asociată datelor utile ar putea fi refolosite până când celelalte dispozitive ajung să trimită efectiv cadrele.

Asigurarea unei copii exclusive în vederea implementării standardului 802.1q și chiar modificarea pachetului (adăugarea sau ștergerea marcajului) sunt operații foarte mari consumatoare de timp și cu impact puternic asupra performanței. Prin urmare, algoritmi folosiți trebuie să asigure ca aceste operații să fie efectuate doar dacă este nevoie și ca, în acest caz, să nu fie efectuate de mai multe ori. Într-adevăr, o dată ce un pachet a fost modificat (de exemplu prin adăugare de marcaj), el poate fi trimis către toate porturile în mod trunchi realizând ulterior doar clonări.

Rezultă că una dintre cele mai importante condiții pentru optimizare este ordonarea trimiterii pachetelor după modul de configurare a porturilor destinație (acces sau trunchi). Ordinea logică în care trebuie să decurgă trimiterea este următoarea:

- se trimite cadrul către toate porturile care au același mod de configurare ca și portul de intrare;
- se modifică pachetul, prin adăugarea sau eliminarea marcajului 802.1q, după caz;
- se trimite cadrul către toate porturile care au mod de configurare opus celui al portului de intrare.

În cazul broadcast este foarte utilă menținerea în VDB a două liste separate pentru porturile configurate în mod acces și în mod trunchi. Astfel, modelul descris mai sus poate fi aplicat foarte ușor, fără iterații inutile. Mai mult, ideea este aceeași indiferent că se face adăugare sau eliminare de marcaj. Diferă doar funcția de modificare a pachetului (adăugare sau eliminare de marcaj) și listele care sunt procesate înainte și respectiv după modificare. În acest sens, cei trei pași descriși mai sus au fost abstractizați în funcția `sw_flood()`. Aceasta verifică tipul portului de intrare și apelează funcția `__sw_flood()` (care implementează efectiv algoritmul) cu parametrii corespunzători (capetele celor două liste și funcția de modificare a pachetului sunt pasate ca pointeri).

Algoritmul pentru multicast este extrem de asemănător (de fapt a fost elaborat prin adaptarea algoritmului deja implementat pentru broadcast). Diferența este că aici lista porturilor nu mai este explicit împărțită după modul de configurare (acces sau trunchi), așa cum am arătat în subsecțiunea anterioară. Pentru a putea folosi același algoritm sunt necesare *două* parcurgeri ale listei de înregistrări corespunzătoare bucket-ului din tabela de

comutare. Acest neajuns a fost parțial eliminat prin poziționarea condiției de potrivire a modului de configurare a portului chiar la începutul listei criteriilor de filtrare.

În rest implementarea pentru broadcast și multicast este identică și în continuare mă voi concentra asupra modului în care este abordată problema clonărilor și copierilor în număr minim.

O primă observație este aceea că atunci când un cadru trebuie trimis pe n porturi, sunt necesare doar $n - 1$ clonări. Într-adevăr, funcția `sw_handle_frame()` primește întotdeauna o structură `sk_buff` separată⁶⁴, pe care este obligată să o elibereze explicit în cazul în care nu va fi eliberată implicit (de exemplu prin depunerea ei în coada de trimitere a unui dispozitiv rețea). Prin urmare, dacă nu există decât un singur port de ieșire nu este necesară nici o clonare, iar structura `sk_buff` va fi eliberată implicit de către driverul dispozitivului de rețea după realizarea transmisiei. Dacă există două porturi de ieșire este necesară doar o singură clonare, pentru a evita apariția unei curse critice (pe care am descris-o mai sus) la trimiterea pe cel de-al doilea port.

Pentru că numărul de elemente din liste nu este cunoscut dinainte (mai ales în cazul multicast, unde se face filtrarea elementelor) și pentru că o parcurgere suplimentară pentru numărare este extrem de ineficientă, am folosit o metodă pe care am numit-o *post-procesarea elementelor*⁶⁵. Această metodă constă în amânarea procesării unui element din listă până la întâlnirea unui nou element. Abia când este întâlnit un nou element în listă se realizează o clonă, elementul precedent este procesat, apoi i se atribuie adresa clonei. După terminarea parcurgerii este necesară o verificare suplimentară pentru a procesa ultimul element din listă în cazul în care aceasta nu a fost vidă.

O descriere simplificată, în pseudocod, a metodei post-procesării elementelor are următoarea structură:

```

/* se consideră că în skb se găsește copia
   separată pe care am primit-o la intrare
   */
prev = NULL;
foreach (elem ∈ listă) {
    if (prev != NULL) {
        tmp = skb_clone(skb);
        procesează_și_eliberează(skb, prev);
        skb = tmp;
    }
    prev = elem;
}
if (prev != NULL) {
    procesează_și_eliberează(skb, prev);
} else {
    dev_kfree_skb(skb); /* eliberare explicită */
}

```

64 Este foarte important de reținut că, deși structura `sk_buff` este separată (are contorul de utilizare egal cu 1), zona de date utile ar putea fi partajată cu o altă structură `sk_buff`. Prin urmare, trebuie făcute verificări suplimentare în cazul modificării pachetului.

65 Aceeași metodă este folosită și de funcția Linux de tratare a cadrelor primite, `netif_receive_skb()`, acolo unde pentru fiecare cadru se apelează funcții de tratare (generice sau la nivelurile superioare ale stivei de rețea) al căror număr nu este cunoscut anterior. Similar codului LISA de comutare a pachetelor, aceste funcții de tratare primesc o copie separată a structurii `sk_buff`, pe care sunt obligate să o elibereze (fie implicit fie explicit). Din păcate, codul respectiv este foarte sărac în comentarii și am putut să-l înțeleg în totalitate abia după ce eu însumi am ajuns la aceeași soluție pentru a realiza doar $n - 1$ copii ale structurii `sk_buff`.

```
}

```

Din păcate, lucrurile sunt mai complicate în contextul modificării pachetelor. Copierea datelor utile este necesară doar dacă în prima listă există cel puțin un element sau dacă zona de date este partajată⁶⁶. O altă observație este aceea că o copiere a `sk_buff` asigură implicit și o clonare, prin urmare după copiere nu este necesară o clonare suplimentară (prima clonare *după* copiere are loc abia dacă în a doua listă există mai mult de un element).

În funcție de numărul de elemente din fiecare listă, am identificat 9 cazuri de bază. Orice alt caz presupune iterații suplimentare la parcurgerea uneia dintre liste (sau la ambele) și nu prezintă interes. Menționez că algoritmul pe care l-am elaborat se comportă corect în toate aceste cazuri și, în plus, asigură numărul minim de clonări și/sau copieri. Tabelul următor prezintă cele 9 cazuri și numărul clonărilor și copierilor. Primele două coloane reprezintă numărul de elemente din prima, respectiv a doua listă.

N_1	N_2	<i>Clonări înaintea modificării</i>	<i>Copiere</i>	<i>Modificarea pachetului</i>	<i>Clonări după modificare</i>
0	0	0	X	X	0
1	0	0	X	X	0
≥ 2	0	$N_1 - 1$	X	X	0
0	1	0	X	V	0
1	1	0	V	V	0
≥ 2	1	$N_1 - 1$	V	V	0
0	≥ 2	0	X	V	$N_2 - 1$
1	≥ 2	0	V	V	$N_2 - 1$
≥ 2	≥ 2	$N_1 - 1$	V	V	$N_2 - 1$

Observații:

- În cazul $N_1 = N_2 = 0$, `sk_buff` este eliberat explicit la sfârșitul algoritmului (nu și în cazul multicast, unde o structură `sk_buff` încă mai este necesară în vederea transmiterii pe toate porturile).
- În cazurile $N_1 = 0, N_2 = 1$ și $N_1 = 0, N_2 \geq 2$ are loc totuși copierea în situația în care datele utile ale pachetului sunt partajate. Acest lucru se întâmplă de obicei atunci când structura `sk_buff` a fost deja clonată înainte de a fi executat codul LISA de comutare a

⁶⁶ Menționez că în cazul în care nu există rutine de tratare de pachete generice (cazul uzual) și, în plus, în prima listă nu există elemente (porturi), nu este necesară copierea datelor utile. Zona de date utile nu este partajată și codul de comutare de pachete nu are nevoie de varianta nemodificată a pachetelor, prin urmare modificarea se poate realiza direct.

pachetelor⁶⁷. Dacă nu s-ar realiza copierea, ar putea să apară o cursă critică⁶⁸.

- În cazul în care are loc copierea, sunt sacrificați 4 bytes de memorie pentru a mări headroom-ul. Cu prețul celor 4 bytes (care de multe ori nu vor fi utilizați) se asigură ca în cazul adăugării ulterioare a unui marcaj 802.1q să nu aibă loc o *a doua copiere* a datelor utile (vezi comentariile legate de adăugarea marcajului 802.1q din subsecțiunea dedicată implementării acestui standard).

4.7. Sincronizări

Deși proiectat având în minte utilizarea pe sisteme integrate și plăci specializate (care cel mai adesea sunt uniprocessor), codul LISA poate fi rulat fără probleme pe sisteme SMP. Toate problemele de sincronizare au fost tratate cu atenție și nici una dintre sincronizări nu se bazează pe existența unui singur procesor în sistem.

Pe parcursul acestei subsecțiuni voi prezentare cele mai dificile probleme de sincronizare care au apărut la dezvoltarea proiectului, precum și modul în care acestea au fost soluționate.

Majoritatea sincronizărilor prezentate se bazează pe RCU. O scurtă descriere a principiilor de sincronizare specifice RCU poate fi găsită chiar în această lucrare, în Anexa A.

