Fiabilitatea în arhitectura calculatoarelor

Mihai Budiu — mailto:mihaib+@cs.cmu.edu
http://www.cs.cmu.edu/~mihaib/

decembrie 2001

Subiect: fiabilitatea în sistemele de calcul

Cunoştinţe necesare: noţiuni elementare despre arhitectura calculatoarelor

Cuvinte cheie: fiabilitate, redundanţă, toleranţa defectelor, cost

Cuprins

1 Introducere

În acest text voi prezenta arhitectura sistemelor de calcul dintr-un singur punct de vedere, şi anume cel al *fiabilităţii*. Prezentarea nu va fi exhaustivă şi nici matematizată; voi folosi mai ales exemple pentru a ilustra cum felurite consideraţii despre fiabilitate influenţează design-ul sistemelor de calcul.

Principalul subiect al teoriei fiabilităţii (reliability theory) este construirea sistemelor fiabile din componente nefiabile. Dacă un sistem ar funcţiona numai atunci toate componentele sale ar fi funcţionale, ar fi virtual imposibil de construit un sistem complex, pentru că fiabilitatea ar descreşte exponenţial cu numărul de componente.

Principala unealtă folosită în construirea sistemelor complexe este *abstracţia*. Un sistem este construit pe nivele; nivelul B este alcătuit din componente de nivel A. La rîndul lor, componente de nivel B sunt folosite ca şi cum ar fi atomice, indivizibile, pentru a construi nivelul C, şi aşa mai departe. Acest proces este inspirat din matematică, unde lemele şi teoremele sunt folosite drept componente elementare în demonstraţiile altor leme şi teoreme. În acest text vom privi alcătuirea unor nivele din arhitectura sistemelor de calcul din punctul de vedere al fiabilităţii pe care o oferă nivelelor superioare. Astfel putem distinge:

1. Nivele care *măresc fiabilitatea*, construind un tot mai fiabil din componente mai puţin fiabile. Acest lucru este obţinut folosind *redundanţă* în stocarea sau calculul informaţiei. Acest tip de nivel este cel mai adesea folosit în construcţia calculatoarelor contemporane. Secţiunea 3 discută multe astfel de exemple.

2. Nivele care *expun lipsa de fiabilitate* nivelelor superioare, lăsîndu-le pe acestea să rezolve imperfecţiunile. Nivelele superioare au adesea informaţii suplimentare despre cerinţele reale de fiabilitate ale sistemului, şi ca atare pot construi fiabilitate pe măsura necesităţilor. Arhitectura Internetului, pe care o vom discuta sumar în secţiunea 6 din numărul viitor al revistei este unul dintre cele mai notorii exemple de acest gen.

3. În fine, anumite nivele *partiţionează* resursele în parţi oarecum independente, izolate una de alta. Partiţionarea are drept efect *izolarea defectelor* (fault containment / fault isolation), astfel încît o defecţiune într-o parte să nu afecteze celelalte părţi. În calculatoare această tehnică este folosită în sistemele de operare şi clustere-le de calculatoare.

La ora actuală circuitele integrate pe scară largă (Very Large Scale Integrated circuits, VLSI) au ajuns la nivele incredibile de fiabilitate. Ca atare arhitecţii calculatoarelor în general privesc nivelul hardware ca fiind "perfect" şi folosesc această abstracţie foarte convenabilă în proiectarea nivelelor superioare. Anumite clase de aplicaţii însă au nevoie de o fiabilitate foarte ridicată (de exemplu, controlul de trafic aerian sau supervizarea centralelor nucleare). În astfel de sisteme critice arhitecţii sistemelor de calcul iau în considerare şi posibilitatea defectelor hardware, pe care le tratează în software.

Miniaturizarea continuă a circuitelor integrate va duce la schimbări în această stare de fapt; trebuie să ne aşteptăm ca în viitor circuitele să conţină din ce în ce mai multe defecţiuni şi să fie din ce în ce mai sensibile la fluctuaţii termodinamice şi particule de înaltă energie din radiaţia cosmică sau chiar din degradarea radioactivă a circuitului integrat! Astfel de schimbări vor cere probabil o regîndire completă a arhitecturii sistemelor de calcul.

2 Fiabilitate

În această secţiune voi introduce terminologia folosită în restul articolului.

2.1 Definiţie

Chiar dacă acest text nu este matematic, voi da o definiţie precisă a noţiunii de fiabilitate.

> Fiabilitatea unui obiect (o componentă sau un sistem) este o funcţie de timp $F(t)$, definită ca probabilitatea ca, în condiţii de mediu specificate, obiectul să funcţioneze adecvat în intervalul de timp $[0, t)$.

Putem face cîteva observaţii foarte importante legate de această definiţie:

- Fiabilitatea este o probabilitate, cuprinsă între 0 şi 1.
- Nu există sisteme perfect fiabile, pentru care $F(t) = 1$ dacă $t > 0$. Putem însă vorbi de sisteme "destul de fiabile", atunci cînd avem în minte anumite condiţii de utilizare şi constrîngeri bugetare pentru implementarea sistemului.
- Nu putem defini fiabilitatea fără a specifica mediul în care sistemul operează. Un calculator care merge foarte bine pe birou nu este neapărat potrivit pe un satelit.

2.2 Redundanţă

Ingredientul cel mai folosit pentru a construi sisteme fiabile este redundanţa. Putem distinge două genuri de redundanţă, spaţială şi temporală:

Redundanţa spaţială foloseşte mai multe componente decît strict necesar pentru a implementa un anumit sistem. Resursele adiţionale fac calcule suplimentare şi rezultatele sunt comparate între ele. În general, cu cît redundanţa unui sistem este mai mare, cu atît poate detecta sau tolera mai multe erori.

Redundanţa temporală constă în folosirea aceluiaşi dispozitiv pentru a calcula acelaşi lucru în mod repetat, după care rezultatele sunt comparate între ele.

2.3 Erori tranziente şi erori permanente

Putem clasifica defectele în două mari categorii:

Erori tranziente sunt erori care se manifestă printr-o malfuncţie temporară a unei componente, dar nu prin defectarea ei definitivă. În sistemele de calcul contemporane, cea mai mare parte a erorilor sunt tranziente.

Erori permanente: se produc la un moment dat şi persistă pînă cînd sistemul este reparat. În această categorie includem şi defectele din fabricaţie.

După cum vă puteţi imagina, redundanţa temporală poate fi folosită numai pentru a tolera defecte tranziente. Pentru a tolera efecte permanente trebuie să avem o formă de redundanţă spaţială.

2.4 Costul fiabilităţii; sisteme echilibrate

Cînd proiectăm un sistem complex este foarte important să "echilibrăm" fiabilitatea părţilor. De exemplu, dacă memoria unui sistem are o fiabilitate mult mai mare decît procesorul, atunci sistemul se va defecta cel mai adesea cu probleme de procesor. Faptul că memoria este de foarte bună calitate nu ne ajută cu nimic; dimpotrivă, probabil că am plătit un preţ mai mare pentru memorie decît ar fi fost strict necesar. În general, o componentă este "destul de bună" dacă nu are cea mai mare probabilitate de defectare.

Întotdeauna cînd discutăm despre fiabilitate trebuie să socotim nu numai costul componentelor fiabile, ci şi costul întreţinerii sistemului. Cît pierdem pe oră atunci cînd sistemul nu funcţionează? Dacă cumpărăm componente foarte fiabile plătim prea mult pentru construcţia sistemului, dar dacă cumpărăm componente cu fiabilitate prea redusă, ne va costa prea mult întreţinerea sistemului. Numai contextul poate dicta cît de fiabil trebuie să fie un sistem: de exemplu, în aplicaţiile critice descrise mai sus, costul ne-funcţionării sistemului este uriaş, aşa încît are sens să investim în componente extrem de fiabile.

3 Creşterea fiabilităţii

În această secţiune voi discuta cîteva tehnici folosite pentru a construi sisteme de calcul fiabile.

3.1 Evitarea defectelor

Evitarea defectelor este o metodologie idealizată, care presupune că toate componentele sunt perfecte. Pentru că hardware-ul de astăzi are o calitate excepţională, nivelul software în calculatoarele obişnuite adoptă o astfel de viziune idealizată. Programatorii presupun că sistemul pe care se rulează programele lor este lipsit de defecţiuni.

Fiabilitatea excelentă a dispozitivelor hardware este obţinută printr-o combinaţie de tehnici, cum ar fi felurite forme de redundanţă, proiectare şi fabricaţie cu precizie foarte ridicată, şi o fază agresivă de testare şi "ardere" (burn-in).

Empiric s-a observat că sistemele tind să aibă o mortalitate care urmăreşte o curbă numită albie, ilustrată în figura 1: sistemele foarte tinere şi cele foarte uzate se strică mult des decît sistemele "mature". "Burn-in" este o fază de testare care foloseşte componentele pînă devin mature; în acest fel, componentele cu mortalitate infantilă ridicată sunt eliminate.

În plus fabricanţii proiectează şi testează sisteme de calcul în condiţii mai nefavorabile decît cele specificate. De exemplu, pe acest fapt se bazează cei care fac "overclocking": specificaţiile unui procesor indică frecvenţa de ceas la care acesta poate opera. Dar în mod frecvent un procesor cu specificaţie de ceas de 1Ghz poate opera la 1.2Ghz, datorită marginilor de toleranţă din fabricaţie.

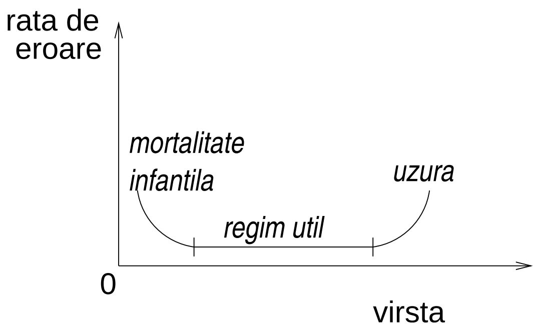

Figura 1: *Graficul ratei de eroare a unui dispozitiv în funcţie de vîrsta sa are adesea forma de albie: dispozitivele foarte noi şi foarte vechi au o probabilitate mai mare de a se defecta.*

Metoda evitării defectelor este cu adevărat extremă. În restul acestui text vom vedea metode mai pesimiste, care recunosc că fiecare din componente se poate defecta şi încearcă se menţină sistemul în funcţionare; această paradigmă se numeşte "toleranţa defectelor" (fault-tolerance).

