Gheorghe M.Panaitescu

SISTEME TOLERANTE LA DEFECTE

Note de curs

Universitatea "Petrol-Gaze" Ploiesti
Catedra Automatică si calculatoare
2009

CUVÂNT ÎNAINTE

Ce reprezintă această lucrarea?

Lucrarea aceasta pe care o consultati este suportul cursului *Sisteme tolerante la defecte*, un ghid al expunerilor săptămânale pe care autorul le-a tinut/le tine pentru anul IV de la specializarea Calculatoare a facultătii Inginerie mecanică si electrică, de cele mai multe ori în fata unei asistențe modeste numeric. Este totodată un text de ajutor în pregătirea pentru verificarea pe care studentii o susțin la sfârsitul primului semestru al ultimului an de studii.
Desigur, nu este exclusă o lectură interesantă si interesată a unor terțe persoane, care nu sunt nici studenți si nici nu predau disciplina atât de atractivă a toleranței la defecte.
Există, desigur, o diferență între versiunea vorbită si cea scrisă a acestui curs: scrisul reclamă concizie, vorbitul permite detalierea unor aspecte care merită sau trebuie detaliate. Comunicarea verbală poate beneficia de întreruperi utile din partea celor care ascultă, ceea ce autorul si-a dorit foarte mult, dar, din păcate, s-a întâmplat foarte rar.

Despre continutul acestei lucrări, despre surse

Nu veti găsi în paginile care urmează foarte multe lucruri originale. Pe tema originalității există în zilele noastre o (să-i spunem) dezbatere asupra dozei de noutate din cursurile scrise de universitarii români. Sunt publicate adesea opinii ale unor oameni care n-au scris vreodată un rând dar dau cu generozitate indicații altora, mai ales când vor să arate cât de incorect este profesorul X în utilizarea surselor bibliografice. Vor fi urme de adevăr în cele scrise ocazional în acele articole de campanie, altfel foarte combative. Nu sunt cel chemat a arbitra în această chestiune. Nu mă pot reține însă de la a face câteva precizări despre lucrarea de fată.
Subiectele tratate aici urmează linia care se regăseste din plin în cursurile prezentate studentilor unor universităti din Statele Unite ale Americii. Ca să scutesc pe cititor, indiferent cine ar fi el, de oricc efort detectivistic, fac trimiteri la cursurile similare tinute la University of Massachusetts (cursul ECE 655), la University of Wisconsin (cursul ECE 753), la University of Idaho (cursul CS449_549) sau la University of Illinois (cursul ECE 542); un “search” pe Internet pe codul cursurilor si pe numele universitătilor poate duce cititorul direct la aceste surse *principale* utilizate în versiunea curentă a acestor Note de curs. Am scris “principale” pentru că în afara slide-urilor sau paginilor postate la locurile indicate, alte sute de pagini au fost parcurse pentru lămurirea unor

detalii, deoarece, este stiut, este absurd să predai o disciplină pe care nu o înțelegi…
Am apelat la surse americane pentru motivul lesne de înțeles: tehnologia calculatoarelor si preocuparea pentru siguranța calculului nu sunt altundeva mai înalte decât în America.
Se poate adăuga ceva original la ceea ce industria si comunitatea academică americană a creat în domeniu? Poate de aici înainte, după lectura acestor Note de curs…
Domeniul este pasionant, punctele de vedere sunt multiple si variate, bibliografia este practic inepuizabilă. De aceea, această versiune a lucrării (a patra, a concea?) este radical diferită de acelea din anii trecuți si nu mă îndoiesc că este diferită si de cea (posibilă) de anul viitor…
Un *feedback* cu observatii asupra acestui curs, la adresa de e-mail a autorului ar fi aproape sigur foarte util pentru realizarea versiunii viitoare.

Ce consecinte ar putea avea lectura acestei lucrări?

Cititorul va găsi aici notiuni fundamentale despre cele patru tipuri de redundanțe care se folosesc separat sau în combinații pentru realizarea toleranței la defecte: redundanțele hardware, redundanțele software, redundanțele informaționale si redundanțele temporale.
Nu este exagerată credința că după audierea cursului de *Sisteme tolerante la defecte* si după lectura acestor Note, studentul absolvent va fi mai aproape de mecanismele utilizate pentru a prelungi la limita posibilului funcționarea corectă a unui calculator, a unui program de calcul, pentru a mări siguranța serviciului unei baze de date, a comunicării în rețelele de calculatoare.

Gheorghe M.Panaitescu
gmpanaitescu@upg-ploiesti.ro

Ploiesti, 5 octombrie 2009

C U P R I N S

INTRODUCERE

Definirea obiectului disciplinei

Ideal, un sistem tolerant la defecte este un sistem capabil a executa corect sarcinile lui de calcul sau de altă natură indiferent dacă apar defecte *hardware* sau *software*.
În practică, niciodată nu poate fi garantată în orice împrejurare executarea acelor sarcini fără întrerupere.
Discutiile de genul celor care urmează se limitează uzual la tipuri de defectări si de erori care sunt mai probabil să apară.

Aplicatii în care toleranta la defecte este necesară

Există în lumea reală aplicatii considerate pe drept cuvânt critice sub aspect vital. Este vorba, de plidă, de aeronave, de reactoarele nucleare, de unele instalatii chimice si de echipamentele medicale. O functionare proastă a unui calculator ataşat unor asemenea aplicatii poate duce la catastrofe. Probabilitatea de defectare a sistemelor de calcul din această categorie trebuie să fie extrem de scăzută, sub unu la un miliard pe ora de operare.
Ambianta aspră este alt element care impune un grad de a tolera defectele. Un sistem de calcul care operează în conditii vitrege, în care sunt prezente perturbatiile electromagnetice, bombardamentul cu particule încărcate sau neutre electric, sau altele asemenea trebuie să fie tolerant la defecte. Un număr mare de disfunctii datorate factorilor de mediu, altminteri destul de probabile, poate face ca sistemul să nu producă rezultate utile decât dacă are încorporată încă din faza de conceptie o doză de tolerantă la defecte.
Din alt unghi, unele sisteme foarte complexe constau în milioane de dispozitive elementare. Fiecare dispozitiv fizic component are o anumită probabilitate de a esua. Un număr foarte mare de dispozitive implică o probabilitate de mers neconform încă mai mare. Sistemul poate manifesta defectări cu o asa frecventă încât el poate deveni inutilizabil.

Măsuri ale tolerantei la defecte

Este foarte important a avea definite *măsuri* care să cuantifice nivelul tolerantei la defecte. Ca în orice domeniu, o măsură este o abstractie matematică care exprimă numai un aspect al naturii obiectelor.
În domeniul sigurantei în functionare a sistemelor, sunt desigur unele măsuri considerate deja traditionale. Se consideră mai întâi că sistemul poate fi în una

din două stări mutual eclusive: *functional* sau *disfunct*. De exemplu, un bec este fie bun, fie ars; un fir este fie continuu, fie întrerupt. În legătură strictă cu aceste două stări se utilizează două măsuri deja traditionale: *fiabilitatea* si *disponibilitatea*.

Fiabilitatea, notată cu $R(t)$ este probabilitatea ca sistemul să fie functional pe durata intervalului $[0, t]$, el fiind functional la momentul $t = 0$.

Disponibilitatea, notată cu $A(t)$ este cota parte din timp în care sistemul este functional în intervalul $[0, t]$. Ca măsură complementară, se defineste o disponibilitate punctuală $A_p(t)$: probabilitatea ca sistemul să fie functional la momentul t. O măsură înrudită este si *MTTF – Mean Time To Failure* – care este timpul mediu cât sistemul este functional înainte ca el să se defecteze pentru ca apoi să fie reparat sau înlocuit.

Sunt însă necesare si alte măsuri în completarea acestora. Presupunerea că sistemul poate fi doar în stările "functional" sau "disfunct" este foarte restrictivă. De exemplu, un procesor cu una din cele câteva sute de milioane de porti blocată în valoarea logică 0 si restul functionale poate fi conjunctural si functional, si disfunct. Poarta aceasta defectă poate afecta iesirea procesorului o dată la 25.000 de ore de utilizare. Procesorul nu este lipsit de orice defectiune dar nu poate fi calificat ca *disfunct*. Pentru a caracteriza astfel de stări, sunt necesare mijloace de apreciere cantitativă suplimentare pe lângă deja traditionalele *fiabilitate* si *disponibilitate*. În continuare sunt discutate câteva măsuri capabile a cuprinde si alte nuante.

Fiabilitatea prin capacitate este tot o probabilitate, probabilitatea ca o anumită capacitate a sistemului (măsurabilă, de pildă un randament) la timpul t să depăsească un prag dat. O altă extindere a gamei de măsuri ia în considerare totul din perspectiva aplicatiei. Aceasta duce la definirea măsurii cunoscută ca *performabilitate*.

Relativ la *capacitatea* unui sistem, fie aceasta capacitatea de calcul. Fie, de exemplu, un sistem cu N procesoare, sistem care se degradează *cu gratie*, adică nu brusc ci gradual. Sistemul se recuperează din starea de disfunctie a unora dintre procesoare si este utilizabil atât timp cât cel putin un procesor este functional. Fie P_i probabilitatea ca i procesoare din cele N să fie functionale. Fiabiltatea sistemului este o sumă a acestor probabilităti după indicele i, excluzând, desigur, valoarea $i = 0$.

$$R(t) = \sum_{i=1}^{N} P_i$$

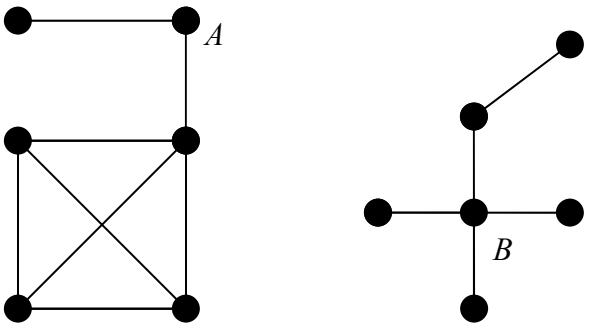
Fie c capacitatea de calcul a unui procesor (de pildă numărul de task-uri de dimensiune fixă pe care le poate executa). Capacitatea de calcul a i procesoare este atunci $C_i = i \bullet c$. Capacitatea de calcul a sistemului este în consecintă $\sum_{i=1}^{N} C_i P_i$.

Relativ la *performabilitate*, în legătură cu aplicatia la care se referă se definesc niste *niveluri de îndeplinire*, $L_1, L_2, \ldots, L_n$. Fiecare dintre acestea reprezintă un nivel de calitate a serviciului efectuat prin acea aplicatie. De exemplu, L_i –

sistemul *i* nu "cade" pe durata *T* a misiunii lui. *Performabilitatea* este un vector [$P(L_1)$ $P(L_2)$ … $P(L_n)$] în care $P(L_i)$ notează probabilitatea ca sistemul să functioneze suficient de bine pentru a permite aplicatiei să atingă nivelul de îndeplinire L_i.

Măsuri ale conectivitătii unei retele

O măsură a conectivitătii trebuie să se focalizeze pe reteaua care leagă procesoarele. *Conectivitatea* clasică se referă la nodurile si liniile unei retele de procesoare si reprezintă numărul minim de noduri, respectiv de linii, care trebuie să cadă înainte ca reteaua să devină neconexă. Această măsură indică vulnerabilitatea retelei la a deveni neconexă. O retea care poate deveni neconexă prin disfunctia unui singur nod (pozitionat critic) este potential mult mai vulnerabilă decât o alta care poate deveni neconexă numai dacă mai multe noduri cad.



Reteaua N_1 Reteaua N_2

Exemple de conectivitate

Măsura rezilientei unei retele

Conectivitatea clasică distinge între două stări ale retelei: conexă si neconexă. Conectivitatea clasică nu spune nimic despre modul cum reteaua se degradează pe măsură ce nodurile cad înainte ca reteaua să devină neconexă. Sunt două măsuri posibile ale *rezilientei*:

- Distanta medie pe perechi de noduri
- Diametrul retelei (maximul distantei între două noduri) cu probabilitătile disfunctiilor pe noduri si/sau pe linii date

Alte măsuri pentru retele

Ce se întâmplă când reteaua încetează a mai fi conexă? O retea care se divide într-o componentă (conexă) majoră si mai multe bucăti (conexe) mărunte poate încă să functioneze. Dar care este functionalitatea unei retele care se divide într-

un număr mare de subretele mici? Sansele de functionare sunt mai reduse. De aceea, o altă măsură a rezilientei la defectare a unei retele este probabilitatea distribuirii componentei majore (celei mai mari) la căderea unui nod, unei linii.

Redundanțe

Redundanțele sunt centrale în realizarea tolerantei la defecte. Redundanta se poate defini ca rezultatul încorporării în sistem, prin proiect, a unor module suplimentare, în asa mod încât functia sistemului să nu fie periclitată de aparitia unei disfunctii. În continuare sunt studiate patru tipuri de redundante.

A. *Redundante hardware*. Acest gen de redundante se crează atunci când se adaugă elemente hardware suplimentare pentru a depăsi efectele disfunctiei unei componente. Redundanta hardware poate fi *statică*, caz în care se realizează mascarea imediată a unei disfunctii. Un exemplu: se utilizează trei procesoare în loc de unul si fiecare execută aceeasi functie. Iesirea majoritară a acestor procesoare elimină iesirea gresită a unuia singur. În cazul redundantei hardware *dinamice*, componentele suplimentare sunt activate numai la aparitia disfunctiei unei componente curent activă, în functiune. Redundanta hardware *hibridă* este o combinatie de metode specifice redundantelor statice si redundantelor dinamice.
B. *Redundante software*. Redundantele software se asigură prin echipe de programare multiple. Se scriu versiuni diferite de software pentru aceeasi functie, pentru aceeasi aplicatie. Redundanta software se bazează pe speranta că o astfel de diversitate dă siguranta că nu toate variantele de program vor esua pe acelasi set de date de intrare.
C. *Redundante informationale*. Redundantele informationale se realizează prin adăugarea de biti la cei originari. Erorile în biti pot fi detectate si chiar corectate. S-au dezvoltat si se utilizează codurile detectoare si corectoare de erori. Redundanta informațională atrage după sine uneori redundante hardware menite a prelucra bitii suplimentari.
D. *Redundante temporale*. Redundantele temporale înseamnă timp suplimentar pentru ca executiile esuate să poată fi repetate. Cele mai multe dintre esecuri sunt tranzitorii si efectul lor se atenuează după un timp. Dacă există timp suficient la dispozitie, modulul disfunct tranzitoriu poate recupera si poate reface calculele afectate.

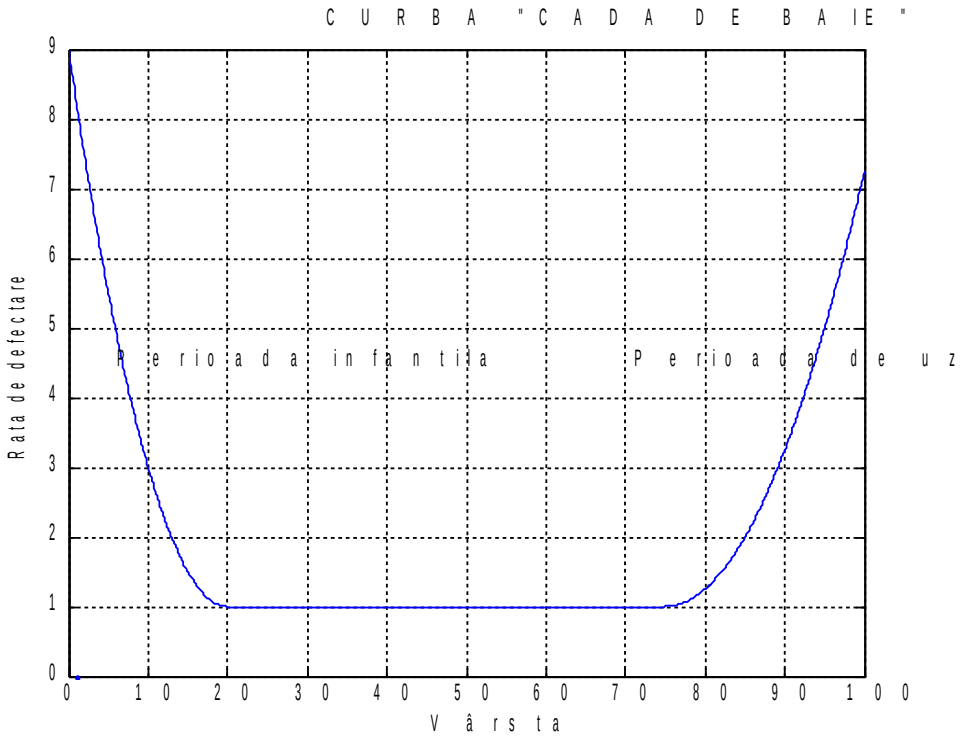
Clasificarea defectelor hardware

Sunt trei tipuri de defecte hardware.
Unele sunt *tranzitorii* si dispar după relativ putin timp. Un exemplu ar putea fi o celulă de memorie al cărei continut este schimbat datorită unei interferente electromagnetice. Rescrierea ei cu continutul corect face ca eroarea să dispară.
Alte defecte sunt *permanente* si nu dispar niciodată, iar componenta trebuie reparată sau înlocuită.