Sincronizarea VDB

Structurile de date ale VDB sunt accesate atât de procesele utilizator prin intermediul apelurilor `ioctl()`, cât și de procesul de comutare. Am explicat deja că procesul de comutare este optimizat pentru broadcast folosind câte două liste înlănțuite pentru fiecare VLAN: una a porturilor în mod acces și alta a porturilor în mod trunchi care fac parte din VLAN-ul respectiv. Porturile nu pot fi înlănțuite direct pentru că același port poate face parte din oricâte VLAN-uri (dacă este configurat în trunchi) și prin urmare trebuie înlănțuit în oricâte liste.

Listele de porturi dintr-un VLAN sunt sincronizate folosind variantele RCU ale macrourilor pentru manipularea listelor înlănțuite. Structurile `net_switch_vdb_link` (vezi Figura 15) folosite pentru a rezolva înlănțuirea unui port în mai multe liste ridică însă probleme de sincronizare.

Sincronizarea la adăugare este asigurată publicând structura `net_switch_vdb_link` nou creată (adăugând-o în listă) abia după inițializarea câmpului `port` (pointer la structura

67 Mai exact, atunci când se folosesc programe de captură a pachetelor pe dispozitivul fizic corespunzător portului de intrare (cum ar fi `tcpdump` sau, mai general, tot ce folosește `libpcap`), se înregistrează în kernel o funcție generică de tratare a pachetelor. Funcțiile generice de tratare sunt apelate de către rutina Linux de procesare a pachetelor primite, `netif_receive_skb()`, înainte să fie apelat hook-ul LISA. Așa cum am mai spus, funcțiile generice de tratare primesc o structură `sk_buff` separată (adică are loc o clonare) și atunci zona de date devine partajată.

68 Procesarea pachetelor primite are loc în context `softirq`, care nu poate fi întrerupt decât de către o întrerupere hardware (în nici un caz de către scheduler pentru a comuta în context proces). În cazul unui program de captură a pachetelor, se realizează o clonă a `sk_buff` și zona de date utile devine partajată. Procesarea conținutului pachetului de către programul de captură va avea loc în context proces (mai exact în spațiu utilizator) abia după terminarea executării rutinei de procesare a pachetelor primite (care rulează în context `softirq`). Între timp rulează codul LISA de comutare a pachetelor, care mută date în zona utilă a pachetului. Când ajunge să fie procesat pachetul de către programul de captură, la adresele indicate de propria structură `sk_buff` se vor "vedea" date alterate.

corespunzătoare unui port) al acesteia. Un apel `netif_wmb()` este folosit înainte de publicare pentru a asigura consistența în cazul reordonării instrucțiunilor.

La scoaterea unui port din VLAN lucrurile se complică puțin. Deși listele sunt sincronizate, un proces ar putea dealoca structura `net_switch_vdb_link` imediat după ce codul de broadcast (rulând pe alt procesor⁶⁹) a extras din listă o referință către ea. Pentru a preveni această situație, codul de broadcast rulează în regiune critică RCU. La nivelul codului de scoatere a portului din VLAN (care rulează întotdeauna în context proces și în afara unei regiuni critice) se scoate întâi structura din listă, apoi se așteaptă (folosind apelul `synchronize_kernel()`) trecerea procesoarelor printr-o perioadă de latență înainte de a dealoca structura. În acest fel se asigură că în eventualitatea în care codul de comutare a apucat să "vadă" structura în listă, acesta se termină înainte ca structura să fie efectiv dealocată.

Adăugarea unui VLAN este doar un caz particular al adăugării unui port într-un VLAN. Deși pointerul la structura `net_switch_vdb_entry` nou creată este publicat în vectorul de VLAN-uri înainte de a adăuga eventualele porturi, sincronizarea se face pentru fiecare adăugare de port în parte după modelul descris anterior⁷⁰.

Ștergerea unui VLAN ridică o problemă în plus: trebuie șterse toate adresele MAC învățate pe VLAN-ul respectiv. Acestea nu pot fi șterse imediat, pentru că atât timp cât VLAN-ul încă există în VDB, procesul de comutare poate învăța MAC-uri noi. Aici problema apare și pe sistemele uniprocessor, deoarece codul de ștergere a VLAN-ului (care rulează în context proces) ar putea fi întrerupt de un softirq care să ruleze codul de comutare.

În realitate tot codul de comutare rulează într-o regiune critică⁷¹, nu numai cel de broadcast de care am vorbit puțin înainte. La ștergerea unui VLAN, acesta este întâi scos din vectorul de VLAN-uri (fără a fi dealocat), apoi se așteaptă trecerea procesoarelor prin starea de latență. Abia ulterior are loc "curățarea" VLAN-ului, care presupune ștergerea tuturor adreselor MAC asociate, scoaterea tuturor porturilor din VLAN și în cele din urmă dealocarea structurii `net_switch_vdb_entry`. Trebuie menționat că, o dată ce VLAN-ul a fost scos din vector și toate procesoarele au trecut prin starea de latență, eliberarea structurilor `net_switch_vdb_link` se poate face imediat (fără sincronizări) pentru că invocări ulterioare ale codului de comutare nu vor mai vedea VLAN-ul (și deci nu îi vor mai parcurge listele).

Sincronizarea FDB

Sincronizarea bazei de date de comutare este cea mai dificilă, pentru că aceasta este citită și poate fi chiar modificată din trei contexte diferite:

- codul de comutare: pentru fiecare cadru primit are loc atât actualizarea (prin procesul de

69 Codul de broadcast rulează în zonă critică RCU. Prin urmare nu poate fi întrerupt pentru a planifica un proces utilizator, deci singurul caz posibil este ca acesta din urmă să ruleze pe un alt procesor.

70 Un port configurat în mod acces nu poate face parte dintr-un VLAN care nu există în VDB. Totuși, pentru porturile în mod trunchi se poate configura accesul la VLAN-uri care nu există în VDB (pachetele nu vor fi însă comutate pe aceste VLAN-uri). La crearea unui VLAN toate porturile în mod trunchi care aveau deja acces la el trebuie adăugate în lista de porturi din VDB pentru a asigura funcționarea corectă a broadcast-urilor.

71 În cazul invocării codului chiar de către rutina de tratare Linux a pachetelor primite (și nu de către trimiterea unui pachet pe o VIF), intrarea în regiunea critică RCU are loc chiar dinainte, în funcția `netif_receive_skb()`.

învățare a adreselor MAC) cât și interogarea bazei de date de comutare (pentru a stabili portul/porturile de ieșire).

- spațiul utilizator: programele de configurare și control pot șterge adrese MAC, pot adăuga adrese statice și pot lista (eventual folosind o filtrare) conținutul tabelii de comutare.
- expirarea cronometrelor: adresele MAC învățate dinamic expiră (și trebuie șterse) dacă nu au fost actualizate într-un anumit interval (configurabil).

Deși toate cele trei contexte enumerate pot face modificări ale FDB, citirile sunt mult mai frecvente decât scrierile, astfel că principiile RCU de sincronizare sunt aplicabile. Totuși, pentru că scrierea este posibilă din mai multe contexte, trebuie folosit și un mecanism clasic de sincronizare. După o analiză atentă, soluția a fost utilizarea unui spinlock la nivelul fiecărui bucket al tabelii de dispersie a FDB.

Spațiul utilizator și expirarea cronometrelor nu sunt critice din punct de vedere al timpului, în sensul că scrierile acestora în FDB sunt ocazionale și, chiar în situația în care ar trebui să aștepte pentru acapararea spinlock-ului, nu apar probleme de performanță. Problemele de performanță ar putea să apară în schimb în cazul mai multor instanțe ale algoritmului de comutare care rulează în paralel pe procesoare diferite. Realizarea zăvorârii la nivelul unui bucket împreună cu puterea de dispersie a funcției hash asigură performanțe ridicate chiar și în situația pe care am descris-o: este probabil ca instanțe diferite ale algoritmului de comutare să aibă nevoie simultan să scrie în FDB, dar este foarte puțin probabil să aibă nevoie să modifice *același bucket*⁷².

Problema cea mai mare a învățării de adrese o constituie trecerea din modul de citire în modul de scriere. Pentru că între scrieri și citiri nu există efectiv o sincronizare (RCU nu este o sincronizare, este doar o modalitate de a asigura consistența datelor), tabela de comutare se poate modifica (din exterior) în timpul rulării algoritmului de comutare. Pentru decizia de comutare aceasta nu este o problemă⁷³. Problemele apar însă în cazul învățării adreselor. Regula este "dacă adresa MAC nu există, creează o înregistrare nouă". Cum scrierile și citirile nu sunt sincronizate, între momentul în care s-a constatat că adresa nu există și momentul adăugării adresa ar putea fi adăugată din exterior⁷⁴.

72 Analiza poate fi dusă chiar mai departe. Instanțe simultane ale algoritmului de comutare presupun procesarea simultană a mai multor pachete pe procesoare diferite. Pe sistemele SMP este folosită afinitatea interfețelor de rețea la procesoare (toate pachetele sosite pe o interfață sunt prelucrate de același procesor). Prin urmare instanțele simultane ale algoritmului vor prelucra pachete venite pe porturi diferite. Având în vedere că în general pachete cu aceeași adresă MAC sursă nu pot sosi pe porturi diferite și că scrierea în FDB se face după adresa MAC sursă, șansele ca două instanțe diferite ale algoritmului de comutare (care rulează simultan) să încerce să actualizeze înregistrări cu aceeași adresă MAC sunt extrem de mici. Dacă puterea de dispersie a funcției hash este mare, se va întâmpla foarte rar ca instanțele simultane ale algoritmului să încerce să facă modificări în același bucket și deci să se aștepte între ele.