3.2 Scheme de votare

O metodă foarte simplă dar scumpă de a tolera erori este de a multiplica fiecare componentă. De exemplu, dacă duplicăm întreg sistemul de calcul, apariţia unui defect poate fi detectată comparînd rezultatele celor două subsisteme.

Prima schemă de toleranţă a defectelor a fost propusă de John von Neumann în 1956 şi se numeşte "redundanţă modulară triplă" (Triple Modular Redundancy). În această schemă trei module fac aceeaşi operaţie şi un modul de "vot" alege rezultatul majoritar. Dacă fiecare componentă are o fiabilitate de peste 50%, fiabilitatea ansamblului este mai mare decît a componentelor. Există şi scheme în care sistemul de votare este replicat, pentru a nu depinde de o singură componentă.

Un astfel de sistem de votare este folosit în calculatoarele care controlează naveta spaţială: sistemul este compus din cinci calculatoare, din care patru fac aceleaşi calcule şi al cincilea este folosit pentru operaţiuni ne-critice. Rezultatele celor patru calculatoare se duc pînă la sistemele controlate (motoare), care calculează local rezultatul votului. În plus, fiecare calculator compară rezultatele cu celelalte trei; cînd unul dintre ele dă rezultate diferite este scos din funcţiune.

Dacă două calculatoare se defectează sistemul intră într-un mod de funcţionare în care rezultatele sunt comparate şi recalculate cînd diferă. Al cincilea calculator conţine un sistem de control complet separat, dezvoltat de altă companie, care intră în funcţiune cînd un bug identic este detectat în celelalte patru programe (această metodă va fi prezentată în secţiunea 5.0.2).

3.3 Procesorul IBM G5 din sistemul S/390

În această secţiune vom discuta un tip de redundanţă hibridă, care foloseşte redundanţa spaţială pentru a detecta erori tranzitorii şi redundanţa temporală pentru a le remedia. Acest sistem este asemănător cu modul de funcţionare cu două defecţiuni folosit de naveta spaţială, descris mai sus. Această tehnică e utilizată în multe sisteme comerciale, dar noi vom discuta microprocesorul G5 folosit de compania IBM în calculatoarele sale mainframe.

Microprocesorul G5 conţine două benzi de execuţie identice, care sunt controlate de acelaşi ceas. Toate instrucţiunile sunt executate în mod sincron de ambele benzi, iar la sfîrşitul benzii rezultatele sunt comparate. Dacă rezultatele sunt identice, rezultatul instrucţiunii este scris în registrul destinaţie sau în memorie. Dacă nu, se generează o excepţie software, care de obicei se soldează cu re-execuţia instrucţiunii-problemă. Erorile tranziente sunt astfel reparate în mod transparent. Această schemă este funcţională pentru că probabilitatea ca o eroare tranzientă să afecteze ambele benzi în acelaşi fel este foarte foarte mică.

Calculatorul S/390 foloseşte redundanţa în multe alte feluri: toate sursele de curent, sistemele de răcire, discurile, etc., sunt duplicate. Sistemul nu are un singur punct critic. Acesta este deci un sistem echilibrat: ce sens ar avea un microprocesor deosebit de fiabil dacă sursa de curent se poate arde adesea?

3.4 Un procesor superscalar tolerant la erori tranziente

O schemă foarte originală care foloseşte doar redundanţă temporală pentru a tolera erori tranziente a fost propusă anul acesta la conferinţa de microarhitectură MICRO 2001 de un grup de cercetători de la universitatea Carnegie Mellon. În această schemă, unui procesor superscalar obişnuit i se fac cîteva modificări simple, astfel încît fiecare instrucţiune citită să fie lansată în execuţie în mod repetat. Mecanismele de redenumire a regiştrilor folosite în procesorul superscalar fac executarea unor instrucţiuni suplimentare, care nu afectează sistemul, un lucru foarte simplu. La sfîrşitul benzii de asamblare rezultatele copiilor lansate în execuţie sunt comparate între ele. Robusteţea depinde de gradul de redundanţă: dacă fiecare instrucţiune este executată de două ori, o eroare se manifestă prin rezultate diferite şi instrucţiunea trebuie re-executată; dacă o instrucţiune este executată de mai mult de două ori, se poate folosi o schemă de votare cu majoritate.

Un astfel de procesor poate fi proiectat să lucreze fie în mod normal, fie în mod cu fiabilitate crescută, depinzînd de tipul de program executat. Performanţa în modul cu fiabilitate ridicată este invers proporţională cu gradul de redundanţă; de exemplu, dacă fiecare instrucţiune este executată de două ori, ne-am aştepta la o scădere a vitezei de calcul la 50%. În realitate, penalizarea este ceva mai mică, din cauză că un program nu foloseşte toate resursele computaţionale. De exemplu, dacă un program foloseşte 80% din resurse, cînd executăm programul duplicînd fiecare instrucţiune avem nevoie de 160% resurse, ceea ce se traduce într-o degradare a performanţei cu 37,5% (100/160 = 62,5 = 100 - 37,5).

3.5 DIVA

O schemă deosebit de interesantă de redundantă spaţială a fost propusă în aceeaşi conferinţă în 1999 de către Todd Austin, profesor la universitatea Michigan. Acest proiect e numit DIVA, de la Dynamic Implementation Verification Architecture: arhitectură cu verificare dinamică.

Spre deosebire de schemele anterioare, DIVA e proiectată pentru a tolera atît erori tranziente, cît şi permanente (cele din urmă doar în anumite părţi ale sistemului). Observaţia centrală pe care se bazează DIVA este că e mai uşor de verificat dacă rezultatul unui calcul e corect decît este de efectuat calculul însuşi. Ca atare, arhitectura DIVA este compusă din două procesoare diferite:

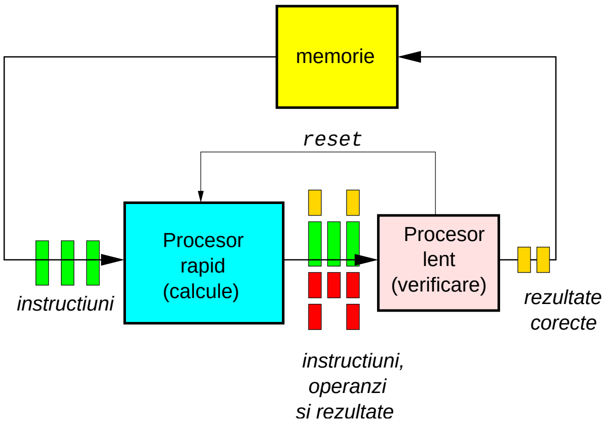

Figura 2: *Procesorul DIVA este compus dintr-un procesor performant care efectuează calculele şi un procesor lent şi foarte robust care le verifică corectitudinea.*

- Un procesor complex, superscalar, foarte optimizat, care face calculele în mod normal;
- Un procesor extrem de simplu, mai lent, dar foarte fiabil, care execută instrucţiunile în ordine, şi este construit folosind tehnici de evitare a defectelor.

Procesorul DIVA funcţionează astfel:

Procesorul complex execută toate instrucţiunile şi calculează rezultatele lor. Rezultatele însă nu sunt scrise, ci sunt transmise procesorului simplu.

Procesorul simplu merge ceva mai încet, şi verifică în paralel toate detaliile rezultatelor primite. Deşi acest procesor este mai simplu, are o treabă mai uşoară, şi ca atare poate atinge aceeaşi performanţă ca cel rapid (exprimată în instrucţiuni procesate pe secundă). Cînd verificarea descoperă o eroare, procesorul simplu calculează rezultatul corect şi re-porneşte procesorul complex de la instrucţiunea următoare.

Foarte interesant este că o arhitectură DIVA poate tolera chiar erori de proiectare în procesorul foarte complicat, pentru că acestea sunt detectate şi corectate de procesorul lent şi simplu. Poate fi chiar avantajos ca procesorul rapid să fie proiectat "incorect", dar extrem de rapid, în cazul în care nu produce rezultate eronate prea frecvent. De exemplu, într-un procesor normal foarte multe circuite suplimentare sunt introduse pentru a trata corect cazul programelor care se automodifică. În realitate, practic nici un program modern nu foloseşte această tehnică în mod curent; procesorul rapid fără aceste circuite poate fi făcut mult mai eficient. Corectitudinea va fi asigurată de procesorul lent.

3.6 Coduri detectoare şi corectoare de erori

Am văzut deja mai multe exemple de folosire a redundanţei spaţiale pentru detectarea şi corectarea erorilor. Costul schemelor prezentate mai sus este substanţial: ele cer o replicare identică a unui întreg sistem. De exemplu, redundanţa modulară triplă are o eficienţă de 33%, pentru că hardware-ul este replicat de trei ori.

E interesant de explorat dacă nu putem obţine aceleaşi beneficii cheltuind mai puţine resurse suplimentare. Deschizători de drumuri au fost în această privinţă Claude Shannon şi Richard Hamming, spre sfîrşitul anilor '40. Vom discuta despre metodele propuse de ei pentru a stoca informaţie în mod fiabil.

Să presupunem că vrem să stocăm nişte informaţii codificate în baza doi într-un mod fiabil. Putem atunci de pildă face două copii ale informaţiei. Dar o defecţiune a unui singur bit va face informaţia de nerecuperat, pentru că acel bit va fi diferit în cele două copii, şi nu putem deduce care este valoarea originală. Slăbiciunea acestei metode constă în faptul că biţii stocaţi nu sunt "robuşti": fiecare bit din mesaj este reprezentat în doar doi biţi din cod. Dacă "amestecăm" biţii din mesaj putem face mult mai bine de atît.

Pentru a obţine rezilienţă la erori trebuie să adăugăm redundanţă în cod; astfel, vom codifica n biţi de informaţie folosind $m > n$ biţi de cod. Cu cît m e mai mare ca n, cu atît mai robust va fi codul nostru. Cuvintele de m biţi care reprezintă coduri corecte se numesc "cuvinte de cod" (code words). Observaţi că nu toate cuvintele de m biţi sunt cuvinte de cod, ci numai 2^n dintre ele.