Există si defecte *intermitente*, caz în care defectul componentei pendulează între o stare activă si o stare benignă. Se poate exemplifica cu cazul unei conexiuni slabe/slăbite.

Rata defectărilor

Rata la care o componentă manifestă disfunctii depinde de vârstă, de parametrii (micro)climatici locali, de eventualele socuri fizice cum ar fi cele date de tensiunea de alimentare, dar si de tehnologia de realizare concretă a acelei componente. Dependenta de vârstă a ratei de defectare este ilustrată uzual de curba cadă-de-baie (v.figura).



La tinerete rata defectărilor este ridicată: sunt sanse bune ca unele module cu defecte de fabricatie să treacă de controlul de calitate si să ajungă pe piată. Pe măsura trecerii timpului aceste unităti/module sunt eliminate si pentru mare parte din viată sistemul manifestă o rată de defectare practic constantă. Pe măsură ce sistemul tinde să devină foarte vechi, efectele îmbătrânirii preiau primplanul si rata defectărilor devine din nou variabilă si crescătoare.

O formulă empirică pentru λ, rata de defectare

Rata de defectare se poate estima cu relatia

$$\lambda = \pi_L \, \pi_Q \, (C_1 \, \pi_T \, \pi_V + C_2 \, \pi_E)$$

în care:
π_L – un factor de învătare care arată cât de matură este tehnologia
π_Q – un factor de calitate a fabricatiei între 0,25 si 20
π_T – un factor de temperatură (de la 0,1 la 1000), proportional cu $exp(-E_a/kT)$ cu E_a energia de activare [eV] asociată cu tehnologia, k constanta lui Boltzmann si T temperatura în K.
π_V – un factor de stres prin tensiune pentru dispozitivele CMOS (de la 1 la 10, depinzând de tensiunea de alimetare si de temperatură); nu se aplică altor tehnologii (se pune egal cu 1)
π_E – un factor de soc ambiant: de la cca. 0,4 (mediu de aer conditionat) la 13 (ambient dur)
C_1, C_2 – factori de complexitate, dependenti de numărul de porti pe chip si numărul de pini în pachet.
Detalii se găsesc în manualele de tip *Handbook*.

Impactul mediului ambiant

Un dispozitiv care lucrează în spatiul interastral, spatiu care este plin de particule încărcate energetic si care este capabil a supune dispozitivele la variatii drastice de temperatură este de asteptat a claca mai des. Similar, calculatoarele din automobile (temperaturi ridicate si vibratii) si din aplicatiile industriale sunt susceptibile a se defecta mai frecvent.

Defect si eroare

Un *defect* în sensul tolerantei la defecte poate fi o disfunctie (mai curând locală) de hardware sau o greseală de software sau de programare. O *eroare* este o manifestare a unui defect (fault). Ca exemplu se consideră un circuit de aditie în binar cu una din liniile de iesire "agățată" la 1. Acesta-i un defect dar încă nu este o eroare. Defectul produce o eroare dacă circuitul este folosit si dacă rezultatul pe linia defectă ar trebui să fie 0 si nu 1.

Propagarea defectelor si erorilor

Atât defectele cât si erorile se pot difuza în sistem. Dacă de pildă un chip scurtcircuitează alimentarea cu energie la masă, asta poate determina ca chip-urile apropiate să clacheze si ele. Erorile se difuzează pentru simplul motiv că o iesire (eronată) a unui element procesor este utilizată frecvent ca intrare a altor elemente procesoare. Exemplul circuitului de adunare este ilustrativ: rezultatul eronat al circuitului poate fi utilizat în alte calcule, desigur cu propagarea erorii.

Zone de continentă

Pentru a limita situatii de genul mentionat putin mai sus, proiectantii încorporează în sistem niste asa-numite *zone de continență*. Cum? Prin bariere care reduc sansa ca un defect sau o eroare dintr-o zonă să se propage într-o altă zonă. O zonă de continentă poate fi creată uneori prin asigurarea de alimentări independente pentru fiecare zonă. Proiectantul unui sistem încearcă prin aceasta să izoleze electric o zonă de celelalte. O zonă de conținere sau de continență a erorilor poate fi creată prin utilizarea de module redundante si prin *voting* pe iesirile lor.

Timpul până la cădere – modelul analitic

Se consideră modelul următor:
Sistemul are N componente identice, toate operationale la momentul $t = 0$. Fiecare componentă rămâne operațională până când apare un defect. Toate defectele sunt permanente si apar în cele N componente independent unul de celălalt. Pentru o componentă oarecare, fie T durata de viată a acelei componente, adică timpul până la aparitia defectului considerat fatal. T este o variabilă aleatoare cu densitatea de probabilitate $f(t)$ si functia de repartitie (cumulativă) $F(t)$.
Într-o interpretare probabilistică, $F(t)$ este probabilitatea ca o componentă să esueze înainte de timpul t, $F(t) = \Pr(T \le t)$. Functia $f(t)$ este un gen de rată momentană a căderii: $f(t)\Delta t = \Pr(t \le T \le t + \Delta t)$ cu Δt oricât de mic.
Ca orice functie densitate de probabilitate

$$\int_0^\infty f(t)dt = 1 \text{ si } f(t) \ge 0 \text{ pentru orice } t \ge 0$$

Cele două functii definite aici sunt în relatiile

$$f(t) = dF(t)/dt,\ F(t) = \int_0^t f(\tau)d\tau$$

Fiabilitatea si rata (hazardul) defectării

Fiabilitatea unei singure componente este

$$R(t) = \Pr(T > t) = 1 - F(t)$$

Probabilitatea (infinitesimală) de cădere a unei componente la momentul t, $p(t)$ este probabilitatea conditionată ca acea componentă să cadă la momentul t fiind functională în orice moment înainte de t.

$$p(t) = \Pr(t \le T \le t + dt/T \ge t) =$$
$$= \Pr(t \le T \le t + dt)/\Pr(T \ge t) = f(t)dt/[1 - F(t)]$$

Rata defectărilor (sau rata hazardului) pentru o componentă la timpul t, este definită ca raportul $p(t)/dt$ si se notează cu

$$h(t) = f(t)\ /[1 - F(t)]$$

si deoarece $dR(t)/dt = -f(t)$

$$h(t) = -1/R(t)\ dR(t)/dt$$

Rata defectărilor constantă

Dacă rata defectărilor este constantă în timp, $h(t) = \lambda$, atunci relatiile

$$dR(t)/dt = -\lambda R(t);\ R(0) = 1$$

sunt o ecuatie diferentială si conditia ei initială. Solutia ecuatiei este

$$R(t) = e^{-\lambda t}$$

Ceea ce conduce la

$$f(t) = \lambda e^{-\lambda t} \text{ si } F(t) = 1 - e^{-\lambda t}$$

O rată a defectărilor constantă se obtine dacă si numai dacă T, durata de viată a componentei are o distributie exponentială.

Timpul mediu până la cădere

MTTF (Mean Time To Failure) este durata medie de viată, media variabilei aleatoare T

$$MTTF = E(T) = \int_0^\infty t f(t) dt$$

cu $E(\cdot)$ notatia pentru speranta matematică (espérance, expected value) a variabilei aleatoare trecută ca argument. Deoarece $dR(t)/dt = -f(t)$

$$MTTF = -\int_0^\infty t \frac{dR(t)}{dt} dt = [-t\ R(t)]|_0^\infty + \int_0^\infty R(t) dt$$

Dar $-t\ R(t) = 0$ pentru $t = 0$. La fel si pentru $t = \infty$ deoarece $R(\infty) = 0$. Asadar

$$MTTF = \int_0^\infty R(t) dt$$

Dacă rata defectărilor este constantă, λ, atunci

$$R(t) = e^{-\lambda t} \text{ si } MTTF = \int_0^\infty e^{-\lambda t} dt = 1/\lambda$$

Distributia Weibull

În multe calcule de fiabilitate, rata defectărilor λ se presupune constantă sau, echivalent, se admite o repartitie exponentială pentru durata de viată T. Sunt însă cazuri în care această presupunere este inadecvată. Ca exemple se pot lua zonele infantilă si de senectute din curba "cadă-de-baie". În asemenea cazuri legea de distributie *Weibull* pentru T poate fi mai potrivită.

Distributia Weibull are doi parametri, λ si β. Functia densitate de probabilitate pentru durata de viată a unei componente este

$$f(t) = \lambda \beta t^{\beta - 1} e^{-\lambda t^\beta}$$

Rata defectărilor în cazul distributiei Weibull este

$$h(t) = \lambda \beta t^{\beta - 1}$$

Rata defectărilor $h(t)$ este descrescătoare, constantă sau crescătoare după cum constanta $\beta < 1$, $\beta = 1$ sau $\beta > 1$, ceea ce o face adecvată si pentru copilăria componentei, si pentru maturitatea ei, dar si, respectiv, pentru vârste avansate.

MTTF pentru distributia Weibull

Functia de fiabilitate pentru distributia Weibull este

$$R(t) = e^{-\lambda t^{\beta}}$$

Timpul mediu până la cădere pentru aceeasi distributie este

$$MTTF = \Gamma(1/\beta)/(\beta \lambda^{1/\beta})$$

cu $\Gamma(x)$ functia Gamma, functia eulerianã de prima specie.
Cazul particular $\beta = 1$ face din distributia Weibull o distributie exponentială, cu rata defectărilor constantă.

TOLERANTA LA DEFECTE HARDWARE

Structuri canonice

Din componente se pot construi diverse structuri mai mult sau mai putin complexe. Structurile complexe pot fi construite din câteva structuri de bază. Se admite tacit sau explicit independenta statistică a căderilor pe care le pot manifesta componentele individuale. Structurile de bază sunt trei: sistemele *serie*, sistemele *paralel* si sistemele *M-din-N*.

Sistemele serie

Un sistem serie constă într-un set de componente conectate astfel încât căderea unei componente provoacă căderea întregului sistem (v.figura[1]).

Fiabilitatea unui sistem serie, $R_s(t)$ este produsul fiabilităților celor N module.
Într-adevăr, sistemul este functional dacă si numai dacă toate modulele componente sunt functionale. Dacă $R_i(t)$ este fiabilitatea componentei i a sistemului atunci, folosind formula pentru intersectia de evenimente mutual independente rezultă

$$R_s(t) = \prod_{i=1}^{N} R_i(t)$$

În particular, dacă fiecare modul i are o rată de defectare constantă, λ_i, atunci

$$R_i(t) = e^{-\lambda_i t}$$

si

$$R_s(t) = e^{-\lambda_s t} = e^{-\sum \lambda_i t}$$

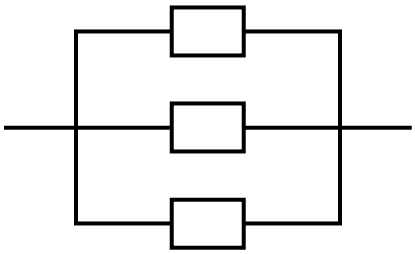
cu $\lambda_s = \Sigma\lambda_i$ rata de defectare constantă a sistemului serie în ansamblu.
Conform relatiei generale de calcul, timpul mediu până la cădere pentru un sistem serie este

$$MTTF_s = 1/\lambda_s = 1/\Sigma\lambda_i$$

[1] Este convenabilă reprezentarea sistemelor prin asa-numitele *diagrame de semnal*. Diagramele de semnal au o singură intrare, plasată uzual în stânga si o singură iesire, plasată uzual în dreapta diagramei. Dreptunghiurile reprezintă module functionale. Starea sistemului este "functional" dacă există un drum de la intrare la iesire. Absenta oricărui drum intrare-iesire corespunde stării "disfunct".

Sisteme paralel

Un sistem paralel este alcătuit dint-un set de module conectate astfel încât pentru ca sistemul să cadă trebuie să cadă toate componentele (v.figura)



Fiabilitatea unui sistem paralel, $R_p(t)$ rezultă din relatia

$$1 - R_p(t) = \prod_{i=1}^{N} (1 - R_i(t))$$

care exprimă probabilitatea esecului concomitent al tuturor componentelor si deci al sistemului. Din nou se utilizează relatia de calcul al probabilitătii unei intersectii de evenimente mutual independente, fiecare cu probabilitatea particulară egală cu $(1 - R_i(t))$. Din relatia de mai sus rezultă fiabilitatea pentru structura paralel

$$R_p(t) = 1 - \prod_{i=1}^{N} (1 - R_i(t))$$

Din nou, în particular, dacă fiecare modul i are o rată de defectare constantă λ_i, atunci

$$R_i(t) = e^{-\lambda_i t}$$

si

$$R_p(t) = 1 - \prod_{i=1}^{N} (1 - e^{-\lambda_i t})$$

Ca exemplu, pentru un sistem paralel cu două module

$$R_p(t) = e^{-\lambda_1 t} + e^{-\lambda_2 t} - e^{-(\lambda_1 + \lambda_2)t}$$

si timpul mediu până la cădere pentru un astfel de sistem este

$$MTTF_{P(2)} = \frac{1}{\lambda_1} + \frac{1}{\lambda_2} - \frac{1}{\lambda_1 + \lambda_2}$$

Timpul mediu până la cădere pentru un sistem paralel alcătuit din N module identice cu rata de defectare λ oarecare este

$$MTTF_{P(N)} = \sum_{i=1}^{N} \frac{1}{i\lambda}$$

relatie verificată si de formula anterioară, dacă $\lambda_1 = \lambda_2 = \lambda$.

Sisteme *M*-din-*N*

Un sistem *M*-din-*N* este un sistem alcătuit din *N* module componente, identice în ceea ce priveste functia de îndeplinit. Un astfel de sistem rămâne functional atât timp cât cel putin *M* componente sunt functionale. Identitatea functională a celor *N* module se extinde de obicei si asupra fiabilitătii lor.
Cel mai bun exemplu si totodată un sistem foarte utilizat este *triplexul* (*TMR* – Triple Modular Redundant). Este vorba aici de trei componente identice care primesc aceasi intrare, suplimentat cu un dispozitiv de votare, *voter*, asupra iesirilor lor. Acesta este un sistem (M = 2)-din-(N = 3): atât timp cât cele mai multe (majoritatea) dintre componente produc rezultate corecte, sistemul este functional.

Fiabilitatea unui sistem *M*-din-*N*

Cele N componente sunt identice. Dacă $R(t)$ este fiabilitatea individuală a unui subsistem atunci fiabilitatea sistemului este probabilitatea ca $N - M$ sau mai putine subsisteme componente să fi căzut până la momentul t

$$R_{\text{M-din-N}}(t) = \sum_{i=0}^{N-M} C_N^i [1 - R(t)]^i [R(t)]^{N-i}$$

cu

$$C_N^i = \frac{N!}{i!(N-i)!}$$

Căderi corelate în sistemele *M*-din-*N*

Independenta statistică a caderilor de componente este cheia fiabilitătii ridicate a sistemelor *M*-din-*N*. Căderile corelate pot diminua considerabil fiabilitatea sistemului. Dacă P_{cor} este probabilitatea ca sistemul să cadă în totalitate atunci

$$R_{M\text{-din-}N\_\text{cor}}(t) = (1 - P_{\text{cor}}) \sum_{i=0}^{N-M} C_N^i [1 - R(t)]^i [R(t)]^{N-i}$$

Moduri de corelare în sistemele *M*-din-*N*

Dacă sistemul nu este proiectat cu grijă, factorul de cădere corelată poate domina probabilitatea generală de cădere. Există moduri diferite de corelare între căderile componentelor, care nu înseamnă în mod necesar o cădere generală antrenată de căderea unei singure componente. Ratele de defectare corelate sunt extrem de dificil de estimat. Din acest motiv, de aici înainte căderile componentelor vor fi considerate statistic independente: un modul nu alterează starea de functionare a celorlalte în momentul căderii lui sau ulterior.