73 Dacă algoritmul a apucat să "vadă" o înregistrare din FDB și aceasta este imediat ștersă, structura va fi încă disponibilă (datorită RCU) și cadrul va fi comutat pe portul din înregistrare. Dacă algoritmul apucă să "vadă" că nu există o înregistrare și aceasta este imediat adăugată cadrul va fi trimis pe toate porturile fără nici o problemă.

74 Pe sistemele uniprocessor acest lucru nu se va întâmpla niciodată, pentru că nu rulează simultan mai multe instanțe ale algoritmului de comutare și algoritmul de comutare nu poate fi întrerupt pentru execuția codului în spațiu utilizator (singurul care ar mai putea adăuga o adresă MAC). Așa cum am arătat deja, pe sistemele SMP învățarea aceluiași MAC în două instanțe simultane ale algoritmului de comutare este puțin probabilă (cu excepția buclelor în topologie, când devine foarte probabilă deoarece cadrele sunt

Pentru a preîntâmpina problemele descrise, soluția a fost utilizarea unui așa-numit *model tranzacțional*. Acesta presupune o verificare suplimentară înainte de a face efectiv adăugarea și presupune mai mulți pași:

- verifică dacă adresa există deja; în caz că există, termină execuția;
- adresa nu există; zăvorăște pentru scriere bucket-ul corespunzător;
- verifică dacă adresa există deja; în caz că există, eliberează zăvorul și termină execuția;
- adaugă înregistrarea;
- eliberează zăvorul.

Acest model elimină problema sincronizării. Făcând o a doua verificare în interiorul regiunii critice spinlock, se asigură că aceeași înregistrare nu este adăugată de două ori. Realizarea unei a doua verificări nu ridică probleme de performanță, deoarece în majoritatea cazurilor adresa există deja și algoritmul se termină după prima verificare.

Modelul tranzacțional împreună cu sincronizarea RCU sunt folosite pentru toate cazurile de adăugare de înregistrări în tabela de comutare. Ștergerea înregistrărilor se face de asemenea după modelul tranzacțional, pentru a preîntâmpina situația în care o înregistrare este ștearsă din afară între momentul identificării ei și momentul ștergerii efective⁷⁵.

O altă problemă de sincronizare legată de FDB este cea a listării din spațiu utilizator a înregistrărilor. Tabela de comutare se poate modifica oricând, prin urmare este necesară parcurgerea acesteia într-o regiune critică RCU. Pe de altă parte numărul rezultatelor la listare nu poate fi cunoscut dinainte (chiar dacă înregistrările ar fi numărate, acestea pot fi șterse sau se pot adăuga unele noi în timpul parcurgerii). Citirea dintr-un dispozitiv caracter nu este o soluție pentru că, în cazul în care bufferul în care se citește nu este suficient de mare se iese din mod kernel (și implicit din regiunea critică RCU). La un apel ulterior al funcției read() nu se poate asigura consistența cu datele citite până atunci (nu este posibilă reluarea parcurgerii tabelii din același punct pentru că aceasta poate fi modificată între apelurile de citire). Citirea dintr-un fișier special din /proc nu este o soluție pentru că este limitată la dimensiunea unei pagini.

Singura soluție rămâne listarea prin intermediul unui apel ioctl(). Pentru a rezolva problema dimensiunii zonei de memorie în spațiu utilizator se folosește tot un model tranzacțional. Se pornește cu o dimensiune "rezonabilă" a bufferului și se încearcă listarea tabelii de comutare. Codul în mod kernel rulează în regiune critică RCU (asigurând astfel consistența datelor) și scrie în buffer până la epuizarea înregistrărilor sau până când acesta se umple. Valoarea returnată către apelul din spațiu utilizator indică motivul terminării algoritmului. În cazul în care au fost epuizate toate înregistrările, listarea s-a terminat cu succes. În caz contrar se realocă bufferul cu o dimensiune mai mare și se face un nou apel ioctl(). Iterația are loc până când bufferul este suficient de mare încât să cuprindă toate înregistrările.

Deși nu foarte eficientă, soluția este acceptabilă din mai multe motive:

- pentru a realiza o citire "atomică" oricum este necesar un buffer suficient de mare încât

multiplicate și copiile ajung aproape simultan). Totuși este posibilă apariția curselor critice între algoritmul de comutare și contextul utilizator.

75 Este de fapt cazul ștergerii tuturor înregistrărilor corespunzătoare unui VLAN, pentru că la ștergerea unei singure înregistrări (din context utilizator) căutarea ei se poate face direct în regiune critică spinlock.

să cuprindă toate înregistrările (indiferent dacă este alocat incremental sau nu);

- listarea tabelii de comutare din spațiu utilizator nu este critică din punct de vedere al timpului;
- folosind doar o regiune critică RCU, listarea din spațiu utilizator nu introduce stări de așteptare în alte fire de execuție (cum ar fi procesul de comutare care este critic din punct de vedere al timpului).

În LISA mai există și alte sincronizări decât cele prezentate mai sus. Le-am ales însă pe cele mai complicate, restul fiind doar niște cazuri particulare ale celor prezentate sau aplicații directe ale principiilor de sincronizare RCU. Ar mai fi de menționat că spinlock-urile din FDB sunt singurele mecanisme clasice de sincronizare folosite în prezent în LISA. Toate celelalte probleme de sincronizare au fost rezolvate cu RCU.

5. Performanțe și testare

Așa cum am arătat în secțiunile anterioare, algoritmi utilizați în LISA au fost optimizați pe cât posibil. Complexitatea foarte mare a codului implicat în procesul de comutare nu permite o analiză amănunțită a performanțelor. Este posibilă doar o *estimare* a acestora, analizând separat o parte din algoritmi. Pentru a putea oferi o imagine clară asupra comportării unui dispozitiv LISA, am recurs la o serie de teste, menite să ofere o imagine reală a performanțelor implementării.

În continuare, voi descrie pe scurt echipamentele utilizate pentru testare, configurația lor, precum și rezultatele obținute.

Unele dintre cele mai mari probleme ale LISA sunt procesarea în software a pachetelor și limitarea impusă de lărgimea de bandă a magistralei sistemului. Cum algoritmi de comutare sunt aplicați pentru fiecare pachet în parte, este probabil ca problemele să apară la o rată mare a pachetelor primite, și nu neapărat atunci când se transferă o cantitate mai mare de date (dar cu pachete de dimensiuni mari).

Prin urmare, primele teste efectuate au fost cele care implică o rată mare a pachetelor de intrare. Am ales pentru testare cadre de 64 bytes (lungime totală, inclusiv antetul de nivel 2), pe care le-am generat folosind `pktgen` – modulul pentru generarea pachetelor inclus în distribuția standard a Linux kernel⁷⁶. Modulul, extrem de flexibil, permite generarea de cadre specificând adresele sursă și destinație (atât la nivel 2 cât și la nivel 3), porturile sau chiar domenii ale adreselor și/sau porturilor. Rata pachetelor generate poate fi controlată specificând o întârziere (o temporizare) adăugată la transmiterea fiecărui pachet, cu precizie de nanosecunde. Rata pachetelor este practic limitată doar de performanțe hardware-ului utilizat.

Cunoscând viteza de transfer a interfeței de ieșire, v_{trnas} [Mb/s], dimensiunea pachetelor len [B], și rata pachetelor generate, r_{out} , se poate calcula temporizarea necesară, t_d :



În general al doilea termen este cu câteva ordine de mărime mai mic decât primul și poate fi neglijat.

⁷⁶ Mai multe detalii pot fi găsite chiar în documentația inclusă în distribuția de kernel, în fișierul `Documentation/networking/pktgen.txt`.

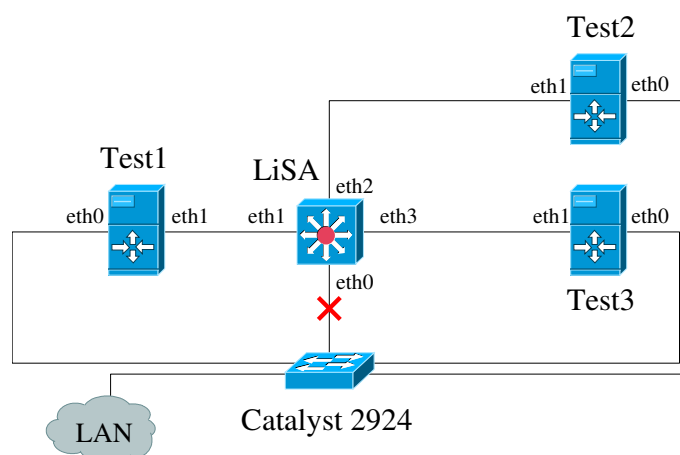


Figura 16 □ Configurația pentru măsurarea ratei pachetelor comutate

Figura 16 ilustrează configurația utilizată pentru măsurarea ratei pachetelor comutate⁷⁷. Interfețele eth0 ale sistemelor de test au fost folosite pentru conectarea în rețeaua locală, iar interfețele eth1 au fost conectate în rețeaua de test (izolată de rețeaua locală prin întreruperea legăturii de pe eth0 a sistemului LISA). Sistemul Test1 a fost folosit pentru a genera pachete, iar sistemele Test2 și Test3 pentru a analiza pachetele comutate de către LISA⁷⁸.