Putem defini *distanţa Hamming* între două şiruri de biţi ca fiind numărul de diferenţe între cele două şiruri. De exemplu, distanţa Hamming dintre 1111 şi 1010 este 2, pentru că cele două şiruri diferă în poziţiile a doua şi a patra. Cea mai mică distanţă Hamming dintre cuvintele unui cod este o măsură foarte bună a robusteţii codului. De exemplu, dacă distanţă Hamming între oricare două cuvinte este mai mare decît 3, atunci o schimbare de 1 bit poate fi întotdeauna corectată: cel mai apropiat cuvînt de cod este cel care a fost modificat de eroare, pentru că toate celelalte cuvinte de cod se vor afla la o distanţă mai mare de 2 de cuvîntul eronat. Astfel, un cod cu distanţă Hamming 3 poate corecta orice eroare de 1 bit, şi poate detecta orice eroare de doi biţi. Un astfel de cod va detecta şi alte erori, de exemplu va detecta unele erori de trei biţi, dar nu orice eroare de trei biţi.

Există efectiv zeci de coduri diferite, fiecare potrivit în alte circumstanţe. Vom prezenta nişte exemple în ceea ce urmează.

3.6.1 Memorii

Dacă aţi cumpărat vreodată memorie pentru PC-ul dumneavoastră, v-aţi lovit desigur de dilema "care tip de memorie trebuie cumpărat". Din punct de vedere al fiabilităţii, există trei tipuri de memorie pe piaţă: memorii neprotejate, memorii cu paritate şi memorii ECC.

Memoriile neprotejate stochează fiecare bit de date în mod separat şi nu oferă nici o protecţie împotriva erorilor. Ca atare sunt cele mai ieftine. Cum însă dimensiunea memoriilor a crescut foarte repede, la ora actuală această soluţie este riscantă, căci probabilitatea ca nici un bit să nu se defecteze este foarte redusă.

Memoriile cu paritate folosesc o metodă foarte simplă pentru a detecta erori de un bit în fiecare octet (şi, în general, erori care schimbă un număr impar de biţi). Pentru fiecare 8 biţi de date aceste memorii stochează un al nouălea bit de paritate, a cărui valoare este calculată astfel încît oricare cuvînt de nouă biţi are un număr par de biţi "1" (de aici şi numele schemei).

Cînd hardware-ul accesează memoria, automat verifică şi paritatea. Dacă paritatea nu este corectă se declanşează o excepţie şi sistemul de operare decide cum trebuie să acţioneze. O soluţie este de a omorî programul care folosea acea memorie şi de a marca memoria ca fiind defectă, astfel încît alte programe să nu o poată refolosi. Verificarea parităţii este o operaţie foarte rapidă, care se poate face foarte simplu în hardware în paralel cu transferul informaţiei.

Memoriile ECC sunt protejate cu un cod sofisticat de corecţie a erorilor (Error Corecting Code). Acest cod poate corecta automat orice eroare de 1 bit care apare într-un cuvînt de 64 de biţi. Pentru acest scop memoria stochează fiecare cuvînt de 64 de biţi folosind cuvinte de cod de 72 de biţi. Observaţi că "risipa" (overhead) acestei scheme este aceeaşi cu cea a parităţii (9/8 = 72/64); această schemă oferă corecţie cu o robusteţe mai mică, pentru că poate corecta o eroare la 64 de biţi, spre deosebire de cealaltă schemă care poate detecta o eroare la 8 biţi.

La fiecare acces la memorie hardware-ul verifică dacă cuvîntul de cod este corect; dacă nu automat calculează cel mai apropiat cuvînt de cod pe care apoi îl decodifică. Aceste operaţii sunt destul de complicate, astfel încît un sistem cu memorii ECC merge cu aproximativ 5% mai lent decît unul cu memorii cu paritate.

3.6.2 Discuri

Cel mai comun suport persistent de informaţie este discul, în multiplele lui încarnări: hard disc, dischetă, disc optic, compact-disc, etc. Informaţiile din această secţiune sunt valabile pentru multe din aceste tipuri de discuri.

Discurile folosesc simultan două metode diferite de redundanţă spaţială; o protecţie sporită este necesară din cauză că discurile funcţionează într-un mediu mult mai aspru decît memoriile: discurile au părţi mecanice în mişcare, care se uzează şi se pot strica mai uşor.

Informaţia este stocată pe disc în *sectoare*. Un sector este relativ mare (comparat cu un cuvînt de memorie), fiind de ordinul a jumătate de kilooctet (512 octeţi). Discurile folosesc sectoare mari pentru că la viteza lor de rotaţie (peste 5000 de revoluţii pe minut) capetele de citire/scriere nu se pot plasa foarte precis pe suprafaţă. Astfel, unitatea elementară în care se scrie pe un disc este sectorul: chiar dacă vrem să modificăm un singur bit, trebuie să rescriem tot sectorul.

Codurile folosite pentru discuri se numesc CRC, de la Cyclic Redundancy Check: coduri ciclice. Un cuvînt de cod constă din chiar cuvîntul de date urmat de informaţii de control; decodificarea codurilor CRC este foarte simplă: se extrage direct cuvîntul de date. Codul de control verifică dacă vreunul din biţii stocaţi e incorect. Un cod ciclic are proprietatea că orice permutare a datelor este protejată de acelaşi cuvînt de control.

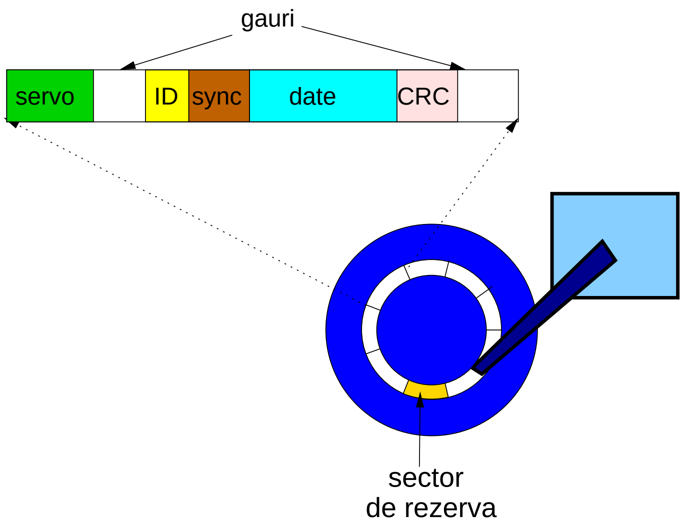

Figura 3: *Formatul unui sector de disc: informaţia servo este folosită pentru controlul mişcării capului, identificatorul indică numărul sectorului curent, informaţia de sincronizare este folosită pentru a sincroniza poziţia capului cu începutul sectorului; datele sunt stocate într-un şir compact, codificate folosind un cod detector de erori. Între două sectoare consecutive este o gaură, care-i dă capului ceva libertate cînd rescrie sectorul (niciodată nu va rescrie începînd chiar din acelaşi loc).*

Cînd codul de control indică defectarea unui sector, discurile folosesc în mod automat a doua formă de redundanţă spaţială: sectoare de rezervă. Pe disc sunt ascunse sectoare invizibile, care sunt folosite atunci cînd sectoarele de date încep sa dea rateuri. În mod transparent software-ul alocă un sector de rezervă în locul unuia defect. Identificatorul de sector este folosit pentru a indica cine pe cine înlocuieşte.

Discurile stochează o *hartă de defecte* care indică sectoarele înlocuite; asta le permite să funcţioneze şi după ce apar defecţiuni, şi permite de asemenea un proces de fabricaţie mai imperfect şi mai ieftin.

3.6.3 Turbo-coduri

Codurile detectoare şi corectoare de erori sunt folosite pe larg în reţelele de calculatoare. Am scris cu alte ocazii în PC Report despre acest subiect, astfel încît în această secţiune voi discuta despre un caz extrem.

Depinzînd de caracteristicile canalului de comunicaţii (distanţă, cost de transmisiune, viteza semnalului, zgomot) se pot folosi coduri mai mult sau mai puţin robuste. În anumite cazuri e preferabil ca erorile să fie detectate şi datele incorecte să fie retransmise, în alte cazuri costul retransmisiei este prea mare, şi ca atare se folosesc coduri corectoare. Folosirea unui cod corector în transmisiunea de date se mai numeşte şi codare preventivă (Forward Error Correction).

Pentru comunicaţia cu sondele spaţiale se folosesc coduri corectoare de erori extrem de robuste, pentru că la astfel de distanţe semnalul electromagnetic are nevoie de multe minute pentru a se propaga. În 1993 un grup de cercetători francezi a inventat o clasă de coduri extrem de robuste, numite Turbo-coduri, care folosind o redundanţă relativ redusă, de 200% obţin o rezilienţă excepţională la zgomot, fiind foarte aproape de limitele maxime teoretice.

Turbo-codurile ilustrează un nou tip de compromis pe care arhitectul îl poate face în ecuaţia robusteţe/cost: costul cel mare al unui turbo-cod nu este în cantitatea mare de informaţie suplimentară, ci în algoritmul de decodificare, care este foarte complicat şi necesită multe iteraţii. După cum am văzut şi în cazul memoriilor, cu aceeaşi redundanţă putem obţine garanţii diferite de fiabilitate, în funcţie de algoritmul de codificare folosit. În cazul comunicaţiei interplanetare costul transmisiunii face costul decodificării insignifiant, deci turbo-codurile sunt potrivite.

3.7 RAID

În această secţiune voi ilustra un alt tip de sistem fiabil redundant, care, spre deosebire de alte soluţii prezentate, are o performanţă mai bună decît sistemul de bază. În plus, acest sistem adaugă o dimensiune nouă în spaţiul opţiunilor fiabilităţii, şi anume capacitatea de a fi reparat în timp ce funcţionează (maintainability).

RAID este o prescurtare de la Redundant Array of Inexpensive Disks, sau set redundant de discuri ieftine. Ideea a fost introdusă în 1987 de cercetători de la universitatea Berkeley din California, şi la ora actuală este obiectul unei industrii anuale de 12 miliarde de dolari.

Ideea centrală în RAID este de a stoca informaţie pe mai multe discuri simultan. Informaţia este codificată redundant, astfel încît să poată fi recuperată dacă oricare din discuri se defectează. Această proprietate este foarte utilă pentru sisteme care trebuie să funcţioneze în foc continuu.

Observaţi că tipul de defecţiune pe care RAID o adresează este diferit de cel descris în secţiunea 3.6.2: aici vrem să operăm cînd un disc este complet distrus, acolo ne interesa să detectăm alteraţii ale informaţiei stocate. Într-un sistem RAID toate aceste tehnici operează simultan: fiecare disc foloseşte coduri CRC şi sectoare de rezervă, iar sistemul RAID foloseşte stocare a informaţiei redundantă.

Există mai multe tipuri de sisteme RAID, dar aici vom discuta unul singur, în care informaţia este scrisă pe 5 discuri, din care 4 conţin date şi unul paritate.