TMR (Triple Modular Redundant) – cluster redundant cu trei module

Sistemul TMR este probabil cel mai important sistem de tipul *M*-din-*N*. Este vorba, asa cum s-a mai spus, de cazul valorilor $M = 2$, $N = 3$ si sistemul este functional dacă cel putin două din componente sunt functionale. Un voter retine si eventual transmite totdeauna iesirea majoritară. Nu trebuie omis faptul că voterul poate cădea el însusi. Fiabilitatea lui este notată $R_{\text{vot}}(t)$ si este strict subunitară. Expresia fiabilitătii sistemului TMR este

$$R_{\text{TMR}}(t) = R_{\text{vot}}(t) \sum_{i=0}^{1} C_3^i [1 - R(t)]^i [R(t)]^{3-i} = R_{\text{vot}}(t) \{3[R(t)]^2 - 2[R(t)]^3\}$$

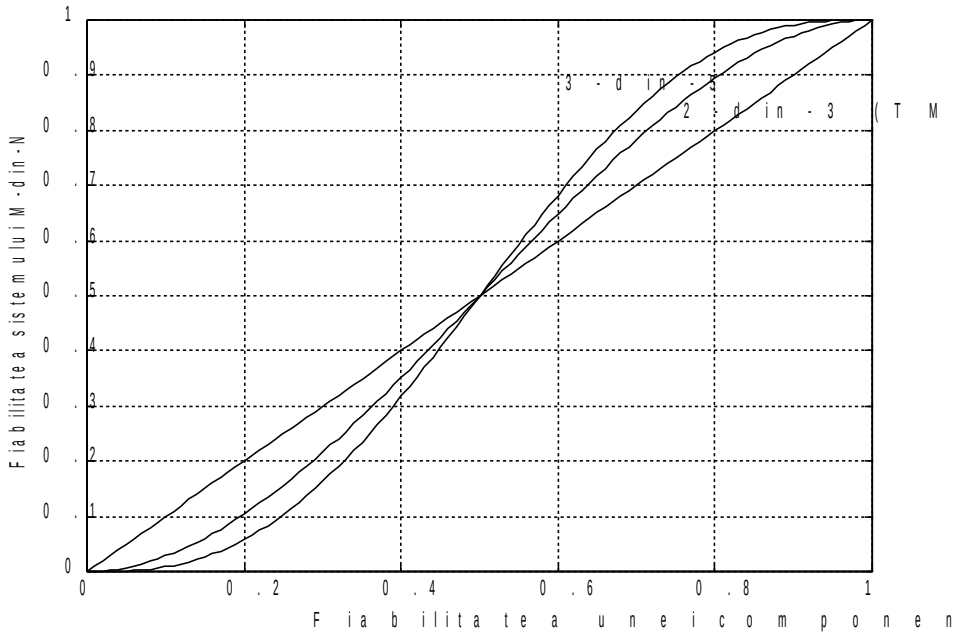
Pentru un sistem TMR cu module care au rate de defectare constante, λ, si în consecintă fiabilităti modulare $R(t) = e^{-\lambda t}$ identice, dacă defectarea voterului este exclusă, $R_{\text{vot}}(t) = 1$, se obtine expresia particulară

$$R_{\text{TMR}}(t) = 3e^{-2\lambda t} - 2e^{-3\lambda t}$$

si timpul mediu până la cădere

$$MTTF_{\text{TMR}} = \int_0^{\infty} R_{TMR}(t)dt = 5/(6\lambda) < 1/\lambda = MTTF_{\text{simplex}}$$

Pare surprinzător: o structură mai sofisticată pare a fi mai putin viabilă (nu fiabilă, viabilă!) decât o componentă a ei. Adevărul exprimat de aceste valori este dincolo de orice îndoială. Dacă se mai adaugă si posibilitatea reală de defectare a voterului, diferenta dintre TMR si modulul unic (simplex) se măreste.



Pentru sublinierea avantajelor pe care le prezintă schemele redundante de tipul TMR sau NMR sunt necesare unele explicatii în termeni de fiabilitate.

NMR (*N*-Modular Redundant) – cluster redundant din *N* module

Sistemul NMR este un cluster *M*-din-*N* cu *N* impar si cu $M = (N + 1)/2$. Se admite din nou că rata de defectare a voterului este neglijabilă, $R_{vot}(t) = 1$.
Graficul alăturat arată dependenta fiabilitătii globale de fiabilitatea unui modul pentru cazurile $N = 3$ si $N = 5$. După cum se observă, pentru fiabilităti ale unui singur modul $R < 0{,}5$, în ambele cazuri redundanta sistemului devine dezavantajoasă. Observatia este generalizabilă si pentru numere *N* mai mari. Uzual însă $R >> 0{,}5$ si atunci triplexul si sistemul 3-din-5, ca orice sistem *M*-din-*N* oferă câstiguri de fiabilitate semnificative.

TMR – compensarea defectelor

Ipoteza de lucru de până acum este "fiecare cădere a voterului produce o iesire eronată si orice cădere a două module componente este fatală". Sunt posibile însă si alte situatii si iată un contraexemplu: un modul produce permanent un 1 logic si un al doilea modul generează permanent un 0 logic; sistemul TMR va functiona adecvat relativ la acest bit. Este vorba aici de o compensare a defectelor. O situatie similară poate apărea cu referire la anumite defecte în circuitul voterului.
Un alt exemplu de compensare a defectelor este acela al "erorilor care nu se suprapun" (non-overlapping faults): un modul poate avea un sumator cu defect si un alt modul poate avea un multiplicator cu defect. Dacă circuitele sunt disjuncte, este improbabil ca ele să genereze simultan iesiri gresite.

Votere

Un voter primeste intrările $X_1, X_2, \ldots, X_N$ de la un cluster *M*-din-*N* si generează o iesire reprezentativă. Voterul cel mai simplu face comparatia iesirilor bit-cu-bit si iesirea rezultă bit-cu-bit pe baza votului majoritar. Această schemă functionează numai când toate procesoarele functionale generează iesiri care se potrivesc bib-cu-bit în totalitate. Dar pentru aceasta procesoarele trebuie să fie identice si să utilizeze acelasi software. Altminteri, două iesiri corecte pot diferi întrucâtva prin bitii cel mai putin semnificativi.

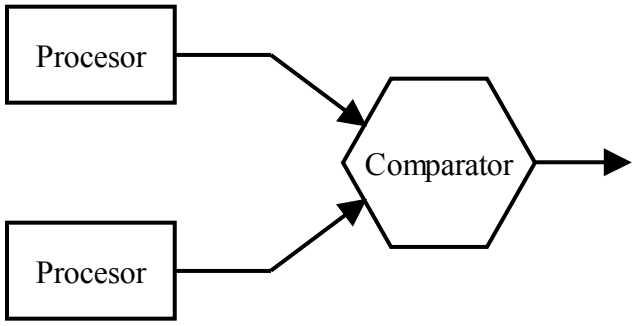
Votarea pluralităţii

Două iesiri *X* si *Y* sunt declarate *practic* identice dacă $|x - y| < \delta$ pentru un δ precizat. Un voter cu *k*-pluralitate caută o multime de cel putin *k* iesiri *practic* identice si ia una din ele, oarecare (sau mediana lor), ca reprezentativă. De

exemplu dacă δ = 0,1 si sunt cinci iesiri 1,10 1,11 1,32 1,49 si 3,00, submultimea {1,10 1,11} va fi retinută de un voter de pluralitate 2.

Sisteme duplex

Într-un sistem duplex (v.figura), ambele procesoare execută acelasi task.



Dacă iesirile sunt concordante, rezultatul este considerat corect. Dacă iesirile sunt diferite, nu poate fi identificat procesorul care a gresit. Un software de nivel mai înalt trebuie să decidă cum trebuie tratată eroarea. Acest lucru se poate face utilizând una din metodele date în continuare.
Prima metodă constă în aplicarea unor asa-numite teste de acceptare. Test de acceptare înseamnă o verificare a domeniului de valori pentru fiecare din iesirile procesoarelor. De exemplu, presiunea într-un boiler trebuie să fie într-un anumit domeniu de valori cunoscut. Se folosesc asadar informatii de natură semantică relativ la task-ul în derulare pentru a preciza care valori ale iesirii indică prezenta unei erori.
Cum se poate selecta gama de valori acceptabilă? Sunt două criterii concurente: *sensibilitatea* si *specificitatea*.
Domeniul de acceptare poate fi îngust ceea ce echivalează cu o probabilitate înaltă de a identifica o iesire eronată, dar si o probabilitate ridicată de a aprecia gresit o iesire bună ca fiind eronată (falsul pozitiv). Domeniul de acceptare poate fi larg si atunci probabilitătile sunt reduse pentru ambele situatii arătate în fraza anterioară. În contextul acestor situatii posibile (nuantat!) *sensibilitatea* este probabilitatea ca testul să recunoască o iesire eronată ca o iesire eronată, iar *specificitatea* este probabilitatea ca testul să recunoască o iesire corectă ca o iesire corectă. Cu aceste definitii, un domeniu de acceptare îngust se asociază cu o sensibilitate înaltă dar cu specificitate scăzută. Un domeniu de acceptare larg înseamnă o sensibilitate scăzută dar o specificitate înaltă.
A doua metodă apelează la o anumită testare la care sunt supuse ambele procesoare. Procesorul care nu trece testul este identificat ca fiind cel cu defect. Testele practicate în realitate nu sunt niciodată perfecte. De aceea se vorbeste despre *acoperirea* testului care este acelasi lucru cu sensibilitatea testului, adică cu probabilitatea ca testul să identifice un procesor defect ca defect.

Transparenta testului, o altă caracteristică a testului aplicat celor două procesoare este complementara acoperirii testului si exprimă probabilitatea ca testul să treacă un procesor defect în categoria "corect".
Metoda a treia este asa-numita recuperare târzie (forward recovery). Această metodă utilizează un al treilea procesor pentru repetarea calculului executat de duplex. Dacă numai unul din cele trei procesoare este defect atunci cel care este în discordie cu procesorul al treilea este apreciat ca defect.
Practic este posibil să se folosească o combinatie a acestor trei metode. Testul de acceptare este cel mai rapid de executat dar adesea este si cel mai putin sensibil.

Fiabilitatea unui duplex.

Cele două procesoare active sunt presupuse a fi identice, fiecare cu fiabilitatea $R(t)$. Durata de viată a duplexului este timpul scurs până când ambele procesoare cad. Se utilizează si un factor de acoperire c, care reprezintă probabilitatea ca un procesor defect să fie diagnosticat corect, identificat si deconectat. Dacă $R_{duplex}(t)$ este fiabilitatea sistemului duplex, atunci

$$R_{duplex}(t) = R_{comp}(t)\,[R^2(t) + 2cR(t)(1 - R(t))]$$

S-a tinut seamă si de $R_{comp}(t)$, care este fiabilitatea comparatorului.
Pentru un duplex cu ratele de defectare ale procesoarelor componente constante (λ) si cu un comparator ideal, adică cu $R_{comp}(t) = 1$, fiabilitatea duplexului are expresia

$$R_{duplex}(t) = e^{-2\lambda t} + 2ce^{-\lambda t}\,(1 - e^{-\lambda t})$$

si durata lui medie de viată este

$$MTTF_{duplex} = 1/(2\lambda) + c/\lambda$$

Duplex cu redundante

Prin duplex cu redundante se întelege un duplex cu două procesoare identice active si un număr (nelimitat) de procesoare în rezervă, înlocuitoare potentiale ale celui care se defectează. Când un procesor cade, căderea lui este detectată cu probabilitatea P_d si un procesor nou îl înlocuieste pe cel defect. Probabilitatea ca acest proces[2] de substituire să producă căderea întregului duplex este $1 - P_s$. Procesul indentificator al erorii este considerat instantaneu, rezervele sunt presupuse a fi totdeauna functionale.
Imediat mai jos se prezintă un model pentru un duplex cu redundanțe. Se admite că fiecare procesor are o rată de defectare constantă λ. Durata de viată a unui procesor are, asadar, o distributie exponentială cu parametrul λ. Timpul

[2] De cele mai multe ori cuvintele *procesor* si *proces* exprimă acelasi lucru în sensul că un procesor găzduieste un proces. Alteori, termenul proces semnifică altceva decât un proces pe un procesor. Cititorul va putea să distingă singur între cele două situatii. Explicatiile vor fi detaliate numai acolo unde se consideră că este cazul.

între două defectări succesive ale unui procesor oarecare este si acesta distribuit exponential cu acelasi parametru λ. Se notează:

$M(t)$ – numărul de defectări ale unuia dintre procesoarele din duplex în intervalul $[0, t]$;

$N(t)$ – numărul de defectări ale sistemului duplex în intervalul $[0, t]$.

Relativ la distributia lui $M(t)$ se parcurge rationamentul care urmează.

Fie Δt un interval de timp mic, atât de mic încât probabilitatea ca în scurtul răstimp Δt să se producă mai mult de o cădere să fie neglijabilă. $M(t + \Delta t) = n$ fie dacă $M(t) = n - 1$ si în intervalul Δt are loc o cădere, fie dacă $M(t) = n$ si în intervalul Δt nu se produce nici o cădere. Se scrie

$$\Pr[M(t + \Delta t) = n] \approx \Pr[M(t) = n - 1]\lambda\Delta t + \Pr[M(t) = n](1 - \lambda\Delta t)$$

având în vedere o firească proportionalitate (aproape riguroasă) a probabilității de manifestare a unui defect în acel scurt interval, cu durata Δt si cu parametrul λ. În relatie se identifică cu usurintă si probabilitatea evenimentului contrar, lipsa producerii unui defect în acel interval Δt, care este $1 - \lambda\Delta t$. Această relatie produce prin trecere la limită, $\Delta t \to 0$, ecuatia diferențială

$$d\Pr[M(t) = n]/dt = -\lambda\Pr[M(t) = n] + \lambda\Pr[M(t) = n - 1]$$

care se poate scrie pentru fiecare n, întrucâtva diferit pentru $n = 0$.

Sistemul de ecuatii diferentiale rezultat completat cu conditiile initiale $\Pr[M(0) = n] = 0$ pentru $n \geq 1$ si $\Pr[M(0) = 0] = 1$ poate fi rezolvat, iar solutia lui este

$$\Pr[M(t) = n] = e^{-\lambda t}(\lambda t)^n/n! \text{ pentru } n = 0, 1, 2, \ldots$$

Prin urmare, $M(t)$ are o distributie Poisson de paramentru λt. Se admite promptitudinea, chiar instantaneitatea în ceea ce priveste restabilirea functionalitătii sistemului, astfel încât după o defectare si aparitia unei erori el functionează din nou un interval de timp fără erori, fără manifestarea vreunui defect.

Ca o aplicatie a celor stabilite mai sus, urmează calculul fiabilitătii pentru duplexul cu redundante.

Duplexul se compune din două procesoare, ceea ce face rata de defectare dublă, 2λ. Rata de defectare a comparatorului se consideră neglijabilă. Probabilitatea a n căderi ale duplexului în intervalul $[0, t]$ este

$$\Pr[N(t) = n] = e^{-2\lambda t}(2\lambda t)^n/n! \text{ pentru } n = 0, 1, 2, \ldots$$

Pentru ca duplexul să nu cadă în ansamblu său, fiecare din căderile particulare trebuie detectate si înlocuite cu succes. Aceasta se produce cu probabilitatea $c = P_d P_s$, produsul a două probabilităti definite mai devreme, cea a detectării corecte a modulului defect si cea de functionare a duplexului, probabilităti a două evenimente independente. Pentru n căderi (independente) probabilitatea devine c^n.

Asadar, fiabilitatea unui duplex în conditiile mentionate este

$$R_{duplex}(t) = \sum_{n=0}^{\infty} \Pr(n \text{ căderi})c^n = \sum_{n=0}^{\infty} \exp(-2\lambda t)(2\lambda t)^n c^n/n! =$$

$$= \exp(-2\lambda t) \sum_{n=0}^{\infty} (2\lambda tc)^n/n! = \exp(-2\lambda t)\exp(2\lambda tc)$$

$$R_{duplex}(t) = \exp[-2\lambda(1-c)t]$$

Iată acum o cale alternativă de stabilire a formulei pentru fiabilitatea unui duplex. Se admite că procesoarele cad la rata λ si atunci rata căderii duplexului este 2λ. Probabilitatea ca fiecare cădere să fie tratată cu succes este c si $(1 - c)$ este probabilitatea ca duplexul să cadă. Căderile care fac duplexul disfunct apar la o rată $2\lambda(1 - c)$. Functia de fiabilitate a sistemului este aceeasi ca mai sus, adică $\exp[-2\lambda(1-c)t]$.

Fiabilitatea unor sisteme mai complexe

Printre sistemele mai complexe se află sistemele NMR în care procesoarele care cad sunt identificate si înlocuite dintr-un stoc indefinit de rezerve identice. Dacă rezervele sunt finite ca mumăr, atunci sumele din expresia fiabilitătii sunt limitate la acel număr de rezerve si nu infinite.

Se mentionează si alte variante ale sistemelor duplex: un procesor este activ în timp ce al doilea este în *standby*. Se admite că procesoarele pot fi reparate atunci când devin nefunctionale.

În cazul sistemelor mai complexe, argumentele de natură combinatorială pot fi insuficiente pentru calculul fiabilitătii. Dacă ratele de defectare sunt constante, pentru evaluarea fiabilitătii se pot utiliza modelele Markov.

Lanturi Markov (scurtă introducere)

Modelele Markov furnizează o tratare structurată pentru deducerea functiilor de fiabilitate ale sistemelor complexe. Un lant Markov este un proces stochastic $X(t)$, o secventă de variabile aleatoare indexate de timpul t, cu o structură probabilistică specială. Pentru ca un proces stochastic să fie un lant Markov, comportarea lui viitoare trebuie să depindă numai de starea lui prezentă si nu de alte stări trecute. $X(t + s)$ depinde de $X(t)$, dar dându-se $X(t)$ si $X(t + s)$ acestea nu depind nemijlocit de orice $X(\tau)$ pentru $\tau < t$. Dacă $X(t) = i$ se spune că la timpul t lantul este în starea i.