Cunoscând numărul de pachete primite de către mașinile de test, se poate calcula rata de ieșire a pachetelor. Acest lucru este posibil pornind de la două ipoteze:

- Mașinile de test primesc toate pachetele emise de către LISA. Pierderea pachetelor este puțin probabilă deoarece interfețele sistemului LISA sunt direct conectate cu ale sistemelor de test. În aceste condiții, apariția erorilor la transmisie/recepție este, de asemenea, foarte puțin probabilă.
- Timpul total în care LISA primește pachete este egal cu timpul total în care transmite aceleași pachete. Având în vedere modul în care funcționează recepția pachetelor (implementarea NAPI), pachetele încep să fie procesate imediat, iar întârzierea cu care acestea părăsesc sistemul este neglijabilă față de durata totală a transmisiei.

Astfel, exprimând timpul total de transmisie, t_s , în două modalități diferite, se poate obține expresia ratei pachetelor transmise, r_{out} , în funcție de rata pachetelor primite, r_{inp} , numărul de pachete primite (generate), N_{inp} , și numărul de pachete comutate (numărate pe mașinile de test), N_{out} :



Pentru testare am ales valori ale ratei pachetelor generate cuprinse între aproximativ 40000

⁷⁷ *Test1*, *Test2* și *Test3* sunt sisteme Dual Xeon / 2.8 GHz, 2MB RAM, chipset Interl 6300ESB, echipate cu două chip-uri de rețea BCM5721 (Broadcom Tigon 3). LISA este un sistem LE-564 produs de firma Commell Systems, <http://www.commell-sys.com/Product/SBC/LE-564.htm>.

⁷⁸ Pachetele au fost capturate cu ajutorul programului *tcpdump*, rulat în mod *logging* (toate pachetele primite se salvează în forma binară într-un fișier în loc să fie analizate ad-hoc și afișate). Din păcate timpul nu a permis prelucrarea fișierelor salvate, pentru o eventuală evaluare a reordonării pachetelor.

și 140000 pps⁷⁹. Astfel a fost posibilă surprinderea punctului în care rata pachetelor de la ieșire începe să difere față de cea a pachetelor de la intrare (apar pierderi de pachete).

Am avut în vedere evaluarea codului specific comutării în contextul implementării VLAN-urilor. Astfel portul LISA în care au fost injectate pachete a fost configurat în mod acces, iar dintre celelalte două porturi unul a fost configurat în mod acces și celelalte două în mod trunchi. Am efectuat două teste, unul pentru transmisii unicast (adresa MAC destinație a fost adăugată static în tabela de comutare) și unul pentru transmisii broadcast. Ambele teste au fost efectuate injectând 1000000 de pachete de 64 bytes. Tabelul următor prezintă rezultatele testului pentru transmisii unicast:

Descriere Unicast, portul sursă în mod acces, portul destinație în trunchi
Lung. cadru 64
Cadre trimise 1000000
ID Test 2

t_d (ns)	PPS in	N_out	Hw IRQ%	Sw IRQ%	Pierderi%	Pierderi	PPS out
22000	41225	1000000	37.2	61.5	0.0000	0	41225
21000	43009	1000000	36.8	63.1	0.0000	0	43009
20000	45001	1000000	34.6	64.4	0.0000	0	45001
19000	47055	1000000	33.0	67.0	0.0000	0	47055
18000	49076	1000000	30.4	69.6	0.0000	0	49076
17000	51771	1000000	29.3	70.7	0.0000	0	51771
16000	54388	1000000	27.9	72.1	0.0000	0	54388
15000	57522	1000000	25.2	74.5	0.0000	0	57522
14000	60845	999957	23.4	76.3	0.0043	43	60842
13000	64770	999801	19.8	80.2	0.0199	199	64757
12000	69147	999592	16.7	83.0	0.0408	408	69119
11000	74188	999363	12.2	87.8	0.0637	637	74141
10000	79958	999270	9.1	90.3	0.0730	730	79900
9000	86673	999273	5.4	94.3	0.0727	727	86610
8000	95651	993160	0.3	99.7	0.6840	6840	94997
7000	106165	896137	1.6	97.9	10.3863	103863	95138
6000	118759	801146	1.6	97.9	19.8854	198854	95143
5000	134116	701882	1.3	98.5	29.8118	298118	94134
4000	142067	658328	1.1	98.4	34.1672	341672	93527

Coloanele din tabel reprezintă, în ordine: valoarea de temporizare folosită pentru generarea cadrelor; rata pachetelor generate; numărul pachetelor comutate; timpul petrecut de CPU în rutine de tratare întreruperi hardware; timpul petrecut de CPU în rutine de tratare softirq; procentul pachetelor pierdute; numărul pachetelor pierdute; rata calculată a pachetelor comutate (numărate de mașinile de test).

Tabelul următor reprezintă rezultatele testelor obținute pentru transmisii broadcast:

⁷⁹ Pachete per secundă.

Descriere	Broadcast, portul sursă în mod acces, porturile destinație în mod acces și trunc
Lung. cadru	64
Cadre trimise	1000000
ID Test	4

t_d (ns)	PPS in	N_out	Hw IRQ%	Sw IRQ%	Pierderi%	Pierderi	PPS out
22000	41224	1000000	25.1	74.9	0.0000	0	41224
21000	43018	1000000	24.0	76.0	0.0000	0	43018
20000	44999	1000000	22.0	78.0	0.0000	0	44999
19000	47061	1000000	19.5	80.5	0.0000	0	47061
18000	49074	999974	17.0	83.0	0.0026	26	49073
17000	51763	999978	15.1	84.9	0.0022	22	51762
16000	54399	999888	12.1	87.9	0.0112	112	54393
15000	57513	999704	9.8	90.2	0.0296	296	57496
14000	60836	999575	7.0	93.0	0.0425	425	60810
13000	64761	999284	3.5	96.5	0.0716	716	64715
12000	69157	991979	0.9	99.1	0.8021	8021	68602
11000	74185	951415	0.7	99.3	4.8585	48585	70581
10000	79972	861396	3.4	96.2	13.8604	138604	68888
9000	86674	796226	3.6	95.7	20.3774	203774	69012
8000	95634	715991	3.2	96.3	28.4009	284009	68473
7000	106194	647198			35.2802	352802	68729
6000	118802	570189			42.9811	429811	67740

Rezultatele obținute la cele două teste sunt reprezentate sintetizat în următorul grafic:

Rata pachetelor comutate

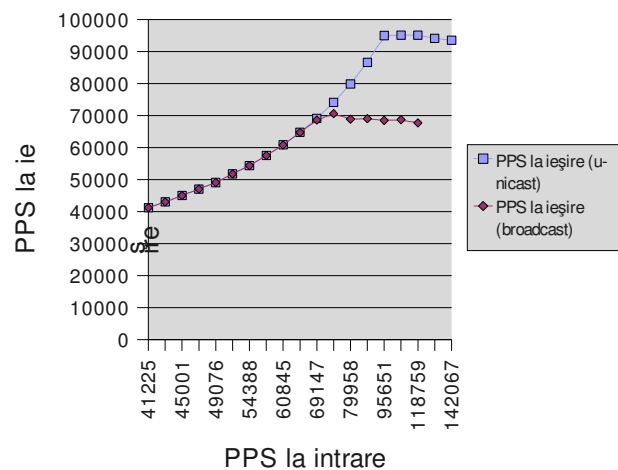


Figura 17 □ Rata pachetelor comutate

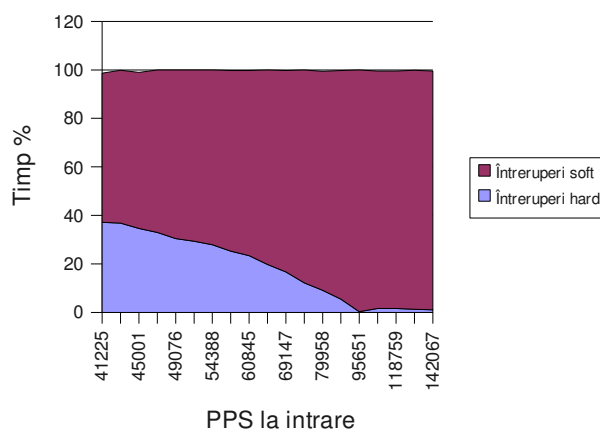
Comportamentul care poate fi observat în Figura 17 este pe deplin explicat din punct de vedere teoretic. Implementarea NAPI asigură menținerea unui nivel aproape constant al numărului de pachete de intrare procesate, atunci când numărul acestora depășește puterea de prelucrare a sistemului. Așa cum era de așteptat, punctul de frângere apare mai repede în cazul traficului broadcast deoarece timpul necesar procesării pachetelor este mai mare în această situație (cadrele trebuie comutate către toate celelalte porturi, apar în plus

parcurgeri de liste în VDB etc.).

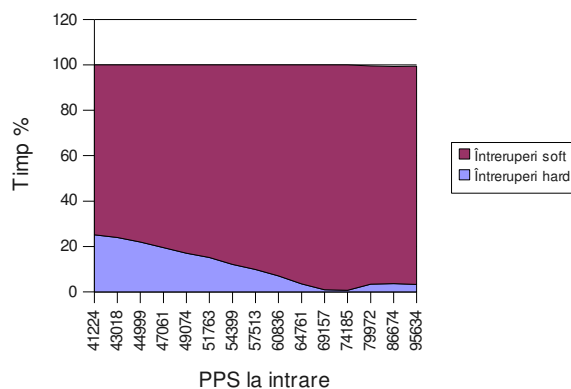
Punctele în care cele două grafice se frâng reprezintă punctele în care încep să apară pierderi de pachete și rata pachetelor comutate nu mai coincide cu cea a pachetelor generate.

Pentru a evidenția comportamentul NAPI, am reprezentat grafic și timpul petrecut de CPU în rutinele de tratare a întreruperilor hardware și software în cazul celor două teste.