Un astfel de sistem RAID se poate afla într-unul din trei moduri de funcţionare:

Funcţionare normală: operaţiile de citire extrag date de pe cele patru discuri cu date. O operaţie de scriere însă strînge patru blocuri de informaţie şi calculează un al cincilea bloc de paritate; fiecare bloc este stocat pe alt disc. Acest mod de scriere se numeşte "striping", adică "feliere", pentru că datele sunt scrise în paralel, cîte o felie pe fiecare disc.

Funcţionarea degradată: este începută cînd un disc se defectează. Atunci citirile şi scrierile de pe discul stricat trebuie să acceseze celelalte patru discuri şi să calculeze informaţia lipsă. Avantajul parităţii este că oricare din biţii lipsă poate fi recalculat ca paritatea celorlalţi patru biţi.

Reconstrucţia: este începută cînd un disc defect este înlocuit. Un proces secundar recalculează informaţia lipsă şi o scrie pe noul disc.

[Restul acestui articol va fi publicat în numărul din februarie]

4 Rezumat

În numărul anterior din Net Report am publicat prima parte a unui articol despre fiabilitatea sistemelor de calcul. Iată aici rezumatul ideilor importante:

- Fiabilitatea este definită ca fiind probabilitatea ca un sistem să se defecteze într-o anumită perioadă de timp şi în anumite condiţii de mediu;
- Sistemele de calcul sunt construite din succesiuni de nivele abstracte, care au diferite fiabilităţi;
- Fiabilitatea unui mecanism poate fi sporită folosind redundanţa spaţială sau temporală; redundanţa temporală poate tolera efectul unor erori tranziente, pe cînd redundanţa spaţială poate ascunde şi erori permanente;
- Un sistem de calcul trebuie să fie echilibrat din punct de vedere al fiabilităţii: toate nivelele trebuie să fie comparabile, altfel cel mai nefiabil nivel este veriga cea mai slabă a lanţului;
- Codurile detectoare şi corectoare de erori pot descoperi şi repara stricăciuni ale datelor stocate sau transmise folosind grade relativ reduse de redundanţă.

5 Fiabilitatea programelor

Ne-am putea aştepta, în mod naiv, ca, spre deosebire de hardware, software-ul să nu aibă nici un fel de probleme de fiabilitate: în definitiv programele nu se uzează, şi sunt executate într-un mediu foarte specializat; în plus, programele sunt obiecte deterministe, deci ar trebui să se comporte de fiecare dată în acelaşi fel cînd procesează aceleaşi date de intrare. Cu toate acestea, de fapt fiabilitatea programelor este mult mai scăzută decît a sistemelor hardware; este potrivit să modelăm deci programele ca sisteme cu fiabilitate imperfectă. În această secţiune discutăm în mod superficial unele dintre motivele lipsei de fiabilitate a programelor şi menţionăm unele tehnici care pot fi folosite pentru a "întări" programele.

Cea mai importantă cauză a malfuncţiilor programelor sunt bug-urile, adică implementări incorecte. Chiar şi programatori foarte pricepuţi produc programe cu defecte. Nu exagerăm deci dacă afirmăm că nu există nici un program substanţial fără bug-uri, aşa cum nu există vreo carte tipărită care să nu aibă erori tipografice. Complexitatea componentelor software este pur şi simplu prea mare pentru a putea fi stăpînită de către oameni, şi cu tot progresul în tehnici de programare, cum ar fi descompunerea programelor în module mici, folosirea unor limbaje de programare evoluate şi a unor scule complexe pentru dezvoltarea, testarea şi analiza programelor, rezultatele sunt încă foarte departe de perfecţiune, şi productivitatea programatorilor nu a crescut substanţial în ultimele două decenii.

Cel mai adesea problemele rezolvate în software sunt atît de complicate încît nici nu pot fi specificate în mod precis. În consecinţă programatorii întîlnesc tot felul de incertitudini cînd încearcă să implementeze soluţiile. O cauză fundamentală a lipsei de fiabilitate a programelor este deci specificaţia incompletă şi imprecisă.

Cele mai insidioase defecţiuni software se manifestă numai cu ocazia unor anumite combinaţii de valori pentru datele de intrare sau pentru anumite succesiuni de evenimente externe, care nu au fost prevăzute de programator. Asemenea combinaţii apar cu probabilitate foarte mică în timpul procedurilor normale de testare, deci adesea supravieţuiesc pînă în faza operaţională.

5.0.1 Biţi "putrezi"

A vedea programele software ca pe o entitate monolitică este o aproximare grosolană a realităţii: un program trece prin nenumărate revizii şi îmbunătăţiri, cum pot mărturisi cei care sunt forţaţi periodic să facă "upgrades". Versiunile noi sunt construite pe scheletul celor vechi, reparînd defecţiunile descoperite şi adăugînd noi funcţionalităţi. Cu toate acestea, procesul reparării defecţiunilor introduce adesea noi defecţiuni, pentru că efectele unei reparaţii au uneori consecinţe nebănuite.

Uluitoarea creştere a performanţei hardware-ului este o motivaţie constantă pentru reînnoirea sistemelor software. Pe măsură ce dispozitivele hardware devin mai ieftine şi mai compacte, ele pot fi integrate în dispozitive electronice mai "deştepte". Toate aceste noi dispozitive au nevoie de nou software care să le manipuleze. Pe măsură ce costul dispozitivelor de stocare a informaţiei scade, din ce în ce mai complexe şi bogate tipuri de informaţie pot fi stocate şi prelucrate. De exemplu, imagini şi muzică sunt tipuri curent manipulate de PC-urile contemporane, şi capacitatea lor de prelucrare a devenit de curînd suficient de puternică pentru a manipula în mod interactiv filme.

Un fenomen legat de acest ciclu permanent de înnoiri este "putrezirea biţilor" (bit rot). Acest fenomen se manifestă pe două planuri: datele stocate cu mult timp în urmă nu mai pot fi folosite în noile sisteme de calcul, pentru că dispozitivele periferice învechite nu mai sunt suportate de fabricanţi, şi programe vechi, care mergeau foarte bine, încep să manifeste erori. "Boala" programelor este legată de mediul în care programele se execută, şi care este în continuă schimbare. De exemplu, multe programe vechi făceau anumite presupuneri despre cît de mari vor fi seturile de date pe care le vor prelucra. Cea mai faimoasă astfel de presupunere este cea care a cauzat bug-ul Y2K: programatorii din anii '60 au presupus că programele lor nu vor manipula niciodată date calendaristice al căror an nu va începe cu cifrele 19. Chiar dacă Y2K a făcut mai mult zgomot decît pagube, astfel de presupuneri se întîlnesc la tot pasul în programele de astăzi. De exemplu, auzim adesea despre dificultatea de a transporta programe de la procesoare pe 32 de biţi la procesoare pe 64 de biţi. Din moment ce orice valoare pe 32 de biţi se poate reprezenta exact atunci cînd folosim 64 de biţi, teoretic nu ar trebui să fie nici o problemă, şi vechile programe ar trebui să funcţioneze corect. În realitate multe programe depind în feluri subtile de precizia datelor pe care le manipulează. Cînd un astfel de program este mutat pe o platformă nouă toate aceste dependinţe se transformă în bug-uri.

5.0.2 Programare cu N versiuni

Domeniul ingineriei programelor (software engineering) se ocupă de metode prin care se poate cuantifica şi îmbunătăţi calitatea programelor. Una dintre soluţiile studiate este foarte înrudită cu tehnicile de votare folosite pentru toleranţa erorilor hardware, care au fost prezentate în prima parte a acestui text. Numele acestei soluţii în lumea software este "programare cu N versiuni". Votarea foloseşte redundanţă spaţială: dispozitivul de calcul este replicat de N ori şi rezultatul final este obţinut prin votul majoritar al rezultatelor individuale.

Bug-urile software sunt persistente: aflat în aceleaşi condiţii programul se va comporta în acelaşi fel. Tehnicile de votare sunt neputincioase dacă toate componentele fac aceeaşi eroare în acelaşi timp. Votarea este utilă pentru tratamentul erorilor tranziente. Programarea cu N versiuni se face deci prin executarea în paralel a N programe diferite, scrise de echipe diferite de programatori, dacă e posibil, folosind scule şi tehnologii diferite. Toate cele N programe rezolvă aceeaşi problemă, dar în moduri diferite. Numai folosind o astfel de strategie tehnica votării poate funcţiona în cazul programelor.

Specificaţii imprecise ale problemei pot fi detectate cu uşurinţă de această tehnică, pentru că implementările diferite pot lua decizii diferite pentru cazurile nespecificate clar. Din nefericire, programarea cu N versiuni este o metodologie foarte scumpă, folosită numai pentru aplicaţii critice, unde siguranţa este fundamentală.

5.0.3 Întinerirea programelor

Diferenţa fundamentală între hardware şi software este că un program poate avea o stare internă arbitrar de complicată. În general, dispozitivele hardware pot fi aproximate ca fiind automate finite (adică spaţiul stărilor în care se pot afla, chiar dacă este foarte mare, este totuşi finit). Chiar şi cele mai simple programe au un spaţiu de stări infinit, mai exact, nu putem pune nici o limită arbitrară dimensiunii spaţiului lor.

Această diferenţă este foarte importantă şi din punct de vedere teoretic: foarte multe proprietăţi interesante ale automatelor finite sunt *decidabile*, adică se pot scrie algoritmi care, atunci cînd primesc descrierea unui automat finit, pot răspunde în mod exact la întrebări legate de orice evoluţie viitoare a automatului. Din păcate, aceleaşi întrebări pentru un sistem cu stare infinită sunt adesea nedecidabile. Într-adevăr, matematicienii au arătat în anii '30 că foarte multe dintre proprietăţile unui sistem software în general nu pot fi calculate de un alt sistem software.

O consecinţă practică a dimensiunii infinite a spaţiului de stări al programelor este că, pe măsură ce un program se execută mai mult timp, cu atît mai complicată poate deveni starea sa internă. Dacă un program nu îşi întrerupe execuţia, chiar dacă va primi aceleaşi date la intrare, ar putea calcula un răspuns diferit. Un bug în program poate corupe starea internă, dar efectele acestei stricăciuni pot deveni vizibile mult mai tîrziu în execuţia programului, cînd programul ia o decizie bazată pe elementele de stare incorecte. Un tip faimos de problemă, în mod normal benignă, asociată cu programele care se execută un timp îndelungat, este scurgerea de memorie (memory leak).