În continuare se tratează exclusiv lanturi Markov cu timp continuu ($0 \le t \le \infty$) si cu stări discrete, $X(t) = 0, 1, 2, \ldots$

Lanturile Markov au o interpretare probabilistică consistentă. Relatia

$$\Pr[X(t+s) = j / X(t) = i, X(\tau) = k] = \Pr[X(t+s) = j / X(t) = i] \quad (\tau < t)$$

exprimă prin probabilităti conditionate dependenta stării următoare exclusiv de starea precedentă. Sistemul, odată ajuns în starea i, rămâne în acea stare un timp aleator care este distribuit exponential cu parametrul λ_i. Se poate spune că sistemul modelat de un lant Markov are o rată constantă λ_i de a părăsi starea i.

Probabilitatea ca lantul/sistemul să treacă din starea i în starea j se notează cu P_{ij}. Rata trecerii din starea i în starea j este atunci $\lambda_{ij} = P_{ij}\ \lambda_i$ si deoarece $\sum_{j \neq i} P_{ij} = 1$ rezultă că $\sum_{j \neq i} \lambda_{ij} = \lambda_i$.

Probabilităti asociate stărilor

Se notează cu $P_i(t)$ probabilitatea ca procesul/sistemul să fie la momentul t în starea i, starea lui initială (la $t = 0$) fiind i_0. Se scriu ecuatii diferentiale pentru $P_i(t)$, ($i = 0, 1, 2, \ldots$). Dacă la momentul t sistemul se află în starea j, după un interval scurt Δt sistemul poate ajunge în starea i în unul din următoarele cazuri:

- El a fost la momentul t în starea i, asadar $j \equiv i$, si nu si-a schimbat starea după Δt. Acest fapt are asociată probabilitatea $P_i(t)(1 - \lambda_i \Delta t)$.
- El a fost la momentul t în starea j, (oarecare, $j \neq i$) si a trecut în intervalul Δt din acea stare în starea i, eveniment care se petrece cu probabilitatea $P_j(t)\lambda_{ji}\Delta t$.

În acest rationament s-a admis că intervalul Δt este suficient de mic pentru ca producerea a mai mult de o tranzitie pe durata lui să fie practic exclusă. Însumând acum termenii asociati tuturor tranzitiilor posibile, pentru diferiti j, se obtine relatia

$$P_i(t + \Delta t) \approx P_i(t)(1 - \lambda_i \Delta t) + \sum_{j \neq i} P_j(t)\lambda_{ji}\Delta t$$

După putină prelucrare algebrică, trecerea la limită $\Delta t \to 0$ produce următoarele ecuatii diferentiale în probabilitătile $P_i(t)$:

$$\frac{dP_i(t)}{dt} = -\lambda_i P_i(t) + \sum_{j \neq i} \lambda_{ji} P_j(t)$$

cu i (si j) parcurgând multimea de stări posibile ale lantului. Si deoarece $\sum_{j \neq i} \lambda_{ij} = \lambda_i$ se poate rescrie

$$\frac{dP_i(t)}{dt} = -\sum_{j \neq i} \lambda_{ij} P_i(t) + \sum_{j \neq i} \lambda_{ji} P_j(t)$$

Acum aceste ecuatii pot fi rezolvate pentru $i = 0, 1, 2, \ldots$, cu conditiile initiale $P_{i_0}(0) = 1$ si $P_i(0) = 0$ pentru $i \neq i_0$.

Un exemplu: Duplex cu unul din module în *standby*: un procesor este activ si unul este în *standby* (în rezervă caldă) si este conectat când cel activ cade. Rata căderii pentru procesorul activ este λ, constantă. Se ia în considerare si un factor de acoperire c, care este probabilitatea ca o cădere a procesorului activ să fie detectată corect si conectarea rezervei să fie reusită. Lantul Markov în reprezentare grafică este dat în figură.

Ecuatiile diferentiale pentru duplexul cu un modul în standby sunt

$$\frac{dP_i(t)}{dt} = -P_i(t)\sum_{j \neq i} \lambda_{ij} + \sum_{j \neq i} \lambda_{ji} P_j(t) \quad i = 0, 1, 2$$

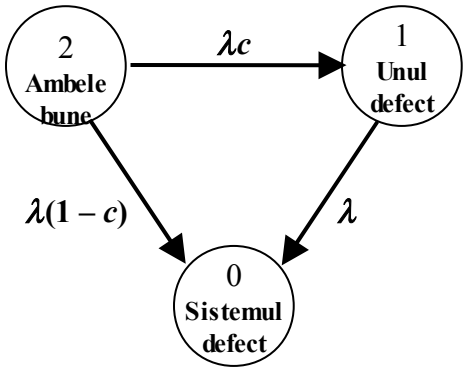
cu detalierea

$$\frac{dP_0(t)}{dt} = \lambda P_1(t) + \lambda(1 - c)P_2(t)$$

$$\frac{dP_1(t)}{dt} = -\lambda P_1(t) + \lambda c P_2(t)$$

$$\frac{dP_2(t)}{dt} = -\lambda P_2(t)$$

si conditii initiale sunt: $P_2(0) = 1$, $P_1(0) = P_0(0) = 0$.



si conditii initiale sunt: $P_2(0) = 1$, $P_1(0) = P_0(0) = 0$.

Pentru a evalua fiabilitatea unui duplex care are un modul în *standby*, mai întâi solutia ecuatiilor diferentiale

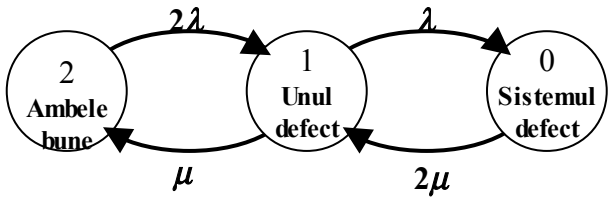
$$P_0(t) = 1 - P_1(t) - P_2(t)$$
$$P_1(t) = c\lambda t \exp(-\lambda t)$$
$$P_2(t) = \exp(-\lambda t)$$

apoi functia de fiabilitate

$$R_{sistem}(t) = 1 - P_0(t) = P_2(t) + P_1(t) = \exp(-\lambda t) + c\lambda t \exp(-\lambda t)$$

Propunem cititorului ca exercitiu obtinerea acestei expresii pe baza unor considerente combinatoriale.

Este adus acum în discutie un sistem duplex care beneficiază de reparatii. Este vorba de două procesoare active, fiecare cu rata de defectare λ si cu rata reparatiilor μ. Modelul Markov al unui astfel de sistem este dat în figura alăturată



Ecuatiile diferentiale în forma generală

$$\frac{dP_i(t)}{dt} = -P_i(t)\sum_{j \neq i} \lambda_{ij} + \sum_{j \neq i} \lambda_{ji} P_j(t)$$

se particularizează pentru cazul în discutie, pentru $i, j = 0, 1, 2$, sub forma

$$\frac{dP_2(t)}{dt} = -2\lambda P_2(t) + \mu P_1(t)$$

$$\frac{dP_1(t)}{dt} = 2\lambda P_2(t) + 2\mu P_0(t) - (\lambda + \mu)P_1(t)$$

$$\frac{dP_0(t)}{dt} = \lambda P_1(t) - 2\mu P_0(t)$$

Conditiile initiale sunt $P_2(0) = 1$, $P_1(0) = P_0(0) = 0$
Solutia acestor ecuatii diferentiale este

$$P_2(t) = \mu^2/(\lambda + \mu)^2 + 2\lambda\mu/(\lambda + \mu)^2 e^{-(\lambda+\mu)t} + \lambda^2/(\lambda + \mu)^2 e^{-2(\lambda+\mu)t}$$

$$P_1(t) = 2\lambda\mu/(\lambda + \mu)^2 + 2\lambda(\lambda - \mu)/(\lambda + \mu)^2 e^{-(\lambda+\mu)t} - 2\lambda^2/(\lambda + \mu)^2 e^{-2(\lambda+\mu)t}$$

$$P_0(t) = 1 - P_2(t) - P_1(t)$$

adică probabilitătile celor trei stări posibile ca functii de timp.
În sistemele duplex fără reparatii, indicatorul cel mai important referitor la functionarea (sau nefunctionarea) lor este fiabilitatea. Pentru sistemele duplex cu reparatii, toti cei trei indicatori – fiabilitatea, disponibilitatea si disponibilitatea punctuală – sunt la fel de importanti. Evaluarea acestora se face în maniera descrisă imediat.

- Disponibilitatea punctuală – $A_p(t)$ – Pr(sistemul la timpul t este operational) = $1 - P_0(t)$.
- Fiabilitatea – $R(t)$ = Pr(sistemul este operational pe intervalul $[0, t]$) – prin eliminarea tranzitiei de la starea 0 la starea 1, rezolvând ecuatiile diferentiale noi care rezultă.
- Disponibilitatea – $A(t)$ – se ia media pe intervalul $[0, t]$ a proportiei din timp în care sistemul este operational. Acesta este un indicator mai relevant pentru sistemele cu reparatii.

Accesibilitatea stationară

Se calculează accesibilitatea stationară, $A(\infty)$ sau A, care este proportia din timp, pe durată îndelungată, cât sistemul este operational.
Se calculează mai întâi probabilitătile stationare, $P_2(\infty)$, $P_1(\infty)$ si $P_0(\infty)$ sau P_2, P_1, P_0. Probabilitătile stationare se pot calcula în două moduri: fie prin trecerea la limită, $t \to \infty$, în expresiile $P_i(t)$, $i = 0, 1, 2$, fie prin anularea tuturor derivatelor $dP_i(t)/dt = 0$, $i = 0, 1, 2$ si rezolvarea sistemului de ecuatii algebrice liniare în P_i cu adaosul $P_2 + P_1 + P_0 = 1$. Rezultă

$$A = 1 - P_0 = P_1 + P_2$$

Starea stationară a unui duplex care admite reparatii

Probabilitătile stationare sunt

$$P_2(t) = \mu^2/(\lambda + \mu)^2$$

$$P_1(t) = 2\lambda\mu/(\lambda + \mu)^2$$

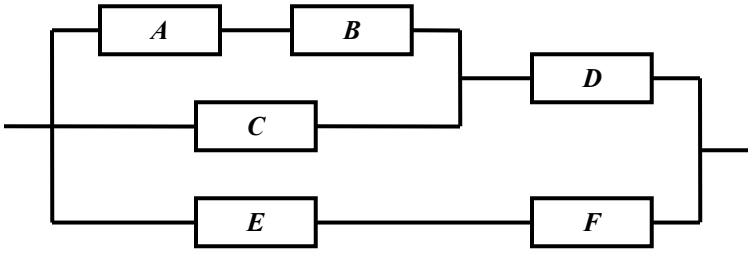
$$P_0(t) = \lambda^2/(\lambda + \mu)^2$$

si accesibilitatea stationară este

$$A = A(\infty) = P_2 + P_1 = 1 - P_0 = (\mu^2 + 2\lambda\mu)/(\lambda + \mu)^2 = 1 - \lambda^2/(\lambda + \mu)^2$$

Structuri complexe

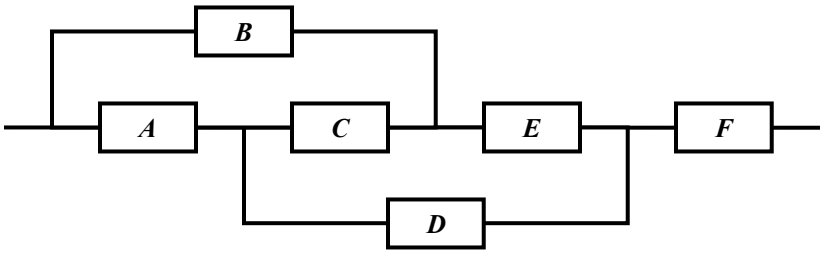
Multe din structurile mai complexe pot fi reduse în etape la structurile canonice serie sau paralel. Pentru ilustrarea acestei idei se dă sistemul din figura care urmează.



Modulele *A*, *B* sunt legate serie si reuniunea lor (*A*, *B*) este un subsistem pentru care se poate calcula fiabilitatea prin relatia consacrată. Modulul *C* si "modulul" (*A*, *B*) sunt asezate paralel si, din nou, există formula de calcul de la sistemele paralel. În continuare, modulul *D* si "modulul" (*A*, *B*, *C*) sunt serie, tot aşa, separat, modulele *E* si *F* sunt înseriate si evaluările sunt posibile cu formulele cunoscute. În cele din urmă, "modulele" (*A*, *B*, *C*, *D*) si (*E*, *F*) sunt în paralel si din nou se poate evalua fiabilitatea ansamblului. Prin urmare, într-un număr finit de pasi se poate evalua prin calcule simple fiabilitatea sistemului alcătuit din modulele {*A*, *B*, *C*, *D*, *E*, *F*} si analog pentru orice sistem decompozabil succesiv în subsisteme serie si/sau paralel.

Sisteme care nu sunt nici serie, nici paralel, nici succesiv reductibile la tipurile serie si/sau paralel.

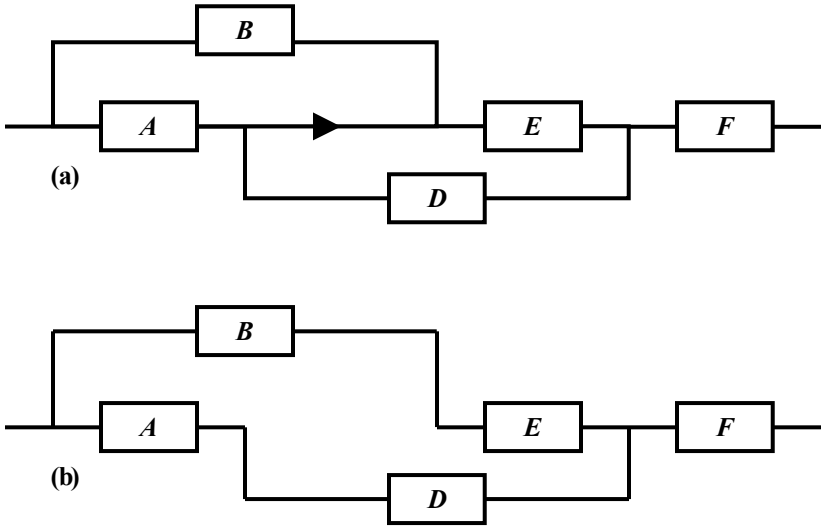
Iată acum un sistem mai complicat. Se poate verifica usor că tentativa de tratare prin descompunere după reteta dată la exemplul introductiv este sortită eseculuí.



Si în acest caz, fiecare drum/cale de la stânga la dreapta reprezintă o configuratie care permite sistemului să opereze corect, de pildă *A*-*D*-*F*.
Fiabilitatea sistemului poate fi calculată prin dezvoltarea în jurul unui singur modul *i*, $i \in \{A, B, C, D, E, F\}$ conform relatiei cu probabilităti conditionate

$$R_{sistem} = R_i \Pr(\text{sistemul lucrează}/i \text{ este functional}) +$$
$$+ (1 - R_i) \Pr(\text{sistemul lucrează}/i \text{ este nefunctional})$$

Se figurează mai jos două grafuri derivate din cel de mai sus: în varianta (a) modulul *i* este functional, în varianta (b) modulul *i* este nefunctional, în particular $i \equiv C$. Modulul *C* a fost selectat pentru ca cele două structuri reprezentate de cele două grafuri să fie mai apropiate de structurile simple decompozabile serie-paralel, paralel-serie etc.