Repartiția utilizării CPU (unicast)



Repartiția utilizării CPU (broadcast)



Se poate observa că, atât în cazul traficului unicast cât și al celui broadcast, procesorul este ocupat aproape constant 100% cu procesarea întreruperilor. Predomină timpul petrecut procesând întreruperi software, deoarece implementarea NAPI (spre deosebire de Softnet) mută cea mai mare parte a procesării de pachete în context softirq (permițând astfel funcționarea sistemului în condițiile unei rate foarte mari a pachetelor primite).

Ceea ce poate contează cel mai mult pentru utilizatorii finali nu este rata pachetelor comutate (care a fost evaluată în condiții de laborator pentru a evidenția comportamentul algoritmilor utilizați), ci lățimea de bandă în cazul unor transferuri obișnuite (în care stiva TCP/IP generează pachete de dimensiune mare, iar rata pachetelor este astfel mult mai mică).

Pentru evaluarea performanțelor în cazul transferurilor de date, am realizat o nouă configurație, prezentată în figura următoare.

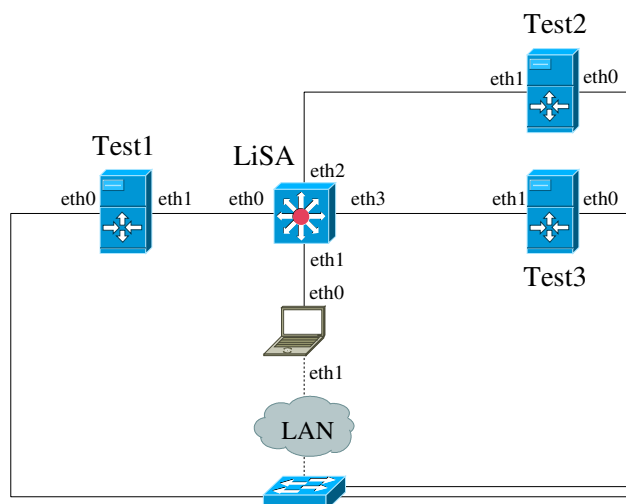


Figura 18 □ Configurația folosită pentru măsurarea lărgimii de bandă

Testele au fost efectuate folosind utilitarul *nc*, indirectat dinspre */dev/zero* pe mașinile folosite ca servere și redirectat către */dev/null* pe mașinile folosite pentru a simula clienții. Lărgimea efectivă de bandă pe fiecare mașină a fost măsurată folosind *iptraf*. Având în vedere că interfața *eth0* de pe sistemul LISA este Gigabit, transferurile au fost făcute asimetric, adică având mașina Test1 la un capăt și mașinile Test2, Test3 și laptopul la celălalt capăt.

Au fost efectuate două teste separate. Primul test a folosit transferuri în ambele sensuri⁸⁰, iar al doilea test transferuri într-un singur sens, respectiv având ca sursă mașina Test1. Tabelul următor prezintă pe scurt rezultatele celor două teste:

	<i>Test1</i>	<i>Test2</i>	<i>Test3</i>	<i>Laptop</i>
Unidir In (Mb/s)	199.59	0.15	1.51	1.41
Unidir Out (Mb/s)	4.43	68.39	68.44	63.24
Unidir Total (Mb/s)	204.02	68.54	69.95	64.65
Bidir In (Mb/s)	148.29	20.37	18.58	23.1
Bidir Out (Mb/s)	69.8	50.66	50.73	49.29
Bidir Total (Mb/s)	218.09	71.03	69.31	72.39

După cum se poate observa, viteza totală de transfer în cele două cazuri este comparabilă (aproximativ 200 Mb/s). Acest fenomen poate fi explicat prin limitarea vitezei de transfer de către lărgimea de bandă a magistralei PCI din sistemul LISA. Ținând cont că orice pachet comutat traversează de două ori magistrala (o dată de la interfața de intrare în memoria centrală și încă o dată din memoria centrală la interfața de ieșire) se poate calcula aproximativ lărgimea de bandă a magistralei PCI: $400 \square 450$ Mb/s.

⁸⁰ Deși în practică această situație nu este foarte des întâlnită, modul de operare *full duplex* al tuturor interfețelor implicate a făcut posibilă rularea acestui test.

Practic viteza de transfer a magistralei este lărgimea de bandă maximă care se poate atinge atunci când pachetele sunt suficient de mari pentru ca rata lor să nu atingă punctul critic. În sprijinul acestei explicații vine și distribuția utilizării CPU pe sistemul LISA în timpul testelor:

<i>Starea</i>	<i>Transfer unidirecțional</i>	<i>Transfer bidirecțional</i>
Inactiv %	20	25
Întrepereri hardware %	48	45
Întrepereri software %	32	30

Se poate observa că procesorul are chiar perioade de inactivitate, ceea ce confirmă că limitarea apare datorită transferurilor pe magistrala PCI (care se realizează prin DMA, fără a solicita procesorul) și nu datorită complexității prea mari a algoritmilor utilizați.

Nu în ultimul rând, trebuie să menționez că sistemul LISA (cu cea mai recentă versiune a codului) a fost folosit în producție timp de două săptămâni, servind ca switch pentru trei stații de lucru. Portul de uplink a fost configurat în trunchi și conectat cu un switch Catalyst 2924. Sistemul a funcționat continuu pe parcursul celor două săptămâni (fără reporniri ale SO) și nu s-au înregistrat probleme. Kernel-ul rulat în timpul perioadei de testare funcțională a fost compilat cu majoritatea opțiunilor de depanare (patch-ul kdb, verificarea alocării memoriei, verificarea socket buffer-elor etc.).

6. Concluzii

Secțiunea anterioară oferă o imagine de ansamblu asupra performanțelor unui sistem LISA atât în condiții speciale, de laborator, cât și în condiții reale de utilizare. Așa cum am arătat, performanțe nu depind atât de algoritmi utilizați (optimizați, de altfel, pe cât posibil), cât de limitările introduse de hardware și de faptul că pachetele trebuie transferate de două ori pe magistrala PCI a sistemului pentru a putea fi comutate.

Așa cum am afirmat încă de la început, ținta principală a LISA este de a oferi facilități foarte avansate (comparabile cu ale produselor de vârf de pe piață) la un preț foarte scăzut.

Fiind bazat pe kernelul Linux și folosind componenta de rețea a acestuia, LISA poate rula pe orice platformă suportată de Linux și poate folosi orice chip-uri de rețea pentru care există drivere în Linux, *fără a fi necesară adaptarea codului*. Practic un sistem vechi, neperformant, astăzi considerat depășit pentru aplicațiile desktop, poate fi transformat extrem de ușor într-un switch multistrat.

Bazat pe o interfață în linia de comandă similară cu a Cisco IOS, LISA oferă un mod de configurare ușor, flexibil și comparabile cu al celor mai performante produse comerciale din domeniu.

Proiectul LISA este o inițiativă open-source, întocmai ca sistemul Linux, pe baza căruia a fost dezvoltat. Scopul este de a încuraja dezvoltarea de software gratuit, sub licență GPL, și de a beneficia astfel de ajutorul benevol al unui număr cât mai mare de dezvoltatori.

Versiunea actuală a LISA este stabilă (nu se cunosc probleme), dar perioada de testare a fost destul de scurtă. Este nevoie de teste pe mult mai multe configurații (cât mai variate) pentru a putea afirma că proiectul poate fi folosit în producție. În acest sens una dintre priorități o constituie integrarea modulului kernel al LISA în distribuția oficială a Linux kernel.

Deși funcțiile de bază pentru comutarea de pachete la nivel 2 și 3 sunt implementate, există multe protocoale auxiliare care momentan lipsesc. Prezint în continuare o listă (sortată pe cât posibil în ordinea priorității) a facilităților care ar trebui implementate în viitorul apropiat:

- Detectarea buclelor în rețea și asigurarea redundanței – algoritmul STP, standardul IEEE 802.d, protocolul VSTP+.
- Optimizarea traficului de multicast prin implementarea IGMP Snooping⁸¹.
- Implementarea mecanismelor de securitate a porturilor (filtrare după adresa MAC etc.).

⁸¹ IGMP Snooping este o modalitate prin care un switch poate direcționa traficul multicast doar către acele porturi pe care sunt conectate stații interesate să primească acel trafic. Switch-ul supraveghează mesajele IGMP trimise de către stații și adaugă automat în tabela de comutare înregistrări statice pentru adresele MAC de multicast.

- Detectarea automată a echipamentelor Cisco direct conectate, folosind protocolul CDP.
- Transferul automat al bazei de date cu VLAN-uri, folosind protocolul VTP.
- Implementarea interfețelor virtuale specifice procesului de rutare (interfețe *loopback* și *null*).
- Posibilitatea de a marca rutele pentru a identifica sursa din care au fost injectate în tabela de rutare.
- Implementarea protocoalelor de rutare RIP, OSPF și BGP.
- Rutarea pe baza adresei sursă (*source routing* sau *policy routing*).

Deși foarte pretențioase, facilitățile enumerate mai sus pot fi implementate dacă proiectul ia amploare și comunitatea open-source se implică în dezvoltarea lui.

Codul sursă, împreună cu o documentație momentan destul de sumară, sunt disponibile pe site-ul oficial al proiectului, <http://lisa.ines.ro/>.

7. Anexe

Anexa A □ Sincronizarea RCU

Termenul *RCU* este de fapt prescurtarea de la *Read-Copy Update* și se referă la mecanisme prin care consistența datelor poate fi asigurată fără folosirea mecanismelor blocante de sincronizare (cum ar fi un *spinlock* sau un *semafor*).