Adesea programele alocă spaţiu temporar de memorie, pe care îl eliberează după ce au terminat calculele care aveau nevoie de el. Dacă programatorul uită să elibereze această memorie se spune că memoria se scurge (leak). Aceasta este o eroare frecvent întîlnită în programare, relativ greu de descoperit. În mod normal o astfel de eroare nu afectează *corectitudinea* programului: rezultatele produse la final sunt corecte. Cînd programul îşi termină execuţia, sistemul de operare recuperează automat memoria scursă. În cazul programelor care se execută timp îndelungat, cum ar fi sistemele de operare sau servere de web, dacă o scurgere se întîlneşte în interiorul unei bucle, cu timpul memoria pierdută va creşte pînă cînd toată memoria sistemului este pierdută. În astfel de cazuri de obicei sistemul îşi încetează execuţia sau funcţionarea sa devine extrem de lentă, din cauză că resursele rămase sunt insuficiente.

Utilizatorii sistemului de operare Windows de la Microsoft au descoperit şi o soluţie pentru această problemă: reboot-area calculatorului. Numele ştiinţific pentru această soluţie este "reîntinerirea programelor" (software rejuvenation). Reîntinerirea este cauzată de repornirea periodică a programelor. Repornirea cauzează iniţializarea stării interne la o aceeaşi valoare iniţială. Tehnica aceasta este aplicabilă numai dacă starea internă a programului nu este importantă şi poate fi pierdută; altfel, întinerirea trebuie să fie combinată cu "checkpoint"-uri. Un checkpoint salvează informaţia importantă pe un mediu de memorie persistent, şi o restaurează după ce programul este repornit.

Reîntinerirea se aplică cu precădere programelor de tip server, care execută tot timpul o buclă, acceptînd cereri de la clienţi şi răspunzîndu-le. Multe servere sunt lipsite de stare (stateless), adică nu păstrează nici un fel de informaţii despre o tranzacţie cu un client după ce tranzacţia s-a consumat. Reîntinerirea este eficace dacă costul repornirilor periodice este mai redus decît costul repornirii după o cădere catastrofică, care poate să implice o procedură sofisticată de recuperare a datelor pierdute. Reîntinerirea este de asemenea folosită cu succes cînd serverele care oferă serviciul au rezerve, astfel încît servere de rezervă pot răspunde clienţilor în timp ce altele se reiniţializează.

5.1 Verificare formală

Verificarea formală este un nume generic pentru o serie întreagă de tehnici sofisticate care certifică corectitudinea, mai ales a sistemelor hardware, dar în ultima vreme şi a unor sisteme software. Verificarea formală se ocupă de descoperirea şi eliminarea bug-urilor, şi în acest sens este o tehnică de creştere a fiabilităţii programelor.

Cheia metodelor de verificare formală este specificarea foarte precisă a comportării componentelor sistemului de analizat (folosind formule matematice) şi verificarea automată a proprietăţilor sistemului în întregime. Dacă ştim cum este construit sistemul, şi dacă ştim comportarea fiecăreia din componente, putem raţiona despre comportarea ansamblului. Raţionamentele pot fi făcute foarte precise folosind diferite variante de logici matematice. Fiecare raţionament este o serie de derivări, în care din fapte ştiute ca fiind adevărate deducem alte adevăruri. Verificarea formală studiază aceste derivări, şi verifică faptul că sunt corecte.

Două aspecte fac verificarea formală o tehnică foarte puternică: (1) calculele minuţioase sunt efectuate de către calculatoare, a căror atenţie nu oboseşte niciodată, (2) certitudinea nu vine din faptul că demonstrăm ceva, ci din faptul că putem verifica dacă demonstraţia este corectă! Am menţionat şi în prima parte a acestui articol, cînd am descris sistemul DIVA, că a verifica corectitudinea unui rezultat este mult mai simplu decît a demonstra rezultatul însuşi. Acest fapt este extrem de folositor în contextul verificării formale, în care programul care face demonstraţiile este extrem de complicat, şi ca atare poate conţine erori (ca orice alt program complex) şi deci genera demonstraţii eronate. Un program care verifică dacă o demonstraţie este corectă însă este mult mai simplu, şi ca atare ne oferă mai multă încredere.

6 Expunerea defectelor

În secţiunile din numărul anterior al revistei, şi în cele de mai sus, despre fiabilitatea programelor, am discutat despre tehnici care pot fi folosite în arhitectura calculatoarelor pentru a mări fiabilitatea pe care un nivel arhitectural o expune nivelelor superioare. Am observat că, cel mai adesea, nivelul hardware este perceput de nivelele superioare ca fiind "perfect", fără defecte. Am văzut o mulţime de mecanisme folosite în construcţia hardware-ului pentru a oferi această iluzie.

În această secţiune voi discuta o metodă complet diferită, folosită cu mult succes în construcţia a două sisteme complexe. Această tehnică pleacă de la asumpţia că nivelul inferior este inerent nefiabil, şi că în loc de a consuma resurse pentru a-i mări fiabilitatea, e preferabil să *expunem* lipsa sa de fiabilitate nivelelor superioare.

Raţionamentul din spatele acestei decizii aparent paradoxale se bazează pe mai multe argumente:

- Costul fiabilităţii poate fi prea mare: dacă fiecare nivel introduce un cost suplimentar pentru a oferi fiabilitate, costul total al nivelelor suprapuse creşte exponenţial cu numărul de nivele. Pentru anumite situaţii acest cost poate deveni prohibitiv.
- Nivelele superioare au propriile lor scheme de toleranţă la defecte. Dacă schemele nivelelor superioare sunt suficient de eficace, ele pot tolera şi defecţiunile nivelelor inferioare. De exemplu, folosind coduri corectoare de erori foarte eficace (discutate în episodul anterior) între două noduri la distanţă într-o reţea de comunicaţii poate face nenecesară transmisiunea fiabilă pe fiecare segment al reţelei.
- În unele cazuri, nivelele superioare nu au nevoie decît de o formă redusă de fiabilitate, şi nu de una perfectă. De exemplu, în unele tipuri de aplicaţii interactive, cum ar fi telefonia, e preferabil ca datele să vină repede şi distorsionate decît să fie livrate cu întîrziere şi impecabile. Date eronate cauzează mici zgomote la recepţie, care sunt tolerate de către urechea umană; sosirea cu întîrziere a semnalului sonor este însă foarte puţin folositoare pentru a crea iluzia unei conversaţii interactive, şi este mult mai perceptibilă.

6.1 Internetul

Una dintre cele mai uimitoare tehnologii a secolului douăzeci este cu siguranţă Internetul. Acesta este o reţea de calculatoare, proiectată iniţial pentru a conecta reţele militare de calculatoare şi de a le permite să opereze chiar şi în condiţiile distrugerii unui mare număr de echipamente din reţea, în cazul unei conflagraţii nucleare. Internetul a evoluat într-o reţea comercială care acoperă toate continentele, cu mai mult de 125 de milioane de calculatoare şi peste 1 miliard de utilizatori.

Internetul nu este prima reţea de dimensiune globală; cu mai mult de un secol înainte de crearea Internetului a apărut telefonul; reţelele telefonice au cu siguranţă întîietatea în acoperirea planetei. Ne-am aştepta ca proiectanţii Internetului să fi folosit multe din tehnologiile folosite în construcţia reţelelor telefonice, despre care există o cantitate mare de informaţii şi o experienţă substanţială. În realitate, nimic nu poate fi mai departe de adevăr: arhitectura Internetului pare a fi în mod deliberat opusă reţelei de telefonie. Nicăieri nu se vede mai bine diferenţa dintre cele două reţele decît în felul în care tratează fiabilitatea.

Reţeaua telefonică a fost proiectată dintru început pentru o fiabilitate excepţională. O centrală telefonică trebuie să însumeze mai puţin de trei minute de indisponibilitate în fiecare an. Numai în circumstanţe absolut excepţionale o conversaţie iniţiată poate fi întreruptă datorită unor probleme din reţea. Reţeaua telefonică va permite stabilirea unei legături numai după ce a rezervat toate resursele necesare pentru transmisiunea promptă a semnalelor vocale pe întregul traseu dintre cele două puncte care comunică. Standarde stricte dictează cît de mult timp poate dura faza de construcţie a legăturii; dacă toate resursele nu pot fi obţinute utilizatorul primeşte un ton de ocupat. Capacitatea reţelei este planificată atent pe baza unor statistici detaliate despre comportarea vorbitorilor astfel încît în condiţiuni normale şansa obţinerii unui ton de ocupat din cauza resurselor insuficiente din reţea să fie extrem de redusă.

Un factor crucial care garantează calitatea conexiunilor telefonice este prealocarea tuturor resurselor necesare înainte ca legătura să fie stabilită. Pornind de la numărul format, prima centrală telefonică calculează o secvenţă de centrale prin care semnalul trebuie să treacă pentru a lega apelantul cu apelatul; acest calcul se bazează pe tabele de rutare pre-calculate cu mare grijă şi stabilite de către proiectanţii reţelei. Fiecare centrală negociază apoi cu cea succesivă folosind un protocol sofisticat de semnalizare, şi alocă capacitate pentru transportul datelor şi pentru comutarea datelor (care în centrală leagă circuitul de intrare cu cel de ieşire). Cînd toate conexiunile punct-la-punct între centrale sunt stabilite se generează un ton de "sunat". Cînd conversaţia a fost iniţiată, semnalul vocal este eşantionat şi digitizat în prima centrală. Pentru fiecare bit din acest semnal s-a prealocat deja o cuantă periodică de timp pe fiecare din circuitele pe care le va traversa. Biţii sunt transmişi unul cîte unul şi traversează toate trunchiurile în aceeaşi ordine în care au fost generaţi, sosind la destinaţie la timp pentru a fi reasamblaţi şi convertiţi la loc într-un semnal auditiv. Din cauza prealocării, de îndată ce un bit intră în reţea, cu o probabilitate extrem de ridicată el va ajunge la celălalt capăt exact cînd trebuie. Cînd unul dintre vorbitori închide telefonul, protocolul de semnalizare intră din nou într-o fază complicată prin care eliberează toate resursele alocate la momentul apelului.

Internetul are o arhitectură fundamental diferită. Nu numai că nu există garanţii despre timpul necesar pentru a ajunge de la emiţător la receptor, dar nu există nici o garanţie că datele nu sunt pierdute sau modificate în timpul transferului. Utilizatorii Internetului obţin un serviciu extrem de "slab", care poate fi enunţat pe scurt astfel: "tu pui date în reţea şi zici unde vrei să ajungă; reţeaua o să încerce să livreze datele acolo".