Dezvoltarea în jurul unui modul poate fi repetată (etapizat) în jurul altui modul si apoi a altui modul s.a.m.d., până când grafurile (diagramele de semnal) rezultate sunt de tipurile mai simple serie-paralel, paralel-serie etc. Figura (a) de pildă necesită încă o dezvoltare în jurul modulului *E*. Graful din figura (a) nu trebuie văzut ca fiind compus – între altele – dintr-o grupă (*A*, *B*) în paralel, conectată în serie cu grupa (*D*, *E*) la rândul ei în paralel. O asemenea tratare ar admite calea *B*-*C*-*D*-*F* care nu este o cale validă decât dacă modulul *C* ar fi bidirectional simetric. Acest detaliu este subliniat în figură prin săgeata atasată arcului care înlocuieste pe *C* când modulul este functional.
Dezvoltare evaluărilor în jurul modulelor *C* si *E* din exemplul dat urmează etapele de mai jos:

$$R_{sistem} = R_C \Pr(\text{sistemul lucrează}/C \text{ este operational}) +$$
$$+ (1 - R_C)R_F[1 - (1 - R_A R_D)(1 - R_B R_E)]$$

Termenul al doilea s-a putut scrie deja după dezvoltarea din jurul modulului *C* pentru că graful (b) este posibil a fi cuprins imediat în una din schemele simple

decompozabile serie-paralel, paralel-serie etc. Prin dezvoltare în jurul lui E rezultă si factorul necunoscut al primului termen:

$$\text{Pr(sistemul lucrează/}C\text{ este operational)} =$$
$$= R_E R_F[1 - (1 - R_A)(1 - R_B)] + (1 - R_E)R_A R_D R_F$$

Printr-o simplă înlocuire si putină prelucrare algebrică se ajunge la expresia

$$R_{sistem} = R_C\,[R_E R_F(R_A + R_B - R_A R_B) + (1 - R_E)R_A R_D R_F] +$$
$$+ (1 - R_C)R_F(R_A R_D + R_B R_E - R_A R_D R_B R_E)$$

Exemplu: $R_A = R_B = R_C = R_D = R_E = R_F = R$ conduce la

$$R_{sistem} = R^3(R^3 - 3R^2 + R + 2)$$

Dacă structura sistemului este mai complicată, calculele pot deveni la rândul lor foarte complicate. În asemenea situatii se pot stabili limitele superioară si inferioară pentru fiabilitatea sistemului. O margine superioară este

$$R_{sistem} \le 1 - \Pi(1 - R_{calea\_i})$$

cu R_{calea_i} fiabilitatea subsistemului alcătuit din modulele serie de pe calea i. În produsul din formulă apar factori asociati tuturor căilor paralele. Pentru sistemul exemplificat mai sus: căile care fac sistemul functional sunt în număr de trei, *A-D-F*, *B-E-F* si *A-C-E-F* si atunci

$$R_{sistem} \le 1 - (1 - R_A R_D R_F)(\,1 - R_B R_E R_F)(1 - R_A R_C R_E R_F)$$

În particular, dacă $R_A = R_B = R_C = R_D = R_E = R_F = R$ atunci

$$R_{sistem} \le R^3(R^7 - 2R^4 - R^3 + R + 2)$$

Marginea inferioară a fiabilitătii se calculează pe baza asa-numitelor *multimi de tăietură minimă* a diagramei-graf a sistemului. O multime de tăietură minimă este o listă minimală de module constituită astfel încât eliminarea (datorată defectării) a tuturor modulelor din acea listă să facă sistemul disfunct. În cazul în discutie, multimi de tăietură minimală sunt $\{F\}$, $\{A, B\}$, $\{A, E\}$, $\{D, E\}$ si $\{B, C, D\}$.

Marginea inferioară a fiabilitătii sistemului din exemplul dat este partea dreaptă a inegalitătii

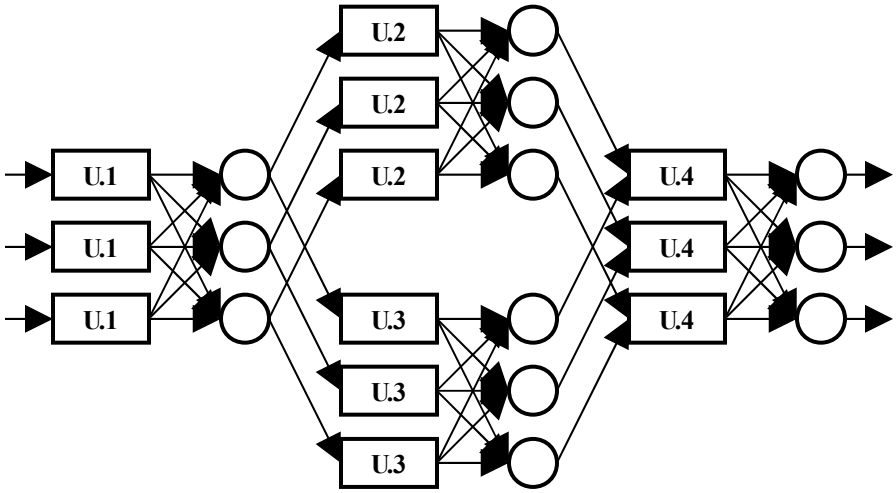
$$R_{sistem} \ge \Pi(1 - Q_{taietura\_i})$$

cu $Q_{taietura\_i}$ probabilitatea ca modulele din tăietura minimă i să fie toate defecte. În particular, dacă $R_A = R_B = R_C = R_D = R_E = R_F = R$, atunci

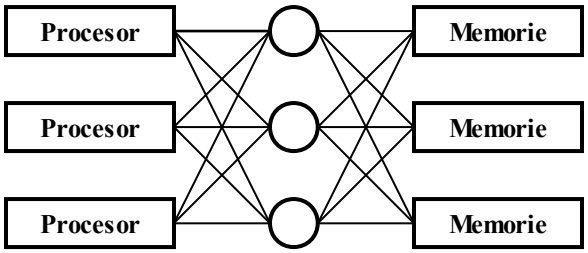
$$R_{sistem} \ge R^5(24 - 60R + 62R^2 - 33R^3 + 9R^4 - R^5)$$

Este readus acum în discutie sistemul NMR (N-Modular Redundant). Figura alăturată ilustrează cazul unor redundante modulare la nivel de unităti.

Voterele (reprezentate prin cercuri) nu mai sunt elemente atât de critice ca în sistemul NMR discutat anterior; situatia cu un singur voter defect nu este mai rea decât situatia în care o singură unitate este defectă. Nivelul la care se aplică replicarea si votarea poate fi încă mai scăzut cu relativa risipă a unor votere aditionale care fac să crească dimensiunea sistemului si întârzierea lui.



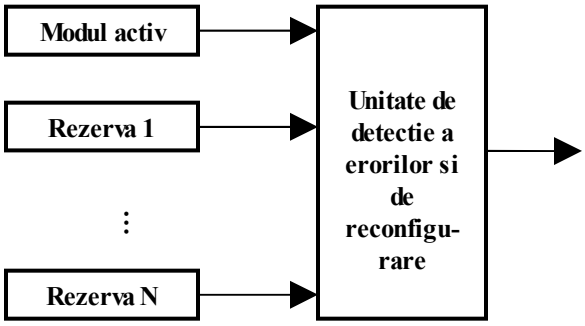
Iată acum un sistem de procesoare si memorii triplicat.



Toate comunicatiile (în ambele directii) între procesoarele triplicate si memoriile triplicate merg prin vot majoritar (voterele, de data aceasta bidirectionale sunt reprezentate si aici prin cercuri). Această organizare are desigur o fiabilitate mai înaltă decât un vot majoritar simplu în structura triplicată procesoare-memorii.

Redundante active/dinamice

Variatiile de mai sus ale sistemelor NMR (*N*-Modular Redundant) au neajunsul că angajează prea mult hardware, este adevărat, pentru a masca instantaneu erorile.
Rezultatele eronate pot fi acceptate temporar dacă sistemul poate detecta erorile si se poate reconfigura de unul singur.
Solutia este înlocuirea modulului defect printr-o rezervă care functionează corect. Iată în figura alăturată o schemă cu redundante active (sau dinamice).



Fiabilitatea sistemului cu rezerve active se evaluează observând că dacă toate modulele de rezervă sunt active (sub tensiune, alimentate) ele au aceeasi rată de defectare si, întocmai ca la sistemul fundamental paralel, fiabilitatea sistemului capabil a detecta disfunctiile si de a se reconfigura este

$$R_{dinamic}(t) = R_{det}(t)[1 - (1 - R(t))^N]$$

În această expresie, s-a notat cu $R(t)$ fiabilitatea unui modul si cu $R_{det}(t)$ fiabilitatea unității de detectie si reconfigurare.

Fiabilitatea pentru cazul rezervelor în *standby*, adică pentru o foarte putin probabilă cădere a modulelor de rezervă (de pildă pentru că nu sunt alimentate, nu sunt puse sub tensiune din motive de economie de energie) fiabilitatea unui sistem cu un modul activ si altul ca rezervă în standby este

$$R_{dinamic}(t) = R(t) + cR(t)\,[1 - R(t)]$$

cu c un asa-numit *factor de acoperire*, altfel spus o probabilitate ca modulul activ să fie corect diagnosticat ca defect si să fie deconectat, iar cel de rezervă, functional, să fie conectat cu succes.

Prin generalizare, în cazul cu N rezerve

$$R_{dinamic}(t) = R(t) \sum_{k=0}^{N} c^k\,[1 - R(t)]^k$$

Este de discutat aici dacă momentele căderii modulului activ au sau nu vreo importantă (exercitiu).

Redundanta hibridă

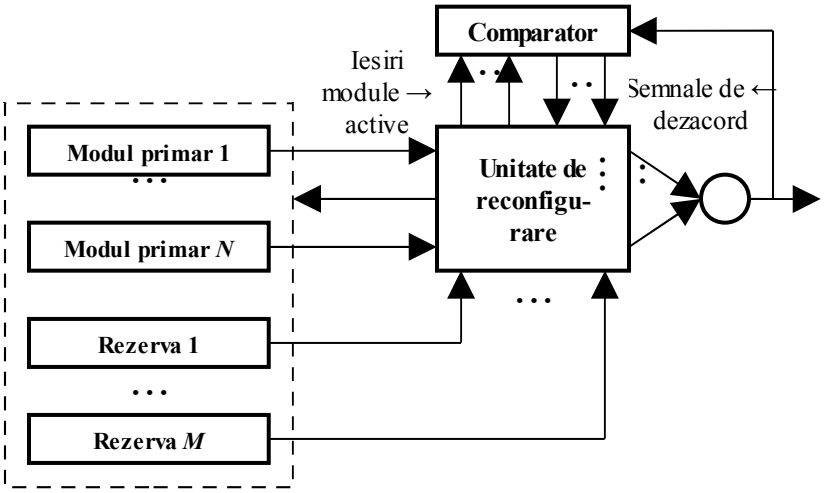
După cum s-a arătat mai devreme, un sistem de tipul NMR maschează defectele permanente si intermitente dar fiabilitatea lui pentru durate lungi de utilizare scade sub aceea a unui singur modul. Redundanta hibridă depăseste acest inconvenient prin adăugarea de module de rezervă pentru înlocuirea modulelor active imediat ce ele devin disfuncte.

Un sistem hibrid constă într-un miez de N procesoare (NMR) si M rezerve.

În cazul unei redundante hibride, fiabilitatea obtinută printr-o grupare de tipul TMR sustinută de M rezerve are expresia

$$R_{hibrid}(t) = R_{voter}(t)R_{reconf}(t)\{1 - mR(t)[1 - R(t)]^{n-1} - [1 - R(t)]^n\}$$

cu $n = M + 3$ numărul total de module, cu $R_{voter}(t)$ si $R_{reconf}(t)$ fiabilitătile voterului si circuitelor de comparare si reconfigurare. Se presupune că orice defect al voterului sau al subsistemului de comparare si reconfigurare produce căderea sistemului. În practică, nu toate defectele din aceste circuite sunt fatale: fiabilitatea va fi, de aceea, mai mare.



Dacă se urmăreste o evaluare mai precisă pentru $R_{hibrid}(t)$ se cere o analiză detaliată a voterului si a circuitelor de comparare si de reconfigurare precum si a modurilor în care acestea pot cădea.
Se mai vorbeste în literatură de redundanta modulară *sift-out* (cernută, cu purjare). Ca si în sistemele NMR toate cele *N* module sunt active. Un voter lucrează pe iesirile tuturor modulelor (încă) operationale. În afara voterului, un circuit de comparare si de comutare compară iesirea fiecărui modul cu iesirile celorlalte module care sunt (încă) operationale. Un modul a cărei iesire diferă de celelalte iesiri este imediat deconectat. Este un sistem întrucâtva mai simplu decât cel cu redundantă hibridă. Un astfel de sistem poate să nu fie făcut prea agresiv în procesul de purjare (cernere – *sifting-out*) deoarece imensa majoritate a defectelor pot fi tranzitorii si pot dispărea într-un timp rezonabil. Eliminarea unui modul se face numai dacă el produce sustinut iesiri eronate, incorecte pe o perioadă de timp potrivit aleasă.

Arhitecturi triplex-duplex

În această tratare a tolerantei la defecte, procesoarele sunt legate împreună pentru a forma duplexuri. Apoi din aceste duplexuri ca "module" se face un triplex. Când procesoarele dintr-un duplex sunt în dezacord, ambele sunt scoase din sistem. Aranjarea triplex-duplex permite o identificare mai simplă a procesoarelor cu probleme. Mai departe, triplexul poate continua să functioneze chiar dacă numai un duplex a rămas functional, deoarece aranjamentul duplex permite detectarea erorilor, a functionării defectuoase.

TOLERANTA LA DEFECTE SOFTWARE

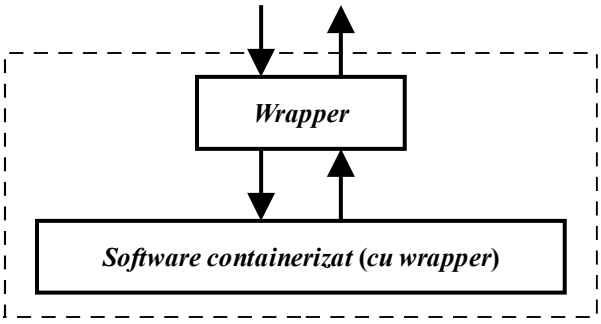
Cauze ale erorilor software

Proiectarea si scrierea software-ului sunt operatii nu lipsite de dificultate. Erorile software sunt datorate atât unor cauze care apartin de esenta realizării aplicatiilor cât si unor cauze accidentale.
Dificultătile de esentă sunt generate de:

- Întelegerea unei aplicatii complexe si a conditiilor de operare.
- Structura care poate cuprinde un număr de stări extrem de mare cu reguli de tranzitie de la o stare la alta foarte complexe.
- Modificările frecvente de software: sunt adăugate noi aspecte pentru a-l adapta la cerintele schimbătoare ale aplicatiilor.
- Platformele hardware si sistemul de operare care se pot schimba în timp si software-ul trebuie ajustat în consecintă.
- Utilizarea ocazională a software-ului pentru a pune de acord măcar partial (paper over) incompatibilitătile între componentele în interactiune ale unui sistem.

Dificultătile accidentale au ca sursă erorile umane.
Consideratiile privind costurile nu pot fi ocolite. De cele mai multe ori se recurge la software COTS (Commercial Off-The-Shelf) care nu este proiectat pentru aplicatii de mare fiabilitate.
Software-ul contine aproape inevitabil defecte. De accea se face tot posibilul ca rata de defectare să scadă sau, pentru a rezolva problema defectelor de software, se folosesc tehnici de tolerare.
O metodă este verificarea formală a corectitudinii software-ului, dar ea nu este practică pentru module mari de software. Testele de acceptare sunt de asemenea utilizate în asa-numitele wrappers (ambalaje) si în blocurile de recuperare. Aceste teste reprezintă mecanisme importante în toleranta la defecte.
Exemplu: dacă un termometru indică –40ºC în miezul unei zile de vară, termometrul respectiv este suspect de functionare proastă.
Verificările temporizate, un alt instrument utilizat în toleranata la defecte, constau în a pune un timer *watchdog* la timpul asteptat (expected) de executie. Dacă timerul este depăsit, se presupune o posibilă disfunctie de hardware sau de software. Verificările temporizate se pot folosi în paralel cu alte genuri de teste de acceptare.
Testele de acceptare se pot realiza în moduri diferite.

O posibilitate constă în verificarea iesirii: testul de acceptare este sugerat uneori în mod natural prin sortare, prin luarea rădăcinii pătrate, prin factorizarea unui număr mare sau prin solutia unor ecuatii.
O altă cale o constituie verificările statistice: de exemplu, multiplicarea matricilor de întregi $n \times n$, $C = A \cdot B$. Tratarea naivă ar consuma un timp $O(n^3)$. Metoda lui Strassen generează la întâmplare un vector de întregi cu n elemente, R. Se calculează $M_1 = A \cdot (B \cdot R)$ si $M_2 = C \cdot R$. Dacă $M_1 \neq M_2$ atunci s-a produs o eroare; dacă $M_1 = M_2$ atunci corectitudinea este înalt probabilă. Operatia se poate repeta prin generarea unui alt vector; complexitatea este $O(mn^2)$, cu m numărul de verificări.
De asemenea, o posibilitate utilizată frecvent este verificarea de domeniu (range). Se stabilesc limite acceptabile pentru iesire. Dacă iesirea este în afara limitelor: eroare. Limitele sunt fie presupuse, fie functii simple de intrări (probabilitatea de testare gresită software trebuie să fie mică). Exemplu: un satelit de telemăsurare care preia imagini termice ale pământului; se fixează limite pentru domeniul de temperaturi. Mai departe, se utilizează corelatiile spatiale. Diferentele excesive între temeperaturile unor zone adiacente sunt suspecte.
Este necesară echilibrarea între *sensibilitate* si *specificitate*. Sensibilitatea este probabilitatea conditionată ca testul să esueze când sunt date iesiri eronate. Specificitatea este probabilitatea conditionată ca eroarea să fie adevărată, reală când testul de acceptare semnalează o eroare.
Limite ale domeniului mai înguste fac să crească sensibilitatea dar fac să crească si rata de alarme false ceea ce înseamnă o descrestere a specificitătii.
O solutie o reprezintă containerele (wrappers), care sunt interfete menite a ameliora robustetea modulelor de software. Aceste containere sunt utilizate si pentru kernelul sistemului de operare, si pentru ceea ce se numeste *middleware*, si pentru software-ul de aplicatie.