Chiar sursele Linux kernel includ documentație destul de amplă pentru RCU (vezi [LKT01]). Publicațiile on-line sunt și ele destul de bogate în materiale despre RCU. Eu voi încerca să fac o prezentare de ansamblu, arătând câteva principii de bază ale RCU, precum și modul în care acestea pot fi aplicate în Linux.

Conform [LKT01], primele publicații despre principii asemănătoare RCU au apărut în 1980. Au trecut 15 ani până când prima implementare de RCU a fost realizată în kernel-ul DYNIX/ptx și încă 3 ani până la apariția unui material care să prezinte implementarea. Prima prezentare a principiilor RCU în Linux a avut loc în 2001 și tot atunci au apărut și primele implementări.

Idea de bază a RCU este de a separa operațiile destructive în două părți: prima asigură că elementul care se șterge nu este "văzut de nimeni", iar a doua realizează efectiv ștergerea. Între cele două evenimente trebuie să existe o "perioadă de grație". Această perioadă asigură ca nici unul dintre cititorii care au apucat să vadă elementul care se șterge să nu îl mai folosească (nu mai au referințe către el) în momentul în care are loc efectiv ștergerea. Spre exemplu, ștergerea unui element dintr-o listă înlanțuită s-ar face în felul următor: se scoate elementul din listă, se așteaptă trecerea perioadei de grație, apoi se *dealocă* elementul.

Avantajul principal al RCU este că firele de execuție cititoare nu trebuie să acapareze vreun dispozitiv de sincronizare, să execute operații în mod atomic sau să aștepte bariere de memorie⁸². Pentru că pe procesoarele moderne toate aceste mecanisme clasice de sincronizare sunt extrem de costisitoare (în ceea ce privește timpul), RCU îmbunătățește simțitor performanțele în situațiile în care predomină operațiile de citire.

Deși nu folosesc mecanisme clasice de sincronizare, cititorii RCU cunosc totuși o așa-numită zonă critică. Dacă în cazul mecanismelor clasice de sincronizare zona critică era delimitată de acapararea și respectiv eliberarea dispozitivului de sincronizare, în cazul RCU aceasta e delimitată de marcarea (și respectiv resetarea marcajului) unei stări speciale a procesorului curent: în această stare nu sunt permise operațiile blocante, iar procesorul nu va comuta în mod utilizator și nu va intra în bucla de inactivitate (idle loop). Este de notat că intrarea și ieșirea din zona critică RCU sunt extrem de rapide, deoarece acestea nu fac decât să incrementeze și respectiv să decrementeze un contor (care indică dacă firul curent

⁸² Pe unele platforme (cum ar fi procesoarele Alpha) este totuși necesară folosirea barierei de memorie datorită benzilor de asamblare (pipelines) și reordonării instrucțiunilor de către unitățile de optimizare.

de execuție este sau nu prelevabil).

Cititorii nu semnalează explicit ieșirea din zona critică, însă proprietățile acestora fac posibilă determinarea perioadei de grație. Cum în timpul zonei critice nu sunt permise blocarea, execuția în mod utilizator și bucla inactivă, este clar că în momentul în care un procesor trece printr-una dintre aceste stări toate regiunile critice ale firelor de execuție care rulează pe el s-au încheiat. Trecerea printr-una dintre stările enumerate poartă denumirea de *stare de latență*⁸³. Se consideră că perioada de grație a trecut după ce toate procesoarele au trecut cel puțin o dată prin starea de latență.

Chiar dacă RCU are în spate o teorie destul de vastă, api-ul se rezumă la doar câteva apeluri, cu ajutorul cărora se pot implementa majoritatea schemelor de sincronizare specifice RCU.

- `rcu_read_lock()` □ marchează intrarea într-o zonă critică de citire. Practic este incrementat contorul de preemptivitate al firului de execuție curent, ceea ce împiedică prelevarea procesorului.
- `rcu_read_unlock()` □ marchează ieșirea din zona critică de citire. Are loc decrementarea contorului de preemptivitate.
- `synchronize_kernel()` □ blochează firul curent de execuție până când toate procesoarele trec cel puțin o dată prin starea de latență.
- `call_rcu()` □ planifică apelarea unei funcții după ce toate procesoarele au trecut prin starea de latență. Apelul `call_rcu()` se întoarce imediat, urmând ca funcția (pasată ca argument) să fie apelată ulterior de către kernel printr-un mecanism de tip *callback*.
- `smp_wmb()` □ introduce o barieră de scriere în memorie, care asigură consistența datelor în contextul reordonării instrucțiunilor. Bariera asigură ca datele să fie scrise efectiv în memorie înaintea executării unor eventuale instrucțiuni ulterioare de citire.
- `rcu_dereference()` □ realizează dereferențierea în siguranță a unui pointer. Aceasta asigură ca procesorul să citească pointerul *înaintea* datelor de la adresa indicată de pointer. Acest lucru este esențial pe procesoarele Alpha. În plus, este o modalitate foarte bună de a documenta codul, întrucât dă indicii clare despre pointerii protejați prin RCU.
- `rcu_assign_pointer()` □ asigură scrierea în memorie a datelor indicate de pointer *înainte* de a se modifica efectiv valoarea pointerului.

Trebuie să menționez că RCU asigură consistența între *mai multe* fire de execuție care *citesc* și *un singur* fir de execuție care *scrie*. Dacă însă există mai multe fire de execuție care scriu, acestea trebuie sincronizate între ele prin mecanisme clasice. De aceea, RCU prezintă avantaje doar în cazul în care predomină operațiile de citire.

Pentru a explica sumar principiile de realizare a sincronizării, voi construi câteva exemple simple pe baza listelor înlănțuite. Pentru implementarea generică de liste înlănțuite din Linux kernel există variante RCU ale celor mai importante macro-uri pentru lucrul cu liste. Acestea asigură consistența *doar pentru înlănțuirea elementelor în listă*. Cu alte cuvinte, folosind variantele RCU ale macrourilor se poate *scoate din listă* în siguranță un element, dar acesta nu poate fi *dealocat* în siguranță fără a folosi sincronizări RCU suplimentare.

⁸³ Termenul original, folosit în literatura de specialitate este *quiescent state*. Se referă la o stare de "liniște", în sensul că nu se execută zone critice RCU.

Pentru exemplele următoare, consider liste înlănțuite care au elemente de următoarea formă:

```
struct list_entry {
    struct list_head lh;
    struct rcu_head rcu;
    /* alte elemente utile */
};
```

Exemplul 1: mai multe fire de execuție parcurg lista pentru citire, un fir de execuție șterge un element din listă.

Citare

```
struct list_entry *entry;
struct list_head *entries;

/* ... */
rcu_read_lock();
list_for_each_entry_rcu(entry, entries, lh)
{
    /* procesare entry */
}
rcu_read_unlock();
```

Sciere

```
struct list_entry *entry;

/* entry e un pointer la elementul care
   trebuie șters
   */
list_del_rcu(&entry->lh);
synchronize_kernel();
kfree(entry);
```

Se consideră că firul de scriere este singurul care modifică lista (în caz contrar trebuie folosite mecanisme clasice pentru a sincroniza între ele firele care fac modificări). În plus, este obligatoriu ca firul care scrie să nu fie în zonă critică RCU (pentru că funcția `synchronize_kernel()` este blocantă și blocarea nu este permisă în zonă critică RCU).

Exemplul 2: mai multe fire de execuție parcurg lista pentru citire, un fir face ștergere în zonă critică

Citare

```
struct list_entry *entry;
struct list_head *entries;

/* ... */
rcu_read_lock();
list_for_each_entry_rcu(entry, entries, lh)
{
    /* procesare entry */
}
rcu_read_unlock();
```

Sciere

```
void free_entry(struct rcu_head *head) {
    struct list_entry *entry =
        container_of(head, struct list_entry,
                    rcu);
    kfree(entry);
}

struct list_entry *entry;

rcu_read_lock();
/* ... */
list_del_rcu(&entry->lh);
call_rcu(&entry->rcu, free_entry);
/* ... */
rcu_read_unlock();
```

Firul de scriere nu poate aștepta trecerea procesoarelor prin starea de latență deoarece se află în regiune critică RCU. Prin urmare, programează apelarea amânată a funcției `free_entry()` (care realizează efectiv dealocarea) și continuă imediat procesarea.

Exemplul 3: Adăugarea unui element

În condițiile listelor înlănțuite manipulate cu variantele RCU ale macro-urilor, problemele de sincronizare sunt rezolvate "de la sine", adică apelurile `list_add_rcu()` și `list_add_tail_rcu()` rezolvă toate problemele de sincronizare.

Totuși, vreau să pun în evidență un alt principiu al sincronizării RCU, și anume consistența datelor din elemente. Acest principiu este foarte important pentru adăugarea și modificarea corectă a elementelor.

Să presupunem că structura `list_entry` pe care am folosit-o pentru exemplificare conține un șir de caractere sub forma unui vector de `char` (`char str[10]`). Considerăm un element al listei pentru care câmpul `str` este modificat caracter cu caracter. Firul care execută modificarea ar putea fi întrerupt de un alt fir, care citește câmpul `str`. Acesta din urmă va citi date inconsistente: în `str` se află doar o parte din noua valoare, apoi caractere din valoarea inițială.

Menționez că realizarea modificării într-o zonă critică RCU nu este o soluție bună. Deși funcționează pe sistemele uniprocessor (pentru că zona critică RCU nu poate fi întreruptă), apar probleme de portabilitate. Pe un sistem SMP un alt procesor poate citi *concomitent* câmpul `str`.

În cazul adăugării de elemente, toate câmpurile trebuie să fie inițializate înainte de a adăuga elementul în listă. Aceasta asigură consistența la o citire concomitentă care începe imediat după adăugarea elementului⁸⁴.