Felul în care informaţia circulă în Internet este complet diferit de reţeaua telefonică: datele sunt divizate în pachete care sunt introduse în reţea în ordinea sosirii. Fiecare pachet poate călători pe o rută complet diferită pînă la destinaţie. Unele pachete se pot pierde, alte pot fi duplicate, şi ele pot sosi la destinaţie în altă ordine decît au fost emise, sau chiar sparte în pachete mai mici.

Pachetele sunt plimbate prin Internet de un protocol numit IP, Internet Protocol. IP funcţionează aproximativ astfel: cînd un calculator intermediar primeşte un pachet se uită întîi la adresa destinaţie înscrisă. Apoi el face nişte calcule simple pentru a decide în ce direcţie pachetul trebuie trimis, mai precis, căruia dintre vecinii săi trebuie să-i dea pachetul. Pachetul este apoi trimis vecinului. Dacă la un calculator intermediar pachetele vin mai repede decît apucă să le trimită mai departe, şi dacă nici nu are unde să le stocheze pentru o vreme, are dreptul să le facă pierdute. Aceasta este principala cauză pentru care datele se pot pierde în Internet.

Spre deosebire de reţeaua telefonică, structura Internetului nu este controlată de un număr mic de companii, ci este în continuă schimbare, de la zi la zi şi de la oră la oră, pe măsură ce noi calculatoare se conectează, noi utilizatori sună folosind modemuri, accidente întrerup conectivitatea şi noi linii de transmisiune sunt instalate. Calculatoarele responsabile pentru transmiterea datelor, numite rutere, discută între ele permanent pentru a afla care este forma curentă aproximativă a reţelei. Aceste informaţii sunt utilizate în procesul de decizie care selectează vecin folosit pentru transmisiunea fiecărui pachet spre destinaţie.

Dată fiind această infrastructură, este uimitor că Internetul funcţionează cîtuşi de puţin, şi că informaţia ajunge cîteodată neperturbată la destinaţie. Fiabilitatea aplicaţiilor din Internet este construită pe baza acestui mediu extrem de nefiabil, folosind două ingrediente:

Lipsa de stare (statelessness): ruterele din Internet nu stochează nici un fel de informaţii despre traficul care le parcurge. (Prin contrast, în reţeaua telefonică, comutatoarele ştiu despre fiecare bit care le traversează de unde vine, unde se duce, şi cînd succesorul lui va sosi.) Un ruter primeşte un pachet, calculează vecinul căruia să-i dea pachetul, şi livrează pachetul. Starea internă a ruterului după livrarea pachetului este aceeaşi cu cea de dinainte. Aşa cum am văzut în cazul reîntineririi programelor, lipsa stării interne face mult mai simplă repornirea unui ruter după o defecţiune. O defecţiune nu pierde informaţii vitale, care nu pot fi recuperate prin alte metode. De asemenea, lipsa stării face relativ uşoară sarcina altor rutere de a prelua traficul în cazul defectării unuia.

TCP: este un protocol numit Transmission Control Protocol, care se execută deasupra protocolului IP. Dacă într-o reţea toate nodurile din interior execută IP, numai sursa şi destinaţia execută TCP. TCP este protocolul care construieşte o transmisiune fiabilă: el asigură că toate pachetele trimise ajung la destinaţie, fără lipsuri sau duplicate, în ordinea în care au fost trimise. TCP reuşeşte această performanţă folosind următoarele mecanisme:

- Numerotează pachetele trimise;
- Foloseşte pachete de confirmare pentru a anunţa sosirea datelor la capătul celălalt;
- Foloseşte alarme pentru a detecta pachetele care nu sunt confirmate pentru un timp îndelungat;
- Foloseşte retransmisii pentru a retrimite pachetele care se pierd în reţea.

Din cauză că pachetele cu confirmări se pot pierde la rîndul lor, unele pachete sunt injectate în mod repetat în reţea, ceea ce poate duce la livrarea unor duplicate; TCP trebuie le elimină folosind numerele de serie ale pachetelor.

Întregul Internet este construit pe nucleul nefiabil oferit de IP: nu numai datele şi confirmările sunt trimise în mod nefiabil, dar chiar şi mesajele de control schimbate între rutere, prin care află despre schimbările din topologia reţelei, şi traficul folosit pentru monitorizarea şi mentenanţa reţelei folosesc aceleaşi mecanisme nefiabile de transmisiune.

În pofida structurii sale aparent şubrede, Internetul este un competitor formidabil al altor forme de distribuţie a informaţiei: radio, televiziune şi telefonie. Costul transmisiunii vocii prin Internet este mult mai scăzut decît folosind reţelele specializate de telefonie. Multe companii importante de telefonie investesc în mod serios în echipamente care vor transporta voce peste protocolul IP.

De ce are Internetul atîta succes, în pofida lipsei sale de fiabilitate?

Răspunsul constă, în parte, în faptul că Internetul nu oferă un serviciu de fiabilitate costisitor de care poate aplicaţiile nu au nevoie. Reţeaua de telefonie face un efort substanţial pentru a oferi un serviciu de o fiabilitate foarte ridicată, dar sistemul folosit este foarte inflexibil şi consumă o cantitate uriaşă de resurse. Practic, fiabilitatea reţelei de telefoane *costă prea mult*.

Internetul mută problema fiabilităţii la un nivel superior, de la IP la TCP. TCP oferă o fiabilitate perfect adecvată pentru multe aplicaţii. TCP este executat numai de către calculatoarele terminale implicate în comunicaţie, şi nu de către rutere. Ca atare algoritmii complicaţi folosiţi de acest protocol nu taxează resursele reţelei, care scalează în mod natural la dimensiuni globale.

Mai mult: aplicaţii care nu au nevoie de livrarea fiabilă a datelor a lui TCP nu sunt obligate să folosească acest protocol. De exemplu, protocoalele folosite pentru posturile de radio din Internet folosesc coduri puternice de corecţie a erorilor (discutate în prima parte a acestui articol) şi nu au nevoie de retransmisii. Pachete pierdute sau întîrziate sunt pur şi simplu ignorate. Acest lucru este acceptabil pentru că utilizatorul final, omul, tolerează semnale cu zgomot.

6.2 Teramac

În această secţiune voi discuta pe scurt despre Teramac, un sistem de calcul dezvoltat de cercetători de la Hewlett-Packard care demonstrează o metodologie extremă în tratamentul fiabilităţii sistemelor. Acest calculator este construit din componente defecte: mai mult de 70% din circuitele sale componente au o malfuncţie oarecare. Cu toate acestea, sistemul funcţionează minunat şi poate efectua calcule extrem de performante.

Să observăm că arhitectura Teramac tolerează numai defecţiuni permanente, care sunt prezente încă de la fabricaţie. Componentele Teramac sunt circuite hardware de un tip anume, numit hardware reconfigurabil. Înainte de a discuta sistemul Teramac voi face o prezentare succintă a structurii hardware-ului reconfigurabil şi voi explica de ce un sistem fiabil poate fi construit din componente reconfigurabile nefiabile.

6.2.1 Hardware reconfigurabil

Într-o primă aproximaţie, circuitele digitale obişnuite sunt compuse din elemente computaţionale simple, numite porţi logice, conectate între ele prin sîrme. Porţile logice sunt construite din tranzistori. Fiecare poartă logică face calcule pe mai multe valori de 1 bit. Porţile logice sunt universale, în sensul că orice proces calcul poate fi exprimat în termeni de operaţii ale porţilor logice.

În hardware-ul reconfigurabil porţile logice nu au o funcţionalitate fixată, iar sîrmele formează o grilă. Fiecare poartă este configurabilă, adică poate fi forţată să efectueze orice operaţie logică. La fiecare intersecţie de sîrme se află un mic comutator configurabil, care poate fi de asemenea programat să conecteze sîrmele. Configurarea porţilor şi a sîrmelor se face prin semnale electrice. Fiecare poartă şi fiecare comutator are o mică memorie asociată, în care-şi stochează configuraţia. Pentru că schimbînd conţinutul acestor memorii putem schimba funcţionalitatea hardware-ului, circuitele acestea se numesc "reconfigurabile".

Hardware-ul reconfigurabil este echivalent cu cel obişnuit, în sensul că orice circuit poate fi implementat folosind ambele tehnologii. Hardware-ul reconfigurabil tinde însă să fie mai ineficient: memoriile şi porţile configurabile ocupă mult mai mult loc decît porţile obişnuite. Pe de altă parte, semnalele electrice care traversează doar sîrme într-un circuit obişnuit trebuie să treacă printr-o serie de comutatoare în hardware-ul reconfigurabil, ceea ce face circuitele mai lente. Un factor de 10 diferenţă în viteză şi densitate este de aşteptat între hardware obişnuit şi cel configurabil de aceeaşi generaţie.

Pentru a programa un circuit reconfigurabil cu funcţiunea unui circuit obişnuit trebuie să asociem fiecare poartă din circuit cu o poartă configurabilă; acest proces se numeşte "plasare"; de asemenea, fiecare sîrmă trebuie asociată cu succesiuni de segmente legate prin comutatoare, în procesul de "rutare" (figura 4).