Intrările sunt interceptate de wrapper. Wrapper-ul le transmite mai departe sau semnalează o exceptie. Iesirile sunt filtrate de wrapper în mod similar.

Exemplu: utilizarea de software COTS[3] pentru aplicatii de mare fiabilitate; componentele COTS sunt containerizate (wrapped) pentru a reduce rata de aparitie a disfunctiilor software. Împachetarea aceasta previne intrări care (1) sunt în afara domenului specificat sau (2) sunt cunoscute ca producătoare de căderi. Similar, iesirile sunt trecute printr-o testare de aceptare.
Exemplul 1: Rezolvarea depăsirilor de buffer. Limbajul C nu execută verificări de domeniu pentru masive (arrays), ceea ce poate produce pagube accidentale sau malitioase. Scrierea unui secvente mari într-un buffer mic face ca bufferul să dea pe-afară, iar zone de memorie din afara bufferului să fie modificate prin suprascriere. Pagubele accidentale se materializează ca defecte de memorare. Pagubele malitioase se manifestă prin suprascrierea unor portiuni din zonele *stack* si *heap* ale programului, o tehnică bine cunoscută de hackeri. Atacul cu distrugerea stivei (stack-smashing): un proces cu privilegii în rădăcină depune adresa lui de retur în stivă. Programul rău-voitor (malitios) suprascrie această adresă de revenire si controlul este redirectat la o locatie de memorie unde hackerul a depus codul său atacator. Codul atacator are acum privilegii în rădăcină si poate distruge sistemul.
În cazul unui container (wrapper) pentru protectia la depăsire de buffer, toate apelurile *malloc* ale unui program containerizat sunt interceptate de container. Containerul retine si urmăreste pozitia de început al memoriei alocate si dimensiunea. Scrierile sunt interceptate pentru a verifica potrivirea pe spatiul alocat, în limitele acestuia. Dacă apar nepotriviri, containerul nu permite scrierea si semnalează o eroare de depăsire a spatiului alocat.
Exemplul 2: Verificarea corectitudinii gestionarului (scheduler) de taskuri. Containerul în jurul gestionarului (scheduler) de taskuri este un sistem tolerant la defecte în timp real. Astfel de gestionări nu utilizează programări *round-robin*[4]; poate utiliza EDF (Earlier Deadline First); execută taskul cel mai timpuriu între taskurile gata de rulare. Poate fi supus unor reguli de prioritate (taskuri dintr-o anumită parte a executiei nu pot fi întrerupte (*preemptable*)).
Un container poate verifica dacă algoritmul de planificare/gestionare este executat corect, adică dacă s-a retinut/selectat taskul cu cel mai apropiat termen de executie si că orice task care soseste cu un termen mai timpuriu primează/prevalează asupra taskului în executie (admitând că ultimul este *preemptable*).
Containerul cere informatii despre taskurile care sunt gata de rulare si termenele lor de încheiere si totodată si dacă sunt curent în executie si sunt *preemptable*.

[3] **Commercial off-the-shelf** (**COTS**) este un termen pentru software si hardware (dar si pentru alte produse din tehnologia informatiei) care sunt gata făcute si sunt larg accesibile publicului prin vânzare, închirere etc.
[4] **Round-robin** este unul din cei mai simpli algoritmi de planificare/gestionare pentru un sistem de operare. Algoritmul atribuie ordonat fiecărui proces un interval de timp egal, fără vreo prioritate. Planificarea round-robin este simplă si usor de implementat si evită situatiile în care un proces nu are acces niciodată la resursele necesare. Planificarea/gestionarea *round-robin* poate fi aplicată si în alte împrejurări cum ar fi gestionarea pachetelor în retelele de calculatoare.

Exemplul 3: Utilizarea de software cu erori (bugs) cunoscute. Se cunoaste din testări sau din informatii provenite de la utilizatori că software-ul esuează pentru un anumit set de intrări, *S*. Containerul interceptează intrările acelui software si verifică dacă intrările sunt în *S*. Dacă nu sunt în *S*, sunt înaintate modulului de software pentru executarea calculelor. Dacă sunt în *S*, returnează sistemului o exceptie potrivită. Suplimentar, uneori redirectează intrarea spre un anumit cod alternativ, scris special pentru a manipula această întrare, acest gen de intrări.
Exemplul 4: Utilizarea unui container pentru a verifica iesirea corectă. Include un test de acceptare pentru filtrarea iesirii. Dacă iesirea trece testul, ea este transmisă în afară; dacă nu, este ridicată o exceptie si sistemul trebuie să trateze cum se cuvine iesirea suspectă.
Printre factorii care favorizează succesul în operatia de containerizare se numără:
(a) Calitatea testelor de acceptare care este dependentă de aplicatie si are impact direct asupra abilitătii containerului de a opri iesirile eronate.
(b) Disponibilitatea informatiilor necesare de la componenta containerizată: dacă acea componentă este gen "black box" (se observă numai răspunsul la o intrarea dată), containerul va fi întrucâtva mărginit; exemplu: un container de gestionare este imposibil de realizat fără informatii despre starea taskurilor care asteaptă să fie rulate.
(c) Extinderea pe care o are testarea software-ului containerizat: testarea extinsă identifică intrări pentru care software-ul esuează.

Reîntinerirea sofware-ului

Exemplu: re-bootarea unui PC. Pe măsură ce procesul evoluează în executie el achizitionează memorie si face inchideri de fisiere (file-locks) fără a le elibera adecvat. De asemenea, spatiul de memorie tinde să devină din ce în ce mai fragmentat. Procesul poate deveni susceptibil la defecte si se poate opri din executie. Pentru a preveni aceste evenimente, se opreste procesul cu premeditare, se curată starea sa internă de partea parazitară si apoi se reporneste.
Reîntinerirea poate fi din timp în timp (periodică) sau predictivă. Perioada de reîntinerire trebuie să echilibreze beneficiile si costurile.

Model analitic pentru reîntinerirea periodică

$\overline{N}(t)$ – numărul de disfunctii asteptat statistic (expected) în intervalul [0, t].
K_f – costul căderii.
K_r – costul fiecărei reîntineriri.
P – perioada reîntineririi.
Prin adunarea costurilor de reîntinerire si costurilor disfunctiei, costul general asteptat pe o perioadă între două reîntineriri este

$$C_{re}(P) = \overline{N}(P)\, K_f + K_r$$

si rata costului reîntineririi este

$$C_{rata} = C_{re}(P)/P = [\,\overline{N}(P)\, K_f + K_r]/P$$

Exemple pe modelul analitic:

(1) – rata de defectare constantă în timp – $\overline{N}(P) = \lambda P$

Rata costului este $C_{rata} = \lambda K_f + K_r/P$; optimă este valoarea $P^* = \infty$ – nu este indicată reîntinerirea.

Reîntinerirea este profilaxia unei rate de defectare crescătoare cu vârsta – aici nu-i vorba de nici un fel de îmbătrânire.

(2) $\overline{N}(P) = \lambda P^2$

Rata costului este $C_{rata} = \lambda K_f P + K_r/P$; optimă este valoarea

$$P^* = \sqrt{K_r /(\lambda\, K_f)}\ .$$

(3) $\overline{N}(P) = \lambda P^n$, $n > 1$

Rata costului este $C_{rata} = \lambda K_f P^{n-1} + K_r/P$; valoarea P optimă este

$$P^* = [K_r /(n - 1)\lambda\, K_f\,]^{1/n}\ .$$

Concluzii: P^* creste odată cu raportul K_r/K_f; rata de crestere se micsorează pe măsură ce creste n; cresterea lui λ are ca efect descrestere a lui P^*.

Perioada de reîntinerire optimă. Estimarea parametrilor K_r, K_f si $N(t)$ se poate face experimental executând simulări ale software-lui. Sistemul poate fi făcut adaptiv: valori initiale prin default si ajustarea valorilor pe măsură ce sunt recoltate date statistice noi.

Reîntinerirea predictivă se face prin monitorizarea caracteristicilor sistemului – volumul de memorie alocată, numărul de fisiere mentinute blocate (locked) etc. Acestea sunt premise pe baza cărora se poate prezice momentul când sistemul va cădea.

Exemplu: un proces consumă memorie la o rată anumită, iar sistemul estimează când el va termina memoria; reîntinerirea trebuie să aibă loc tocmai înaintea clacării prezise.

Software-ul care implementează reîntinerirea predictivă trebuie să aibă acces la suficientă informatie asupra stării procesului pentru a face astfel de predictii.

Dacă software-ul predictiv este parte a sistemului de operare, astfel de informatii se colectează usor. Dacă acesta este un pachet care rulează peste sistemul de operare fără privilegii speciale, trebuie constrâns să utilizeze interfetele oferite de sistemul de operare.

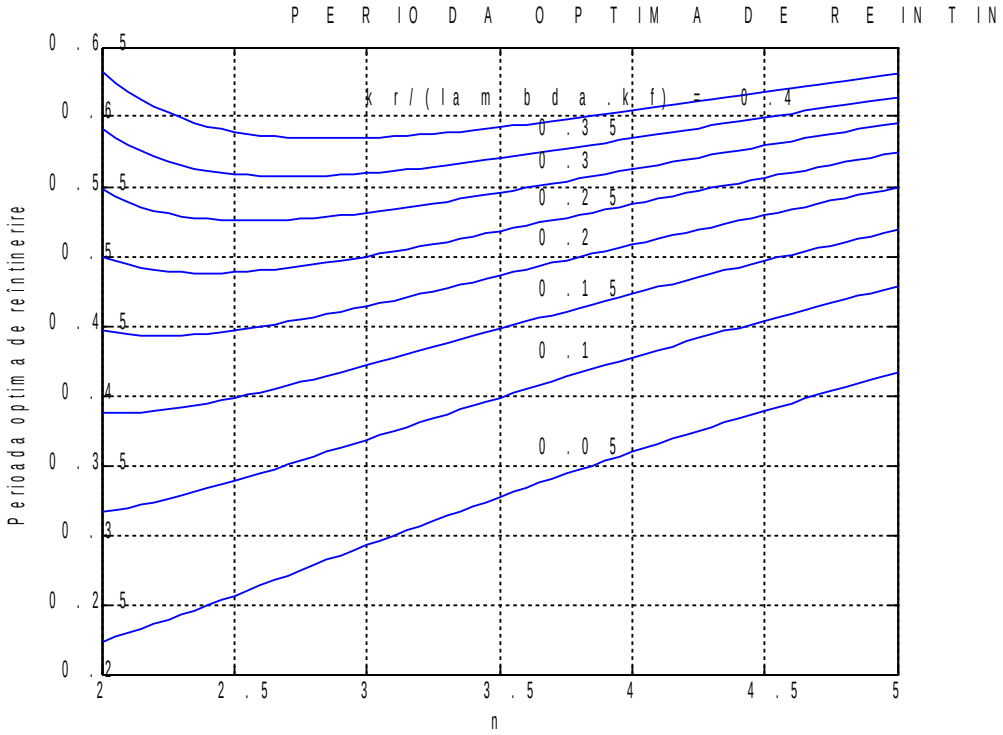
Exemplu: Sistemul de operare Unix oferă următoarele utilităti de colectare a informatiilor de stare:

iostat – extrage procentajul de utilizare a CPU la nivelurile utilizator si sistem, precum si un raport asupra utilizării fiecărui dispozitiv I/O.

netstat – indică conexiunile în retea, tabelele de rutare si un tabel al tuturor interfetelor de retea.

nfsstat – furnizează informatii despre statisticile de kernel ale serverului de fisiere din retea.

vmstat – furnizează informatii despre utilizarea procesorului, despre activitatea cu memoria si paginarea[5], despre trape[6] si despre actiunile I/O.



Predicitia momentului căderii prin metoda celor mai mici pătrate

Cu informatia despre stare colectată, urmează a identifica tendintele si a prezice momentul căderii.
Exemplu: urmărind alocarea memoriei pentru un proces, sunt colectate valorile $\mu(t_0)$, $\mu(t_1)$, …, $\mu(t_p)$, cu $\mu(t_i)$ memoria alocată la momentul t_i.
Se poate face o regresie polinomială

[5] Dacă un program face referire la o locatie de memorie care se află într-o pagină de memorie virtuală inaccesibilă, calculatorul generează o eroare de pagină. Hardware-ul trece controlul la sistemul de operare într-un loc unde poate încărca din memoria auxiliară pagina cerută (de pildă un fisier paginat de pe disc) si pune un flag pe *on* pentru a spune că pagina este accesibilă. Hardware-ul va lua apoi locatia de start a paginei, va adăuga offset-ul la bitii de ordin inferior din registrul de adrese si va accesa locatia de memore dorită.

[6] În calcul si în sistemele de operare, o trapă este un gen de întrerupere sincronă provocată de conditii exceptionale (de pildă împărtirea cu zero, aceesul la memorie nepermis) într-un proces utilizator. Uzual, o trapă produce o comutare la modul kernel în care sistemul de operare execută o anumită actiune înainte de a returna controlul procesului care l-a generat. În unele utilizări, termenul de trapă se referă specific la o intrerupere intentionată a initia o comutare contextuală la un program monitor sau la un program depanator (debugger). Într-un SNMP (Simple Network Management Protocol), o trapă este un tip de PDU (Protocol Data Unit) utilizat pentru a raporta o alertă sau alt eveniment asincron relativ la un subsistem gestionat.

$$f(t) = m_n t^n + m_{n-1} t^{n-1} + \ldots + m_1 t + c$$

cu coeficienti care să minimizeze

$$\sum_{i=0}^{p} [\mu(t_i) - f(t_i)]^2$$

Regresia cea mai simplă este desigur cea liniară: $f(t) = mt + c$.
Polinomul poate fi utilizat pentru extrapolări (prudente) în viitor si prognoze asupra momentului când procesul va ajunge în criză de memorie.

Cele mai mici pătrate ponderate

În regresia standard, fiecare punct $\mu(t_i)$ are pondere egală în stabilirea coeficientilor de regresie. O variantă mai bună este aceea cu ponderi diferite: se selectează w_0, w_1, …, w_p si apoi se calculează coeficientii din $f(t)$ pentru a minimiza

$$\sum_{i=0}^{p} w_i [\mu(t_i) - f(t_i)]^2$$

Ponderile permit diferentierea efectelor unor anumite puncte. De exemplu, ponderile $w_0 < w_1 < \ldots < w_p$ fac ca datele mai recente să influenteze mai mult coeficientii de regresie decât datele mai vechi.
Regresia (potrivirea de curbe) este vulnerabilă la impactul câtorva, putine puncte care sunt neuzual de mari sau neuzual de mici. Prin acest efect relatia poate rezulta distorsionată. Sunt accesibile tehnici de a face potrivirea mai robustă prin reducerea impactului unor astfel de puncte.

Tratamentul combinat

Reîntinerirea predictivă lucrează cu un timer resetat la momentul reîntineriii. Dacă timerul expiră, reîntinerirea se face indiferent de momentul prezis pentru producerea următoarei căderi.
Nivelul reîntineriii poate fi cel al aplicatiei sau cel al nodului. Depinde de locul unde s-au degradat resursele sau sunt pe cale de epuizare. Reîntinerirea la nivelul aplicatiei constă în suspendarea unei aplicatii individuale, efectuarea unei curătiri a stării sale (prin colectarea de *garbage*, re-initializarea structurilor de date etc.) si apoi repornirea acelei aplicatii. Reîntinerirea la nivel de nod se face prin re-bootarea nodului si afectează toate aplicatiile în executie pe acel nod.

Programarea în *N* versiuni

N echipe de programatori independente dezvoltă software cu aceleasi date initiale, cele *N* versiuni se execută în paralel, se execută un vot asupra iesirilor. Dacă programele sunt dezvoltate independent este foarte putin probabil ca ele să clacheze la aceleasi intrări.