Pentru a asigura consistența datelor la modificare, soluția este *înlocuirea* întregului element, parcurgând următorii pași:

- se alocă un element nou;
- se copiază elementul vechi peste cel nou;
- se fac modificările în elementul nou;
- se înlocuiește elementul vechi cu cel nou.

Practic de la acești pași vine și denumirea de RCU. Principul actualizării prin citire și copiere (*read-copy update*) este exact cel prezentat mai sus.

Problemele de consistență a datelor pot fi mult mai complicate decât exemplele pe care le-am prezentat. Literatura de specialitate oferă și soluții (cum ar fi marcarea pentru ștergere, dar ele depășesc obiectul acestei lucrări.

Anexa B □ Comutarea fără-copiere

În cazul general, comutarea unui pachet de pe o interfață de intrare pe una de ieșire presupune două copieri ale pachetului. În continuare, voi explica de ce sunt necesare aceste copieri și cum pot fi evitate în cazul unui hardware mai performant.

Majoritatea chip-urilor de rețea sunt capabile să transfere date direct cu memoria centrală folosind mecanismul DMA. Memoria din interiorul chip-ului de rețea este în general destul de mică: este comparabilă cu dimensiunea maximă a unui pachet, adică are aproximativ 1.5 Kbytes. Pentru a compensa acest neajuns, se folosesc niște zone de memorie tampon circulare (*ring buffers*). Există două astfel de buffere separate: unul pentru recepție și altul pentru trimitere de pachete.

⁸⁴ Din considerente de reordonare a instrucțiunilor, ar trebui folosit explicit apelul `smp_wmb()` între inițializarea câmpurilor și adăugarea elementului în listă (reordonarea ar putea executa instrucțiunile de adăugare în listă înaintea unor instrucțiuni care inițializează elemente, deși în codul sursă apar în ordinea corectă). În cazul listelor sincronizate prin RCU, macro-urile de adăugare rezolvă această problemă.

O dată primit un pachet, acesta este transferat imediat în memoria centrală (prin DMA) pentru a putea recepționa un nou pachet în memoria de pe chip. Într-un registru de stare se setează bitul care indică primirea unui pachet și procesorul este semnalizat prin intermediul unei întreruperi. La nivelul kernel-ului, driverul trebuie să aloce un socket buffer și să copieze datele din buffer-ul circular în zona de date utile ale socket buffer-ului. Copierea trebuie să aibă loc cât mai repede cu putință, pentru a asigura ca în buffer să rămână spațiu pentru a primi noi pachete.

Alocarea zonei de date utile a socket buffer-ului direct în bufferul circular de recepție nu este posibilă. Chiar și în cazul în care pachetul va fi trimis mai departe pe o altă interfață, nu poate nimeni garanta că acesta apucă să fie procesat de kernel și copiat de interfața de ieșire până când bufferul apucă să bucleze și ajung să fie suprascrise vechile date. În cazul în care pachetul este destinat chiar mașinii respective (nu trebuie comutat sau rutat, ci face parte dintr-un socket local), aproape sigur acesta nu va apuca să fie copiat în spațiu utilizator până când bufferul de recepție buclează. Prin urmare, este absolut necesară alocarea unui spațiu de memorie separat și realizarea imediată a copierii din bufferul circular de către driver.

La trimitere lucrurile sunt destul de asemănătoare. Procesorul depune unul sau mai multe pachete în bufferul circular de trimitere, apoi instruește chip-ul plăcii de rețea să înceapă trimiterea. Acesta din urmă va copia pe rând pachetele în memoria proprie, le va trimite și, în cele din urmă, atunci când toate pachetele au fost trimise și bufferul circular se golește, procesorul va fi semnalizat printr-o întrerupere.

Bufferele circulare sunt alocate de driver la inițializare, înainte de a configura chip-ul. Zona de memorie folosită trebuie să fie capabilă DMA, pentru ca accesul chip-ului să fie posibil. După alocare, adresele bufferelor sunt depuse în niște registre speciale ale chip-ului și rămân nemodificate pe toată durata de viață a driverului.

La trimitere este necesară o a doua copiere a pachetului tocmai pentru că adresa bufferului de trimitere este fixă. Datele pachetului se găsesc sigur în altă zonă, fie că socket bufferul a fost creat de către un driver la recepția unui pachet pe o interfață, fie că a fost creat de un socket pentru trimiterea datelor.

Cele două copieri ale datelor au un impact puternic asupra performanței. Producătorii de hardware au căutat să elimine acest neajuns prin îmbunătățirea logicii implementate în chip-urile de rețea.

Rezolvarea a fost destul de simplă: în loc să se păstreze chiar datele pachetului în bufferele circulare, se păstrează pointeri către acestea.

La recepția pachetelor, driverul cooperează prin *prealocarea* de socket buffere (acestea sunt alocate înainte ca pachetele să fie primite). O dată cunoscută adresa zonei de date utile, aceasta poate fi depusă în bufferul circular de recepție. Atunci când primește un pachet, chip-ul de rețea îl va transfera în memoria sistemului la adresa din buffer în loc să îl transfere direct în buffer. Cum adresa din buffer este exact adresa zonei de date utile a socket buffer-ului, practic chip-ul de rețea transferă pachetul *direct în socket buffer*, fără a mai fi nevoie de o copiere suplimentară. Adresa bufferului circular de recepție rămâne în continuare fixă.

Se impun însă câteva observații. Pentru ca chip-ul de rețea să depună pachetul direct în socket buffer, partea de date utile a acestuia trebuie alocată într-o zonă de memorie

capabilă DMA. Driverul beneficiază de cooperarea kernel-ului pentru a îndeplini această cerință. O altă problemă este că dimensiunea pachetului nu este cunoscută la momentul alocării zonei de date utile a socket buffer-ului. Cum chip-ul de rețea depune pachetul direct în socket buffer, trebuie luate măsuri speciale pentru a asigura ca scrierile să nu se facă în afara zonei alocate.

La trimitere, lucrurile sunt oarecum asemănătoare. Datele utile ale pachetului se găsesc în zone de memorie capabile DMA. În loc să fie copiate datele în sine în bufferul de trimitere, se depune doar un pointer către ele. La transmitere, chip-ul va prelua datele direct din zona de date utile a socket buffer-ului.

Această abordare a permis implementarea foarte ușoară a unei alte facilități, cunoscută sub numele de *Scatter-Gather I/O*. Aceasta permite "asamblarea" pachetului direct de către chip-ul de rețea, la trimitere. Spre exemplu antetul de nivel 2 și PDU de nivel 2 pot să se găsească în zone neadiacente de memorie. La trimitere se vor depune în bufferul circular adresele celor două fragmente, iar asamblarea pachetului se va face direct în memoria tampon a chip-ului. Aceasta scutește copieri suplimentare în memoria centrală pentru asamblarea pachetului.

Socket bufferele implementează funcționalitate specială pentru a păstra și manipula pachete fragmentate. Mai mult, stiva de rețea din kernel realizează automat asamblarea unui pachet fragmentat înainte de trimitere în cazul în care driverul nu suportă pachete fragmentate.

Anexa C □ Algoritmul STP

Algoritmul de comutare a pachetelor poate fi rezumat în ultimă instanță la 3 reguli:

- Dacă adresa destinație a cadrului este cunoscută și portul destinație este același cu cel de intrare, ignoră cadrul.
- Dacă adresa destinație a cadrului este cunoscută și portul destinație este diferit de cel de intrare, trimite cadrul pe portul destinație.
- Dacă adresa destinație a cadrului nu este cunoscută, trimite cadrul pe toate porturile mai puțin cel de intrare.

Reamintesc faptul că regulile descrise mai sus se aplică întotdeauna în interiorul VLAN-ului pe care a sosit cadrul.

În cazul în care într-o rețea există mai multe switch-uri și cel puțin o legătură redundantă, pot apărea probleme dacă toate switch-urile aplică doar cele 3 reguli de mai sus.

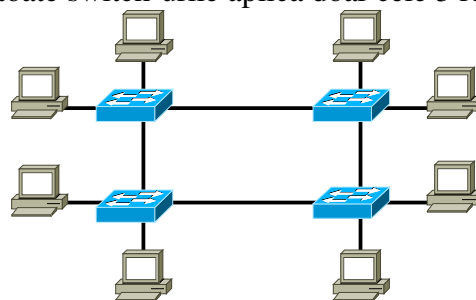


Figura 19 □ Exemplu de rețea redundată

Pentru exemplul din Figura 19 se poate arăta destul de ușor că, o dată ce una dintre stații a trimis un cadru, acesta se va reîntoarce în switch-ul direct conectat cu stația care a trimis cadrul. Cum switch-urile nu au nici o modalitate prin care să recunoască un cadru care a trecut deja prin ele, acesta va bucla la infinit prin rețea.

Totuși legăturile redundante pot fi extrem de utile pentru a asigura funcționarea rețelei chiar în cazul în care se defectează unul dintre switch-uri sau se întrerupe o legătură. Pentru a face posibilă funcționarea unei rețele cu legături redundante, proiectanții au dezvoltat protocolul STP (Spanning-Tree Protocol). Acesta alege din graful rețelei doar un arbore de acoperire, restul de legături (care ar închide cicluri) fiind ignorate.

Principalul avantaj al STP este că nu necesită un dispozitiv central pentru funcționare. Cu alte cuvinte este un protocol descentralizat. Pentru a putea funcționa descentralizat, protocolul începe prin alegerea (dinamică) a unui switch rădăcină. Apoi toate celelalte switch-uri aleg calea optimă spre rădăcină.