6.2.2 Fiabilitatea sistemului Teramac

Calitatea circuitelor reconfigurabile exploatată de Teramac pentru a obţine fiabilitate este faptul că porţile logice configurabile sunt esenţialmente interschimbabile. Cercetătorii proiectului Teramac au dezvoltat un program de plasare care foloseşte o hartă de defecte ale circuitelor reconfigurabile. Această sculă ocoleşte porţiunile inutilizabile şi rutează conexiunile în jurul defectelor, exploatînd doar porţiunile funcţionale ale fiecărui circuit. Cercetătorii au creat şi o serie de scule care descoperă şi cataloghează defectele. Sculele acestea folosesc chiar programabilitatea circuitelor pentru a le

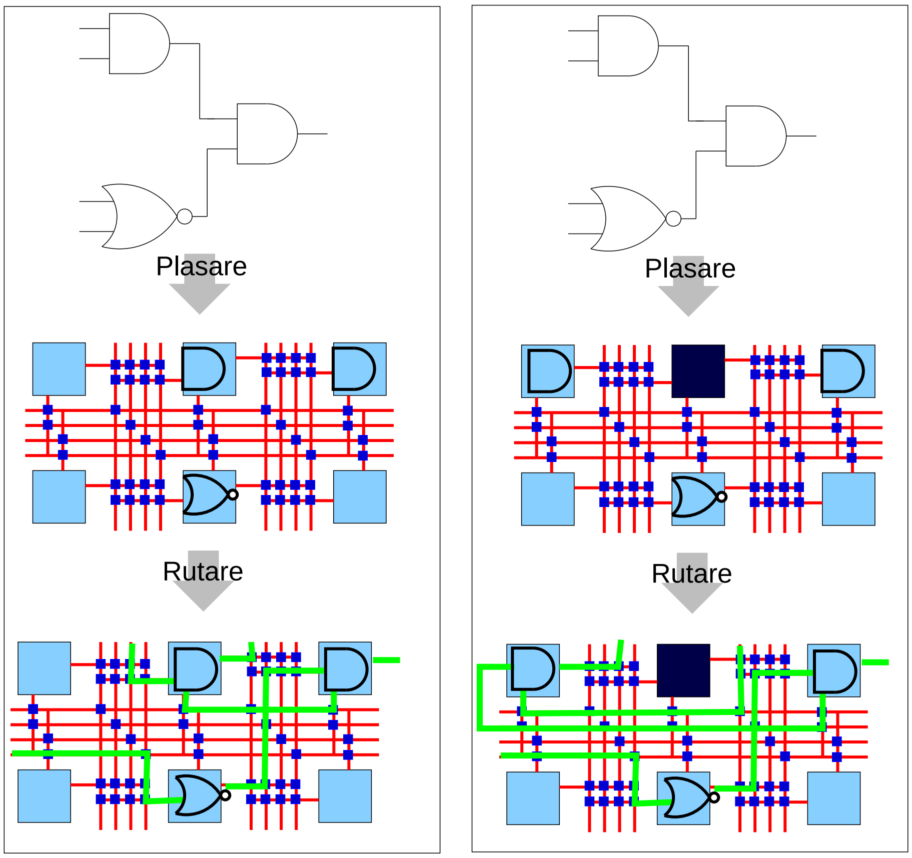

Figura 4: **stînga:** *Procesul de plasare asociază fiecare poartă logică din circuitul de implementat cu o poartă universală. Procesul de rutare conecteaza'porţile universale folosind segmente de sîrmă legate cu comutatoare.* **dreapta:** *Dacă unele din porţile universale sunt defecte, plasarea şi rutarea le pot ocoli, sintetizînd un circuit perfect funcţional.*

configura ca dispozitive care se auto-testează. Fiecare porţiune din fiecare circuit este programată se efectueze calcule simple şi să verifice corectitudinea rezultatelor. Micile programe de test sunt "plimbate" pe suprafaţa circuitului, acoperind toate porţile logice. Proiectarea unor programe de auto-testare este o sarcină mai complicată decît ar putea părea la prima vedere. Programele trebuie să descopere o mulţime de defecte posibile şi trebuie să nu poată fi păcălite de defecţiuni (de exemplu, dacă chiar partea care compară rezultatele cu cele corecte este defectă). Programele de testare aplică în mod repetat calcule care amestecă toţi biţii; astfel, apariţia unei singure erori se va propaga rapid la toţi biţii din rezultat, fiind uşor de depistat.

Proiectul Teramac a avut un succes enorm, apărînd chiar pe prima pagină a unor ziare de mare tiraj. Principala sa contribuţie a constat în a demonstra că defectele din hardware pot fi expuse nivelelor superioare, şi pot fi tratate în întregime în software, fără ca costul plătit în performanţă să fie prohibitiv. Această metodologie este o schimbare completă de paradigmă în arhitectura calculatoarelor, care probabil va avea din ce în ce mai multe aplicaţii în viitor.

7 Algoritmi aleatori

În această secţiune voi discuta o nouă direcţie în gîndirea despre calculatoare, care acceptă lipsa de fiabilitate ca pe un lucru natural, chiar şi la nivelele cele mai ridicate. E vorba de folosirea algoritmilor aleatori, care incorporează un element de şansă, şi care, ca atare, nu se comportă întotdeauna la fel, ba chiar, cîteodată pot da răspunsuri eronate.

Teoria complexităţii ne învaţă că în general există un compromis între timpul de execuţie al unui program şi calitatea rezultatelor pe care le poate oferi. Pentru multe clase importante de probleme nu există algoritmi care să fie simultan rapizi şi care găsesc soluţia optimă.

Singura soluţie în astfel de cazuri este să renunţăm la una dintre calităţi. Dacă nu avem suficient timp, va trebui să ne mulţumim cu soluţii care nu sunt optime, dar poate sunt suficient de aproape de optim, presupunînd că le putem calcula rapid.

Cei mai spectaculoşi algoritmi care au apărut folosesc evenimente aleatoare pentru a calcula răspunsurile rapid, dar pierd din precizie. Trebuie să ne imaginăm aceşti algoritmi ca avînd capacitatea să arunce un zar şi în funcţie de rezultat să facă o acţiune sau alta. Putem distinge două mari clase de astfel de algoritmi:

Algoritmii Monte Carlo de obicei dau răspunsuri corecte, dar ocazional pot da un răspuns greşit. "Ocazional" trebuie citit "cu probabilitate foarte mică". Cu alte cuvinte, dacă "zarurile" nu ies mereu "şase-şase", algoritmul va da un răspuns corect, altfel rezultatul poate fi eronat. Algoritmii Monte-Carlo sunt folosiţi pe larg în simularea fenomenelor fizice pe calculator, aproximînd calcule complicate prin eşantionare.

Algoritmii Las-Vegas dau întotdeauna răspunsuri corecte. Cel mai adesea aceşti algoritmi se şi execută rapid. Cu probabilitate foarte scăzută însă, ei ar putea să se execute pentru un timp foarte îndelungat. Un exemplu tipic de astfel de algoritm este "quicksort", care aranjează un şir de numere în ordine crescătoare. Quicksort alege (într-una din variantele sale) în mod aleator un "pivot", un element din şir, după care mută toate elementele mai mici decît pivotul la stînga şi cele mai mari la dreapta şi sortează în mod recursiv cele două şiruri obţinute. Pentru cele mai multe alegeri ale pivotului timpul de execuţie este scurt; dacă însă algoritmul are ghinion şi alege mereu cîte un pivot aproape de valoarea maximă din şir, timpul de execuţie este mult mai lung.

7.1 Amplificarea probabilităţilor

Foarte surprinzător este faptul că pentru unele probleme nu se cunoaşte nici un algoritm eficient determinist, dar sunt cunoscuţi algoritmi aleatori rapizi. Şi mai surprinzător este faptul că astfel de algoritmi sunt folosiţi pe scară foarte largă, fiind un pilon de bază al comerţului electronic!

Cum de ne bazăm pe algoritmi care ştim că pot eşua?

Totul este o chestiune de valori ale probabilităţilor de eşec. Ştim că orice sistem poate eşua; cu o probabilitate foarte mică se poate petrece un cutremur în următoarea oră, care poate strica un calculator oricît de solid. Dacă algoritmii aleatori au probabilităţi mai mici de atît, pînă la urma ei reprezintă un risc acceptabil.

Din fericire există metode pentru a scădea probabilitatea de eroare a unui algoritm aleator. Aceste metode sunt asemănătoare cu metodele care cresc fiabilitatea sistemelor hardware, prezentate în prima parte a acestui articol, folosind redundanţa.

O metodă foarte similară cu redundanţa temporală este folosită pentru a *amplifica* probabilitatea de succes a algoritmilor Monte-Carlo. Această metodă pur şi simplu execută algoritmul în mod repetat şi alege rezultatul folosind un vot majoritar. Probabilitatea ca majoritatea instanţelor algoritmului să eşueze scade foarte repede cu numărul de repetiţii (exponenţial de repede), şi poate fi adusă practic la o valoare oricît de mică.

7.2 Criptografie

Importanţa criptografiei pentru societatea modernă nu poate fi supraestimată. Timpurile cînd criptografia era apanajul armatelor este de mult apus. De fiecare dată cînd tastaţi o parolă într-un calculator, folosiţi criptografie; de fiecare dată cînd faceţi cumpărături prin Internet, browser-ul dumneavoastră foloseşte criptografie pentru a trimite numărul cărţii de credit.

Cu atît mai interesant este faptul că nici unul dintre sistemele criptografice folosit pe scară largă este demonstrat în mod matematic ca fiind sigur! De fapt întreaga tehnologie a criptografiei se bazează pe nişte probleme matematice al căror răspuns este încă necunoscut.

Una dintre aceste probleme este dacă întrebarea dacă un număr poate fi descompus rapid în factori primi. Deşi aţi învăţat în şcoala primară cum se face asta, dacă veţi încerca să aplicaţi algoritmul acela unor numere mari (de exemplu, cu cîteva sute de cifre), nu aveţi nici o speranţă. Nici măcar calculatoarele nu vă pot ajuta aici: la ora aceasta toţi algoritmii cunoscuţi pentru a descompune un număr în factori primi sunt foarte costisitori (ca timp de execuţie). Unele dintre cele mai folosite criptosisteme sunt bazate pe presupunerea că descompunerea în factori este grea (adică că nu există nici un algoritm rapid).

Putem vedea siguranţa unui sistem criptografic tot ca pe o formă de fiabilitate: dacă probabilitatea ca sistemul să fie înfrînt într-o perioadă lungă de timp este foarte scăzută, atunci sistemul criptografic este fiabil.

Un alt fapt surprinzător despre mulţi dintre algoritmii folosiţi în criptografie este că ei sunt de tip Monte-Carlo, cu alte cuvinte, ei pot funcţiona incorect cu o probabilitate nenulă. De exemplu, mulţi algoritmi încep calculele generînd un număr prim foarte mare. De exemplu, programul `ssh`, (secure shell) care este probabil cel mai folosit program sigur pentru acces la un calculator la distanţă în mod interactiv, trebuie să asigneze calculatorului de pe care porneşte conexiunea un identificator bazat pe un astfel de număr prim generat aleator. Generarea acestui număr prim poate eşua în mai multe feluri:

1. Două calculatoare diferite ar putea genera acelaşi număr din întîmplare, şi atunci unul ar putea să pretindă că este celălalt;

2. Nu se cunoaşte nici un algoritm eficient determinist care generează numere prime. Felul în care numărul prim este generat este următorul: un număr aleator impar este generat, după care se verifică dacă este prim. Procesul se repetă pînă cînd se obţine un număr prim. Acest proces este un algoritm Las Vegas, pentru că probabilitatea de a genera într-una numere compuse este nenulă;

3. Nu se cunoaşte nici un algoritm eficient determinist pentru a verifica dacă un număr este prim! Toţi algoritmii practici sunt de tip Monte-Carlo, pentru că ar putea declara (cu probabilitate mică) un număr compus ca fiind prim. Într-un astfel de caz securitatea oferită de `ssh` nu este garantată.