Ipoteze de lucru: căderile sunt independente statistic; probabilitatea de a claca a unei versiuni particulare este p.
Probabilitatea de a avea exact m căderi din N versiuni este

$$P(N, m, p) = C_N^m p^m (1 - p)^{N-m}$$

Probabilitatea căderii sistemului (pentru N impar) este $\sum_{m=(N+1)/2}^{N} P(N, m, p)$

Problema *comparării consistente*. Programarea în N versiuni nu este simplu de implementat. Chiar dacă toate versiunile sunt corecte, atingerea unui consens este dificilă.
Exemplu: $V_1, \ldots, V_N$ – N versiuni scrise independent pentru calculul mărimii x si compararea ei cu c.
x_i – valoarea lui x calculată cu versiunea V_i.
B_i – variabilă booleană: $B_i \equiv (x > c)$.
Compararea cu c se spune că este *consistentă* dacă $B_i = B_j$ pentru orice pereche (i, j), cu $i, j = 1, \ldots, N$.
Cerinta de consistentă.
Exemplu: se calculează o functie de presiune si temperatură $f(p, t)$; dacă $f(p, t) \leq c$ se întreprinde actiunea A_1, dacă $f(p, t) > c$ se întreprinde actiunea A_2. Fiecare versiune produce tipul de actiune de întreprins; ideal, dacă toate versiunile sunt consistente, atunci ele indică aceeasi actiune.
Versiunile sunt scrise independent, se utilizează algoritmi diferiti de evaluare a functiei $f(p, t)$, valorile vor fi întrucâtva diferite.
$c = 1{,}0000$; $N = 3$.
Toate cele trei versiuni operează corect si valorile generate sunt 0,9999, 0,9998 si 1,0001. $B_1 = B_2 =$ false – actiunea A_1, $B_3 =$ true – actiunea A_2.
Apare o inconsistentă, cu toate că versiunile sunt toate corecte.
În problema consistentei se formulează următoarea
Teoremă: Cu exceptia algoritmului trivial care aplică orice intrare pe acelasi număr (aplicatia identică), nu există vreun algoritm care să garanteze că doi întregi de n biti care diferă cu mai putin decât 2^k vor fi aplicati pe aceeasi iesire de m biti, cu $m + k \leq n$.
Demonstratie: Se face prin reducere la absurd. Se presupune că există un astfel de algoritm (care nu aplică orice intrare pe acelasi număr). Se alege cazul $k = 1$. Numerele 0 si 1 diferă prin mai putin de 2^k astfel că ambele vor fi aplicate pe acelasi număr, să spunem L. Similar, 1 si 2 vor fi aplicate în L. La fel 3, 4, … vor fi aplicate de acest algoritm în L. Acest fapt contrazice ipoteza initială.
Un rezultat similar se mentine si pentru numere reale care diferă putin unul de altul (exercitiu).
Problema comparării consistente are nuantele expuse imediat.
Dacă versiunile nu concordă, ele pot fi cu defecte sau fără. Versiunile-esec multiple pot produce iesiri gresite identice datorită defectelor corelate. Votul sistemului va alege în astfel de cazuri o iesire gresită. Problema se poate ocoli prin obligarea versiunilor să decidă asupra unei valori de consens a unei variabile. Înainte de a verifica dacă $x > c$, versiunile cad de acord asupra unei

valori x de utilizat adăugând cerinta de a specifica ordinea comparatiilor pentru comparări multiple. Aceasta poate reduce diversitatea de versiuni, crescând potentialul pentru esecuri corelate.
Acestă schemă poate să degradeze performanta: versiunile care termină evaluarea mai devreme trebuie să astepte.
Semnale de încredere. Fiecare versiune calculează $|x - c|$; dacă rezultatul este sub un δ pozitiv precizat, versiunea anuntă o joasă încredere în iesirea sa. Voterul dă ponderi reduse versiunilor de încredere scăzută.
Problemă: dacă o versiune functională are $|x - c| < \delta$, sunt sanse mari ca acelasi lucru să se întâmple si pentru alte versiuni, ale căror iesiri vor fi devalorizate de voter. Frecventa aparitiei acestei probleme si durata ei depind de natura aplicatiei.
De exemplu, în aplicatii unde calculul depinde numai de intrările cele mai recente si nu de valori trecute, problema consensului poate apărea mai rar si poate dispărea repede.

Dependenta versiunilor

Esecurile corelate între versiuni pot provoca cresterea probabilitătii generale de cădere cu ordine de mărime.
Exemplu: $N = 3$, poate tolera până la o versiune gresită pentru orice intrare; fie $p = 0{,}0001$ – o iesire incorectă la fiecare 10.000 de executii.
Dacă versiunile sunt stochastic independente, probabilitatea de a esua a sistemului de trei versiuni este $p^3 + 3p^2(1 - p) = 3.10^{-8}$.
Se presupune acum o dependentă stochastică si existenta unui defect comun pentru două versiuni, defect care se manifestă o dată la un milion de executii provocând căderea sistemului.
Probabilitatea eseculului pentru sistemul de trei versiuni creste la peste 10^{-6}, mai mult de 30 de ori fată de situatia când defectele nu sunt corelate.
Modelul cu versiuni corelate. Spatiul intrărilor este divizat în regiuni: probabilităti diferite de preluare de intrări din regiunea care produce esecul unei versiuni.
Exemplu: algoritmul poate avea instabilitate numerică într-o parte a spatiului intrărilor; rata disfunctiilor în acea parte este mai mare decât cea medie.
Ipoteză de lucru: versiunile sunt independente stochastic în fiecare subspatiu dat S_i; Pr[atât V_1 cât si V_2 să esueze|intrare din S_i] =

$$= \Pr[V_1 \text{ esuează}|\text{intrare din } S_i].\Pr[V_2 \text{ esuează}|\text{intrare din } S_i]$$

Probabilitatea neconditionată ca o versiune să esueze

$$\Pr[V_1 \text{ esuează}] = \Pr[V_1 \text{ esuează}|\text{intrare din } S_i].\Pr[\text{intrare din } S_i]$$

Probabilitatea neconditionată ca ambele să esueze

$$\Pr[\text{atât } V_1 \text{ cât si } V_2 \text{ să esueze}] =$$
$$= \Pr[\text{atât } V_1 \text{ cât si } V_2 \text{ să esueze}|\text{intrare din } S_i].\Pr[\text{intrare din } S_i] \neq$$
$$\neq \Pr[V_1 \text{ esuează}].\Pr[V_2 \text{ esuează}]$$

Exemplul numeric 1:

Se identifică două subspatii de intrare S_1 si S_2, fiecare cu probabilitatea 0,5.
Probabilitătile conditionate de esec sunt:

Versiunea	S_1	S_2
V_1	0,01	0,001
V_2	0,02	0,003

Probabilitătile neconditionate de esec sunt:

$$\Pr[V_1 \text{ esuează}] = 0{,}01.0{,}5 + 0{,}001.0{,}5 = 0{,}0055$$
$$\Pr[V_2 \text{ esuează}] = 0{,}02.0{,}5 + 0{,}003.0{,}5 = 0{,}0115$$

Dacă versiunile ar fi independente, probabilitatea ca ambele să esueze pentru aceeasi intrare ar fi

$$0{,}0055.0{,}0115 = 6{,}33.10^{-5}.$$

Probabilitatea reală de esec conjugat este întrucâtva mai mare

$$0{,}01.0{,}02.0{,}5 + 0{,}001.0{,}003.0{,}5 = 1{,}02.10^{-4}$$

Cele două versiuni sunt corelate pozitiv: ambele sunt mai predispuse la esec în S_1 decât în S_2.
Exemplul numeric 2.
Probabilitătile conditionate de esec sunt:

Versiunea	S_1	S_2
V_1	0,01	0,001
V_2	0,003	0,02

Probabilitătile neconditionate de esec sunt la fel ca în exemplul de mai sus.
Probabilitatea de esec combinat este

$$0{,}01.0{,}003.0{,}5 + 0{,}001.0{,}02.0{,}5 = 2{,}5.10^{-5}$$

Mai putin decât probabilitatea combinată evaluată mai sus sau decât produsul probabilităților individuale.
Tendintele de cădere sunt corelate, în acest caz, negativ: V_1 este mai bun în S_1 decât în S_2, invers pentru V_2; V_1 si V_2 îsi maschează deficientele reciproc. Ideal ar fi ca versiunile multiple să fie corelate negativ. În practică corelatiile sunt pozitive deoarece versiunile rezolvă aceeasi problemă.

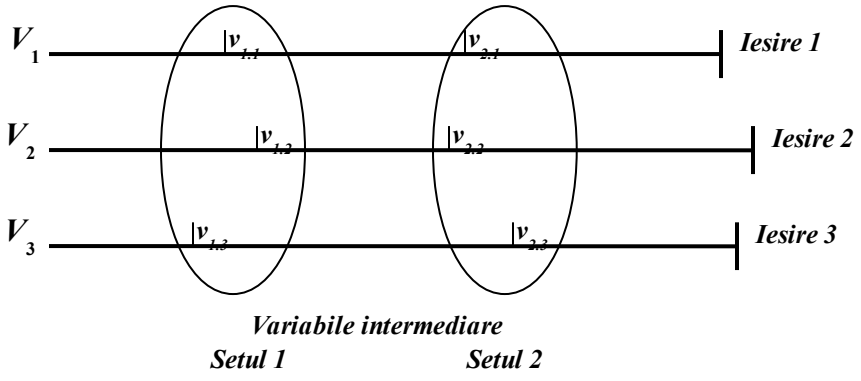
Cauzele corelării versiunilor

Specificatii comune: comune sunt si erorile în specificatii (elementele de pornire) care se propagă la software.
Dificultatea intrinsecă a problemei: algoritmii pot fi mai dificil de implementat pentru unele intrări, producând declansarea defectelor pentru aceleasi intrări.
Algoritmi comuni: algoritmul însusi poate contine instabilităti în anumite regiuni ale spatiului intrărilor. De aceea, versiuni diferite pot avea instabilităti în aceeasi regiune.

Factori culturali: în interpretarea unor specificatii ambigue, programatorii fac greseli similare.
Platforme software si hardware comune: dacă se utilizează acelasi hardware, acelasi sistem de operare si acelasi compilator, defectele lor pot declansa un esec corelat.
Din aceste motive si din altele, se recomandă obtinerea independentei versiunilor prin diversitatea incidentală.
Se creează obligatia ca aceia care dezvoltă module software diferite să lucreze independent unul de altul. Se impune interdictia ca echipele care lucrează la module diferite să comunice între ele. Întrebările privind ambiguitătile din specificatii sau oricare altă problemă trebuie adresate unei autorităti centrale care are obligatia de a face corectiile si actualizările necesare pentru toate echipele. Inspectia sofware-ului trebuie să fie foarte atent coordonată astfel încât inspectorii unei versiuni să nu scape direct sau indirect informatii despre o altă versiune.
Obtinerea independentei versiunilor prin metode de diversitate fortată.
Specificatiile initiale se fac deliberat diverse. Versiunile sunt si ele create deliberat cu capabilităti diferite. Se folosesc limbaje de programare diferite. Până si instrumentele de dezvoltare si compilatoarele se diversifică. La fel sistemele de operare si hardware se aleg diferite.
Specificatii initiale diverse. Multe disfunctii de software sunt datorate cerintelor formulate prin specificatie. Diversitatea poate începe chiar din faza de specificare: specificatiile pot fi exprimate într-un formalism diferit. Erorile de specificare nu vor coincide pe versiuni, fiecare specificare va declansa un profil de defecte de implementare diferit.
Versiuni cu capabilităti diferite. Un exemplu poate fi o versiune rudimentară care furnizează iesiri mai putin precise dar încă acceptabile. O a doua versiune mai simplă, mai putin predispusă la defecte si bazată pe un algoritm mai robust. Dacă cele două versiuni nu concordă, o a treia versiune poate ajuta la determinarea versiunii corecte din cele două de până atunci. Dacă a treia versiune este foarte simplă, pentru a dovedi corectitudinea pot fi folosite metodele formale.
Limbaje de programare diferite. Limbajele de programare afectează calitatea software-ului.
Exemple:

- Assemblerele sunt mai predispuse la erori decât limbajele de nivel mai înalt.
- Natura erorilor este diferită: în programele scrise în C se depăseste usor memoria alocată, este imposibil ca memoria să fie gestionată strict în vreun limbaj.
- Utilizarea fără defect a pointerilor în Fortran este imposibilă deoarece Fortranul nu are posibilitatea de a defini pointeri.
- Fată de C sau Fortran, Lisp-ul este un limbaj mai natural pentru o parte din algoritmii de inteligentă artificială.

Limbajele diferite de programare au biblioteci diferite si compilatoare diferite; programele vor avea esecuri necorelate (sau, chiar mai bine, corelate negativ).
Alegerea limbajului de programare. Problemă ridicată de programatori: este necesar ca toate versiunile să folosească cel mai bun limbaj pentru rezolvarea unei probleme sau unele versiuni ar trebui scrise în alte limbaje mai putin potrivite?
Dacă se utilizează acelasi limbaj, rata de defectare individuală este mai redusă dar esecurile posibile sunt corelate pozitiv. Dacă limbajele utilizate sunt diferite, unele rate de defectare particulare pot fi mai mari dar rata de defectare generală a sistemului de *N* versiuni poate fi mai redusă dacă esecurile sunt mai slab corelate. Echilibrul (*tradeoff*) este dificil de rezolvat; nu există vreun model analitic, de aceea este necesară o experimentare extinsă.
Instrumente de dezvoltare si compilatoare diverse. Diversitatea de instrumente si de compilatoare face posibilă o "diversitate de notatii" care poate să reducă corelatiile pozitive între disfunctii. Utilizarea de instrumente de dezvoltare diverse si de compilatoare diverse (posibil purtătoare de defecte ele însesi) pentru versiunile diferite pot produce fiabilităti mai mari.
Sisteme de operare si hardware diverse. Iesirile sistemului depind de interactiunea între software-ul de aplicatie si platforma sa, sistemul de operare si procesorul. Atât procesorul cât si sistemul de operare au specificitatea lor prin defectele (*bugs*) pe care le contin. Este asadar o idee bună de a completa proiectarea software diversă cu diversitatea hardware si de sistem de operare. Urmează natural rularea fiecărei versiuni pe procesoare si sisteme de operare diferite.
Testarea spate-n-spate. Compararea valorilor unor variabile intermediare sau a iesirilor pentru aceeasi intrare reprezintă o cale de a identifica erorile, erori care uzual nu coincid.



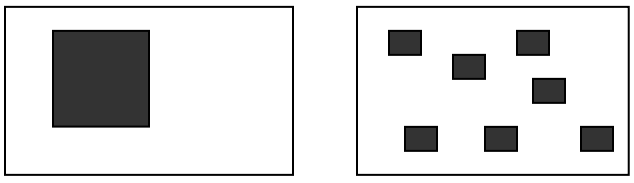
Variabilele intermediare furnizează o observabilitate crescută asupra comportării programelor. Dar definirea variabilelor intermediare de observat constrânge pe cei care dezvoltă programe să producă aceste variabile, ceea ce reduce diversitatea în programare si independenta.

Versiune unică sau versiuni multiple (*N*)? Este o întrebare care nu are un răspuns simplu. O presupunere de ordin practic: dezvoltând *N* versiuni, cheltuielile cresc de *N* ori fată de cele pentru o singură versiune. Unele părti ale procesului de dezvoltare de software pot fi comune. De pildă, dacă toate versiunile utilizează aceeasi specificatie, trebuie dezvoltat si pus la punct numai un set de specificatii.
Gestiunea unui proiect pe *N* versiuni impune overhead aditional.
Costurile pot fi reduse prin identificarea celor mai critice portiuni ale codului si prin dezvoltarea de versiuni diferite numai pentru acestea.
Fiind dat un timp total si un buget, sunt două alegeri posibile: (a) dezvoltarea unei singure versiuni utilizând întregul buget sau (b) dezvoltarea a *N* versiuni.
Nu există un model satisfăcător pentru a alege între cele două posibilităti.
Rezultate de natură experimentală. Sunt putine studii experimentale asupra eficientei programării în versiuni multiple. Rezultatele publicate sunt generate numai de activitatea derulată în universităti. Iată un studiu făcut la universităti din Virginia si din California:

- 27 de studenti au scris un cod pentru aplicatii anti-rachetă.
- Unii nu aveau nici o experientă în industria software, iar altii aveau peste zece ani de experientă.
- Toate versiunile au fost scrise în Pascal.
- Prin metode de verificare a ipotezelor statistice standard s-au identificat 93 de defecte corelate: dacă versiunile ar fi fost independente statistic, numărul de defecte corelate asteptat ar fi fost nu mai mare de 5.
- Nu s-a identificat nici o corelatie între calitatea programelor produse si experienta programatorilor.

Diversitatea de date

Spatiul intrărilor unui program poate fi divizat în regiuni asociate cu defecte si regiuni fără vreo asociere cu defectele programului. Programul esuează dacă si numai dacă i se aplică o intrare din regiunea asociată cu defecte.
Se consideră (nu tocmai nerealist) un spatiu de intrare cu două dimensiuni (v.figura). În ambele cazuri, regiunile cu defecte ocupă o treime din spatiu.



Dacă se perturbă întrucâtva intrarea, valorile noi de intrare pot cădea în regiunea fără vreo asociere cu defectele. Această procedură este numită *diversitate de date*.