Funcționarea STP se bazează pe două tipuri de cadre speciale, numite BPDU (*Bridge Protocol Data Unit*):

- BPDU de configurare (Configuration BPDUs). Sunt folosite pentru alegerea switch-ului rădăcină, pentru a anunța costurile acumulate și valorile unor cronometre.
- BPDU de anunțare a schimbării topologiei (Topology change notification BPDUs). Aceste cadre sunt folosite atunci când apar schimbări ale topologiei (de exemplu atunci când se defectează o componentă).

Algoritmul STP rulează în 3 etape:

1. Selectarea switch-ului rădăcină. Acesta va fi rădăcina arborelui de acoperire generat. Spre deosebire de algoritmul clasic al arborelui minim de acoperire, la STP alegerea rădăcinii nu se face după costuri. În schimb, fiecare switch are un identificator unic de 6 bytes. Schimbând între ele BPDU de configurare, switch-urile aleg rădăcina ca fiind switch-ul cu identificatorul cel mai mic dintre ele.
2. Determinarea portului rădăcină al fiecărui switch. Fiecare switch selectează interfața de rețea care are calea de cost minim spre switch-ul rădăcină. Acesta devine portul rădăcină al switch-ului.
3. Determinarea *switch-ului desemnat* pentru fiecare segment LAN. Atunci când un segment de LAN este conectat cu mai multe switch-uri prin care există căi diferite spre switch-ul rădăcină, doar unul dintre acestea trebuie ales pentru a trimite traficul spre switch-ul rădăcină. Astfel se asigură o topologie în formă de arbore. Portul switch-ului desemnat care conectează segmentul de LAN se numește *port desemnat*.

Toate porturile de switch-uri care nu au fost alese ca porturi rădăcină sau porturi desemnate sunt blocate. Acestea nu vor transporta cadre normale, dar vor primi în continuare cadre BPDU. Astfel un port blocat se poate reactiva în cazul defectării unei componente.

Pentru a preîntâmpina buclele temporare (care pot apărea atunci când unele switch-uri nu cunosc încă topologia globală a rețelei), au fost introduse două stări suplimentare ale unui port, pe lângă cea de blocare și cea în care comută pachetele. Diagrama de tranziții pentru porturi și condițiile de trecere dintr-o stare în alta depășesc scopul acestei lucrări. Mai multe detalii despre acestea se pot găsi în [Wehrle01], subsecțiunea 12.2.4 (Spanning-Tree

Protocol).

Rularea algoritmului STP în rețelele care folosesc VLAN-uri și legături în trunchi ridică o serie de probleme. Algoritmul STP așa cum a fost definit are sens în interiorul aceluiași domeniu de broadcast⁸⁵. Într-o rețea în care se folosesc VLAN-uri, fiecare dintre acestea reprezintă un domeniu de broadcast. Din acest punct de vedere, o legătură în trunchi este o legătură fizică aparținând mai multor domenii de broadcast. Cum STP a fost conceput pentru a asigura redundanța la nivelul legăturilor fizice, aplicarea lui în cazul rețelelor care folosesc VLAN-uri poate fi soluționată în mai multe feluri. Pentru o descriere amănunțită a posibilelor soluții și a avantajelor și dezavantajelor oferite de fiecare, se pot găsi mai multe detalii în [Finn01].

Standardul 802.1q specifică folosirea unei singure instanțe a algoritmului STP, care rulează pe VLAN-ul nativ⁸⁶. Cadrele BPDU sunt transmise întotdeauna fără marcaj 802.1q, asigurând astfel compatibilitatea cu dispozitivele care nu suportă legături în trunchi.

Cisco Systems a dezvoltat protocolul PVST+ (Per-VLAN Spanning Tree Plus), care este compatibil cu arborele de acoperire unic, specificat de standardul 802.1q, dar permite rularea unei instanțe diferite a algoritmului STP pe fiecare VLAN. Pentru fiecare arbore de acoperire există o singură cale activă, dar totuși, într-o rețea Cisco, aceasta poate fi diferită pentru fiecare VLAN. O prezentare sumară a PVST+ poate fi găsită în [McQuery01].

Anexa D □ Porturi în trunchi și rutare între VLAN-uri cu Linux bridge și 8021q

Am afirmat deja la începutul secțiunii de implementare că funcționalitatea (combinată) de porturi în trunchi, comutare și rutare între VLAN-uri este posibilă folosind modulele deja existente din Linux kernel, respectiv *bridge* și *8021q*. Deși posibilă, implementarea cu ajutorul acestor module are multe dezavantaje, pe care le-am enumerat deja în secțiunea dedicată implementării. Aici mă voi rezuma la a prezenta un exemplu de arhitectură, precum și comenzile care realizează configurarea.

Să presupunem că avem o mașină cu 4 interfețe de rețea (eth0, ... eth3), pe care dorim să le configurăm astfel:

- eth0 în trunchi, cu acces la VLAN-urile 1 și 2;
- eth1 în trunchi, cu acces la VLAN-urile 1 și 3;
- eth2 în mod acces, în VLAN-ul 2;
- eth3 în mod acces, în VLAN-ul 3.

Presupunem în plus că între VLAN-urile 2 și 3 dorim să realizăm și rutare, având adresa 192.168.2.254/24 pe VLAN-ul 2 și adresa 192.168.3.254/24 pe VLAN-ul 3.

Sucesiunea comenzilor care realizează această configurație este următoarea:

```
modprobe bridge
```

85 Domeniul de broadcast reprezintă acea parte a rețelei în care se propagă un cadru având ca adresă MAC destinație adresa de broadcast. Acesta este un mod foarte simplu și intuitiv de a denumi un segment de LAN conectat exclusiv cu echipamente de nivel 2.

86 Este vorba de VLAN-ul din care se consideră că fac parte cadrele fără marcaj 802.1q care sosesc pe un port configurat în trunchi.

```
modprobe 8021q

vconfig set_name_type DEV_PLUS_VID_NO_PAD
vconfig add eth0 1
vconfig add eth0 2
vconfig add eth1 1
vconfig add eth1 3

brctl addbr br1
brctl addif br1 eth0.1
brctl addif br1 eth1.1

brctl addbr br2
brctl addif br2 eth0.2
brctl addif br2 eth2

brctl addbr br3
brctl addif br3 eth1.3
brctl addif br3 eth3

ifconfig br2 192.168.2.254 netmask 255.255.255.0
ifconfig br3 192.168.3.254 netmask 255.255.255.0

ifconfig eth0 up
ifconfig eth1 up
ifconfig eth2 up
ifconfig eth3 up
```

Se observă numărul relativ mare de comenzi care sunt necesare. Configurarea devine extrem de anevoioasă în cazul mai multor porturi în trunchi și al unui număr mare de VLAN-uri comutate între acestea.

Pentru exemplul de mai sus (cu 4 interfețe de rețea), cazul cel mai defavorabil îl constituie configurarea în trunchi a tuturor interfețelor și accesul lor la toate cele 4094 VLAN-uri disponibile utilizatorului. În acest caz ar fi necesare $4 \cdot 4094 = 16376$ interfețe virtuale de tip 8021q, plus încă 4094 interfețe virtuale de tip bridge și tot atâtea bridge-uri. În aceste condiții (pur ipotetice, firește) o serie de algoritmi bazați pe identificarea unei interfețe de rețea prin căutare liniară în lista tuturor interfețelor devin extrem de neperformanți. Un exemplu concludent este funcția `dev_get_by_name()`, care localizează o interfață după nume, parcurgând liniar lista tuturor interfețelor și apelând `strcmp()` pentru fiecare dintre acestea.

Anexa E □ Echivalențe de termeni

Acquire (d. dispozitive de sincronizare) □ Acaparare; zăvorâre

Buffer □ Zonă tampon; Ring ~ □ Zonă tampon circulară

Lock □ Zăvor

Management Switch □ Switch configurabil

Preemption □ Prelevare

Race Condition □ Cursă Critică

Routing □ Rutare

Quiescent State (d. RCU) □ Stare de latență

Switch □ Comutator

Switching Table □ Tabelă de comutare

Timer □ Cronometru

8. Bibliografie

- **Javvin01**: The Javvin Company, *VLAN: Virtual Local Area Network and IEEE 802.1Q*, <http://www.javvin.com/protocolVLAN.html>
- **Cisco01**: Cisco Systems, *CCNA Curriculum*, http://www.cisco.com/en/US/learning/netacad/course_catalog/CCNA.html
- **Salim01**: Salim, Jamal Hadi; Olsson, Robert; Kuznetsov, Alexey, *Beyond Softnet*, 2001
- **Wehrle01**: Wehrle, Klaus; Pahlke, Frank; Ritter, Hartmut; Muller, Daniel; Bechler, Marc, *The Linux Networking Architecture*, 2004
- **Kuznetsov01**: Kuznetsov, Alexey; Salim, Jamal Hadi; Olsson, Robert, *NAPI Howto*, file:///linux-2.6.*/Documentation/networking/NAPI_HOWTO.txt
- **Lawyer01**: Lawyer, David S., *Text-Terminal-Howto*, <http://www.tldp.org/HOWTO/Text-Terminal-HOWTO.html>
- **Cisco02**: Cisco Systems, *Multicast in a Campus Network: CGMP and IGMP Snooping*, <http://www.cisco.com/warp/public/473/22.html>
- **LKT01**: The Linux Kernel Team, *Kernel Source RCU Documentation*, file:///linux-2.6.*/Documentation/RCU/
- **Finn01**: Finn, Norman, *Multiple Spanning Trees in 802.1Q*, 1996
- **McQuery01**: McQuerry, Steve, *Interconnecting Cisco Network Devices (ICND)*, 2003