Cu toate acestea, în practică `ssh` funcţionează foarte bine pentru că:

1. Există aproximativ 10^{150} numere prime cu 512 cifre, adică mai multe decît numărul estimat de atomi din univers. Probabilitatea ca două calculatoare să aleagă acelaşi număr este infimă;

2. Densitatea numerelor prime este relativ ridicată: între primele n numere întregi există aproximativ $n/\log n$ numere prime. Cu probabilitate mare algoritmul Las Vegas de generare a unui număr prim va găsi deci unul destul de repede (după un număr de repetiţii proporţional cu numărul de cifre dorite);

3. Probabilitatea de eroare a algoritmului Monte-Carlo de verificare a primalităţii poate fi micşorată în mod arbitrar folosind tehnica amplificării, descrisă mai sus.

7.3 Algoritmi aleatori în reţele

Dacă credeţi că algoritmii aleatori sunt apanajul teoreticienilor, vă înşelaţi. Ei sunt folosiţi în contexte extrem de aplicate. Voi ilustra cu un exemplu de folosire a unui algoritm Las-Vegas în reţelele de calculatoare. Acest algoritm este folosit cu variaţii în (cel puţin) două contexte:

- Este folosit în protocolul Ethernet ca să arbitreze transmisiunea de date a mai multor calculatoare pe un singur cablu care le conectează;

- Este folosit în protocolul Bluetooth de comunicaţie radio pentru a alege un nod "master" care va arbitra comunicaţia prin mediul electromagnetic a tuturor nodurilor din vecinătatea sa.

În aceste protocoale algoritmul Las Vegas este folosit pentru a distruge simetria intrinsecă în algoritmii determinişti. Dacă doi algoritmi determinişti pleacă din acelaşi punct şi fac mereu aceleaşi operaţii, ei vor fi mereu în aceeaşi stare. Dacă dorim cumva să diferenţiem între cei doi algoritmi, simetria poate fi indezirabilă.

Voi ilustra această problemă cu protocolul Ethernet. Ethernet este cel mai folosit tip de reţea locală, inventat de Bob Metcalfe în 1973. În varianta originală de Ethernet, despre care discut aici, mai multe calculatoare sunt legate la un cablu coaxial. Acest cablu e folosit în mod asemănător cu televiziunea prin cablu pentru a transmite date: un calculator emite datele, iar celelalte ascultă cablul; calculatorul care este destinaţia finală a datelor (identificat în pachetul de date) preia datele din reţea, celelalte calculatoare le ignoră. Spre deosebire de televiziunea prin cablu, în care emiţătorul este unul singur, în cazul Ethernet-ului oricare din calculatoarele conectate poate transmite. Problema este că două calculatoare diferite nu pot transmite simultan, pentru că atunci semnalul este bruiat. Algoritmul aleator este folosit pentru a decide cine transmite atunci cînd mai multe calculatoare vor simultan să folosească cablul, astfel:

1. Cînd un calculator are date de transmis, el începe prin a asculta cablul, ca să vadă dacă este ocupat;

2. Cînd transmisiunea curentă se termină, calculatorul începe să transmită el însuşi datele, în speranţa că nimeni altcineva nu mai doreşte să facă acelaşi lucru;

3. Dacă două calculatoare transmit simultan ele vor observa semnalul bruiat şi atunci intră în faza aleatoare, care selectează cine transmite primul:
   (a) Ambele calculatoare dau cu zarul, alegînd un număr între 1 şi 2. Fiecare numără pînă la valoarea aleasă şi apoi încearcă din nou să transmită;
   (b) Dacă ambele au ales acelaşi număr se va petrece o nouă "coliziune"; atunci ambele aleg un număr între 1 şi 4 şi repetă procedura;
   (c) La fiecare nouă coliziune intervalul în care se aleg numerele aleatoare îşi dublează lungimea, pînă cînd un calculator alege un număr mai mic şi ca atare emite primul.

Creşterea exponenţială a intervalului în care se generează numerele aleatoare se numeşte "exponential back-off". Această creştere rapidă asigură faptul că probabilitatea de coliziune descreşte exponenţial de rapid cu timpul.

Algoritmul descris mai sus se mai numeşte şi CSMA/CD, de la Carrier-Sense Multiple Access, Collision Detect, adică "ascultarea mediului de transmisie pentru accese multiple cu detectarea coliziunilor"; este un algoritm Las Vegas relativ simplu. Reţelele moderne Ethernet sunt ceva mai complicate, dar continuă să se bazeze pe acest algoritm. Eficacitatea sa este demonstrată de succesul acestui tip de reţea.

8 Spre o nouă arhitectură a calculatoarelor

În acest text, întins pe două numere de revistă, am discutat mai multe feluri în care consideraţii privind fiabilitatea influenţează construcţia calculatoarelor. Un mesaj important al acestui text este că, deşi fiabilitatea ridicată este dezirabilă, un arhitect întotdeauna trebuie să ia în consideraţie şi costul plătit pentru a o obţine.

Calculatoarele moderne sunt construite dintr-o serie de nivele abstracte, care oferă funcţionalităţi din ce în ce mai puternice. Fiecare nivel are o fiabilitate diferită şi foloseşte tehnici diferite pentru a oferi nivelelor superioare imaginea unei fiabilităţi sporite. În general hardware-ul oferă lumii software aparenţa perfecţiunii în această privinţă, adică o fiabilitate excepţional de ridicată.

Tendinţele tehnologiei indică însă că arhitectura calculatoarelor viitorului va fi supusă unor schimbări radicale, unul dintre motive fiind chiar schimbarea majoră a fiabilităţii unora dintre nivele. De exemplu, miniaturizarea continuă a componentelor electronice va fi însoţită de o degradare a fiabilităţii însoţită de apariţia tot mai frecventă a defecţiunilor permanente şi tranziente. Costul plătit pentru a masca aceste defecte prin tehnici tradiţionale creşte extrem de rapid: costul extrem de ridicat al unei fabrici de semiconductoare de ultima generaţie, de ordinul a cîteva miliarde de dolari, este doar primul simptom al acestui fenomen.

Îmi permit să speculez că o schimbare de perspectivă în ceea ce priveşte abordarea fiabilităţii poate avea consecinţe enorme. De exemplu, dacă renunţăm să mai construim hardware "perfect", putem reduce în mod absolut dramatic costul de fabricaţie. Proiectul Teramac a arătat o metodă prin care hardware cu defecte poate fi folosit cu succes.

Mai multe grupuri de cercetare lucrează în mod activ pentru a defini arhitecturile viitorului. Am prezentat de curînd în Net Report (mai-iunie 2001) eforturile grupului din care fac eu parte. Propunerea noastră poate fi rezumată astfel:

- Iluzia unui hardware perfect trebuie eliminată; imperfecţiunile din nivelul hardware trebuie să fie expuse nivelului software şi rezolvate de acesta;
- Pentru a face acest lucru, baza sistemelor de calcul trebuie să fie hardware-ul reconfigurabil, care este suficient de flexibil pentru a fi reprogramat după nevoi;
- Microprocesorul trebuie să fie redus la un rol secundar şi înlocuit cu hardware generat specific pentru fiecare aplicaţie cu ajutorul compilatoarelor;
- Numărul de nivele abstracte trebuie să fie micşorat în mod dramatic, pentru a reduce costul suplimentar plătit (overhead), care creşte exponenţial;
- Trebuie folosite în mod constant tehnici care descoperă defecte şi folosesc rezerve pentru a ocoli defectele de fabricaţie;
- Calculele trebuie să fie efectuate folosind date codificate folosind coduri robuste, pentru a preveni efectele erorilor tranziente;
- În fine trebuie folosite pe scară largă metodele de verificare formală pentru a ne asigura că sistemele de calcul pe care le folosim sunt corect construite.

Ştiinţa calculatoarelor este relativ tînără; cu siguranţă că viitorul ne rezervă o mulţime de surprize în ceea ce priveşte tehnologiile, arhitectura şi algoritmii cei mai eficienţi.

9 Alte surse de informaţie

- Două articole din IBM Journal of Research despre microprocesoarele fiabile G4 şi G5 de la IBM:
  - `http://www.research.ibm.com/journal/rd/435/spainhower.html`,
  - `http://www.research.ibm.com/journal/rd/414/webb.html`
- Informaţii despre procesorul DIVA:
  - Două prezentări făcute de inventatorului acestui sistem:
    - `http://huron.cs.ucdavis.edu/Micro32/presentations/p_austin.pdf`
    - `http://www.cs.utexas.edu/users/cart/arch/slides/austin.pdf`
  - Un articol care descrie schema DIVA: `http://jilp.org/vol2/v2paper7.ps`
- Articolul meu din PC Report din iulie 1999 despre redenumirea regiştrilor, tehnică folosită de procesorul superscalar tolerant la erori:
  `http://www.cs.cmu.edu/ mihaib/articole/articole.html#redenumire`.
- Un articol care descrie microprocesorul superscalar tolerant la erori:
  `http://www.ece.cmu.edu/ babak/papers/micro01-dual.ps`
- Un articol din PC Guide despre structura hard-discurilor:
  `http://www.pcguide.com/ref/hdd/index.htm`

- O pagină de web dedicată Turbo-codurilor:
  http://www.ee.virginia.edu/CSL/turbo_codes/home.html.
- Un articol despre sisteme RAID din Byte.com:
  http://www.usbyte.com/common/raid_systems.htm.
- Articole ale mele anterioare din PC Report:
  - Despre funcţionarea Internetului: mai 1998
    http://www.cs.cmu.edu/ mihaib/articole/articole.html#internet,
  - Despre hardware reconfigurabil: iulie 1998
    http://www.cs.cmu.edu/ mihaib/articole/articole.html#reconf,
  - Despre algoritmi aleatori: august 1997
    http://www.cs.cmu.edu/ mihaib/articole/articole.html#algoritmi,
  - Despre nanotehnologie şi arhitectura calculatoarelor: mai-iunie 2001,
    http://www.cs.cmu.edu/ mihaib/articole/articole.html#nano,
    http://www.cs.cmu.edu/ mihaib/articole/articole.html#spatial,
  - Despre verificarea formală: noiembrie 2001,
    http://www.cs.cmu.edu/ mihaib/articole/articole.html#metacompilare,