Un modul de software care utilizează teste de acceptare reface calculul cu intrări perturbate si reverifică iesirile. Este vorba aici de o redundantă masivă: se aplică seturi usor diferite de intrări pe diferite versiuni de software si se face o votare pe rezultate.
Perturbarea despre care se vorbeste mai sus poate fi explicită si/sau implicită. Explicită este atunci când se adaugă unui subset de intrări selectat un termen-deviatie redus. Implicit este când se colectează intrări pentru program astfel încât ne putem astepta ca ele să fie usor diferite.
Exemplul 1: Controlul prin software al proceselor industriale; intrările sunt presiunea si temperatura unui boiler. La fiecare secundă se măsoară (p_i, t_i) si se servesc ca intrări regulatorului. Măsurătorile la momentul i nu sunt mult diferite de cele de la momentul anterior $i - 1$. Perturbarea implicită poate să consiste în utilizarea perechii (p_{i-1}, t_{i-1}) în loc de (p_i, t_i). Dacă perechea (p_i, t_i) este în regiunea cu defecte, perechea (p_{i-1}, t_{i-1}) poate că nu este de-acolo.
Perturbarea explicită, intrări reordonate.
Exemplul 2: Se adună numerele în virgulă mobilă a, b, c: mai întâi se face $a + b$, apoi se adună c. Valori: a = 1.1E+20, b = 5, c = –1.1E+20. În functie de precizia utilizată, $a + b$ poate fi 1.1E+20 si rezultatul $a + b + c = 0$.
Se schimbă ordinea intrărilor la a, c, b: atunci $a + c = 0$ si $a + c + b = 5$.
În exemplul 2 re-exprimarea este exactă, iesirea poate fi utilizată asa cum este (dacă trece testul de acceptare sau votul).
În exemplul 1 re-exprimarea este inexactă, este probabil ca $f(p_i, t_i) \neq f(p_{i-1}, t_{i-1})$. Se utilizează iesirea brută ca o alternativă degradată dar acceptabilă sau se încearcă corectarea înainte de utilizare, de pildă prin dezvoltarea Taylor

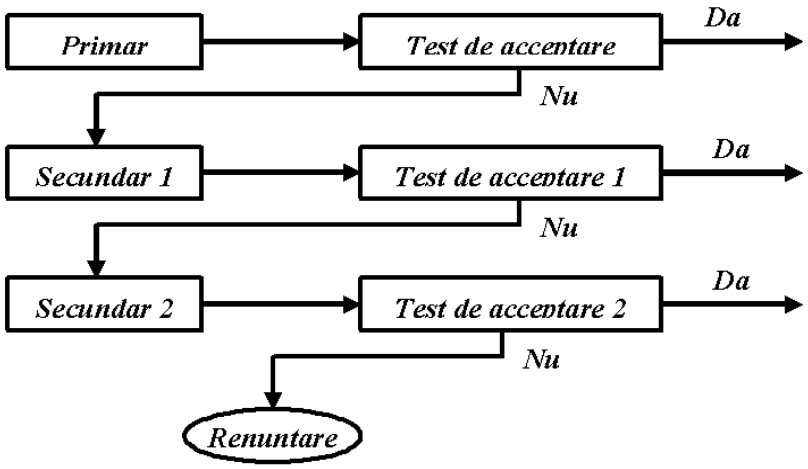
$$f(t) = f(t_0) + \sum_{n=1}^{\infty} \frac{(t - t_0)^n f^{(n)}(t_0)}{n!}$$

SIHFT (Software Implemented Hardware Fault Tolerance)

Variatia din diversitatea de date poate fi utilizată pentru rezolvarea căderilor permanente hardware. Fiecare intrare este multiplicată cu o constantă k si programul este transformat pentru a corecta această multiplicare.
Exemplu: constructul **`if (x=y) then`** …
Pentru valorile x = 001, y = 000, dacă egalitatea este verificată de hardware care are bitul x_0 agătat-la-0, se va calcula eronat $x = y$.
SIHFT cu k = 2 produce x' = 010, y' = 000 si defectul nu actionează, iar circuitul determină corect că $x \neq y$.
Transformarea programului în sensul de a compensa multiplicarea cu k nu este dificilă. Singura problemă este găsirea unei valori potrivite pentru k:
(1) Ne asigurăm că este posibil a găsi tipuri de date potrivite astfel ca depăsirile aritmetice în jos si în sus să nu apară.
(2) Se alege k astfel încât să fie capabil a masca o mare cotă din defectele de hardware; se fac studii experimentale prin injectarea de defecte.

Realizarea unui bloc de recuperare

N versiuni, o executie: dacă aceasta esuează, executia e comutată la un backup.
Exemplu: o versiune primară si două versiuni secundare. Versiunea primară se execută, iesirea se dirijează la testul de acceptare. Dacă iesirea nu este acceptată, starea sistemului este restabilită (*rolled back*) si este pornită versiunea secundară 1, si tot asa în continuare. Dacă toate vesiunile esuează, calculul însusi esuează. Succesul tratării prin bloc de recuperare este conditionat de independenta disfunctiilor pe versiuni diferite si de calitatea testului de acceptare.



Model analitic pentru tratarea cu bloc de recuperare. Ipoteză de lucru: versiunile diferite esuează independent.
Notatii:
E – evenimentul "iesirea unei versiuni este eronată"
T – evenimentul "testul esuează" (testul nu detectează un defect, o eroare)
f – probabilitatea eseculului unei versiuni: $f = \Pr(E)$
s – sensibilitatea testului: $s = \Pr(T|E)$
σ – specificitatea testului: $\sigma = \Pr(E|T)$
N – numărul de versiuni software.
Calculul probabilitătii unei disfunctii urmează schema de mai jos.
Se calculează probabilitatea complementară, aceea a succesului. Pentru ca schema să aibă succes, trebuie să aibă succes într-una din etape i, $1 \le i \le N$. La etapele 1, …, $i - 1$ atât software-ul cât si testul esuează, la etpa i versiunea software este corectă si iesirea trece testarea

$$\Pr(\text{succes în etapa } i) = [\Pr(E \cap T)]^{i-1} \Pr(\overline{E} \cap \overline{T})$$
$$C = \Pr(T|E)\Pr(E) = s.f$$
$$\Pr(\overline{E} \cap \overline{T}) = \Pr(\overline{E}) - \Pr(\overline{E} \cap T)$$
$$\Pr(\overline{E}) = 1 - \Pr(E) = 1 - f$$
$$\Pr(\overline{E} \cap T) = \Pr(\overline{E} \mid T)\Pr(T)$$

$$\Pr(\overline{E} \mid T) = 1 - \sigma$$

$$\Pr(E|T) = \Pr(E \cap T)/\Pr(T)$$

în consecintă

$$\Pr(T) = \frac{\Pr(F \cap T)}{\Pr(E \mid T)} = \frac{s.f}{\sigma}$$

$$\Pr(\overline{E} \cap T) = (1 - \sigma)\frac{s.f}{\sigma}$$

si

$$\Pr(\overline{E} \cap \overline{T}) = (1 - f) - (1 - \sigma)\frac{s.f}{\sigma}$$

Prin substituire si prin însumare după *i* se obtine

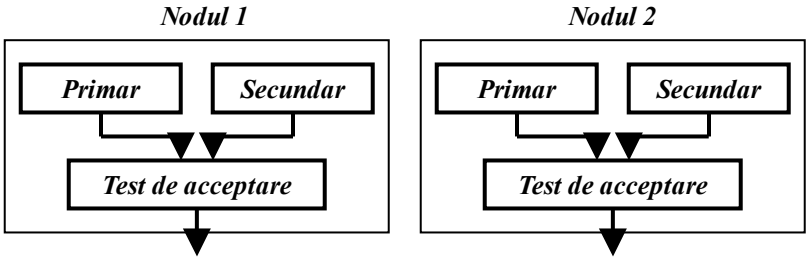
$$\Pr(\text{schema să aibă succes}) = \sum_{i=1}^{n} (s.f)^{i-1}\left[(1 - f) - (1 - \sigma)\frac{s.f}{\sigma}\right] =$$

$$= \left[(1 - f) - (1 - \sigma)\frac{s.f}{\sigma}\right]\frac{1 - (s.f)^n}{1 - s.f}$$

$$\Pr(\text{esec}) = 1 - \Pr(\text{schema să aibă succes})$$

Importanță critică are specificitatea σ a testului de acceptare care trebuie să fie înaltă.

Blocuri de recuperare distribuite. Două noduri poartă cópii identice ale versiunilor primară si secundară.



Nodul 1 execută versiunea primară. În paralel, nodul 2 execută versiunea secundară. Dacă nodul 1 esuează la testul de acceptare, este folosită iesirea nodului 2 (dat fiind că aceea a trecut testul). Iesirea nodului 2 poate fi utilizată si dacă nodul 1 nu reuseste să producă o iesire într-un timp pre-specificat.

Odată ce versiunea primară esuează, rolurile versiunilor primară si secundară sunt inversate. Nodul 2 continuă să execute copia secundară care acum este tratată ca primară. Executia în nodul 1 a versiunii primare este utilizată ca un backup. Aceasta continuă până când executia din nodul 2 este semnalată ca eronată; atunci sistemul comută invers, utilizând executia din nodul 2 în rol de backup.

Revenirea (*roollback*) nu este necesară. Se economiseste timp, timp util pentru sistemele în timp real cu termene foarte ferme pentru taskuri.

Schema poate fi extinsă la *N* versiuni (una primară plus *N* – 1 versiuni secundare cu rulare în paralel pe *N* procesoare).
Tratarea exceptiilor. O exceptie indică faptul că ceva s-a întâmplat în timpul executiei, ceva care merită atentie. Controlul este transferat la un *handler* de exceptie, o rutină care actionează potrivit.
Exemplu: când se execută *y = a*b*, dacă apare o depăsire, rezultatul este incorect; se generează un semnal de exceptie.
Tratarea efectivă a exceptiilor poate aduce o crestere semnificativă a tolerantei la defecte a sistemului. În multe programe, peste jumătate din liniile unui cod sunt dedicate tratării unor exceptii.
Exceptiile pot fi utilizate pentru a rezolva:

- Disfunctii de domeniu sau de limite
- Evenimente extra-ordinare (nu neapărat disfunctii) care necesită o atentie aparte
- Esecuri de temporizare.

Disfunctii de domeniu si de limite. O disfunctie de domeniu se produce când sunt utilizate intrări ilegale.
Exemplu: dacă *x*, *y* sunt numere reale si se încearcă $x = \sqrt{y}$ cu $y = -1$, se produce o disfunctie de domeniu.
O disfuntie de limite apare când programul produce o iesire sau execută o operatie care este percepută ca eronată într-un anume sens.
Exemple:

- Atingerea unui sfârsit de fisier (EOF) în timpul citirii unui fisier de date
- Producerea unui rezultat care violează un test de acceptare
- Încercarea de a tipări o linie care este prea lungă
- Generarea unei depăsiri aritmetice de tip *overflow* sau *underflow.*

Evenimente extra-ordinare (*neobisnuite*). Exceptiile pot fi utilizate pentru a asigura tratarea specială a unor evenimente rare dar perfect normale.
Exemplu: citirea unui ultim articol al unei liste dintr-un fisier poate declansa o exceptie pentru a notifica pe cel care a invocat citirea că acela a fost ultimul articol.
Disfunctii de temporizare. În aplicatiile în timp reral, taskurile au termene fixe (*deadlines*). Dacă termenele sunt depăsite, se poate declansa o exceptie. Handlerul de exceptie decide ce trebuie făcut în replică (de pildă comutarea la o rutină backup).
Cerinte relativ la handlerele de exceptii.
(1) Trebuie să fie usor de programat si de utilizat. Să fie modulare si separabile de restul software-lui. Să nu fie amestecate cu alte linii de cod într-o rutină; ar putea fi greu de înteles, de depanat, de modificat.
(2) Tratarea exceptiei nu trebuie să impună un overhead substantial asupra functionării normale a sistemului. Exceptiile să fie invocate numai în împrejurări într-adevăr exceptionale. Tratarea exceptiei să nu fie o povară în cazul uzual fără conditii de exceptie.

(3) Tratarea exceptiilor nu trebuie să compromită starea sistemului, să nu-l lase inconsistent.

Definitii pentru fiabilitatea software

Disfunctie: abaterea comportării software-ului de la cerintele utilizatorului.
Fiabilitate: probabilitatea de operare software fără disfunctii într-o ambiantă definită, pentru o perioadă de timp specificată.
Software-ul nu se deteriorează în timp ca hardware-ul; el rămâne constant, acelasi dacă nu sunt operate schimbări.
Dacă în perioada testării sunt detectate si eliminate defecte, fiabilitatea software creste în timp.
Notatii:
N – numărul de defecte existente la începutul testării (poate fi o variabilă aleatoare)
$M(t)$ – numărul de defecte detectate si eliminate până la momentul t
$N - M(t)$ – numărul de defecte rămase la timpul t.

Modele pentru fiabilitatea software

Se încearcă predictia ratei de defectare viitoare a software-ului ca functie de numărul de defecte eliminate sau de numărul de defecte rămase la momentul t.
Spre deosebire de modelele fiabilitătii hardware, acestea sunt netestate în sensul cel mai larg al notiunii.
Modelele disponibile dau adesea rezultate contradictorii.
Când testarea porneste, defectele mai "usoare" sunt capturate curând. Defectele rămase sunt din ce în ce mai dificil de capturat; fie sunt mai greu de întâlnit, fie efectul lor este mascat de calcule următoare manifestării lui.
Rata la care defectele nedescoperite încă produc disfunctii scade pe măsură ce testarea avansează. Rata de defectare este descrisă fie ca o functie descrescătoare de $M(t)$, fie ca o functie crescătoare de $N - M(t)$.
Modelul Jelinski-Moranda.
Rata de defectare $\lambda(t)$ este proportională cu numărul de defecte rămase în software: $\lambda(t) = c[N - M(t)]$. Când un defect este detectat, el este imediat eliminat; timpul până la disfunctia următoare este distribuit exponential cu parametrul $\lambda(t)$.
Problemă: nu toate defectele se manifestă egal; unele apar mai frecvent, altele mai rar si sunt dificil de (sur)prins.
Modelul Littlewood-Verall.
Timpul scurs între disfunctii este distribuit exponential cu parametrul $\lambda(i)$ în care $i = N - M(t)$ este numărul de defecte rămase. Parametrul $\lambda(i)$ este o variabilă aleatoare distribuită cu o densitate de probabilitate de tipul *gamma*

$$f(x) = \frac{[\psi(i)]^{\alpha} x^{\alpha-1} e^{-\psi(i)x}}{\Gamma(\alpha)}$$

Functia *gamma* este într-un fel o generalizare a functiei factorial

$$\Gamma(t) = \int_0^{\infty} e^{-x} x^{t-1} dx \quad \text{si} \quad \Gamma(t+1) = t\Gamma(t)$$

Functia $\psi(i)$ si parametrul α se determină prin observarea experimentală a software-ului.

Modelul Musa-Okumoto.

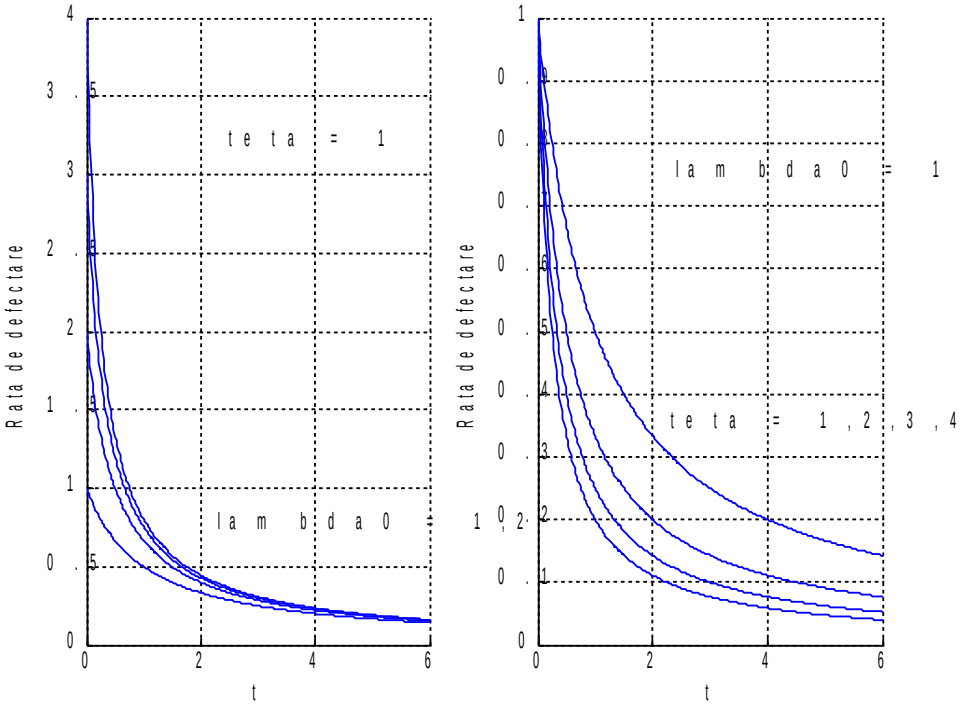
Este un model cu timpul de executie Poisson logaritmic. Este un model mai larg utilizat pentru fiabilitatea software.

Rata de defectare după timpul *t* este

$$\lambda(t) = \lambda(0) \exp[-\theta \mu(t)]$$

cu θ o constantă si cu $\mu(t)$ valoarea medie a lui $M(t)$, numărul de disfunctii manifestate si înlăturate înainte de momentul *t*.

$$\frac{d\mu(t)}{dt} = \lambda(t)$$



$$\lambda_0 = \frac{d\mu(t)}{dt} e^{\theta \mu(t)}$$

$$\frac{de^{\theta \mu(t)}}{dt} = \theta \frac{d\mu(t)}{dt} e^{\theta \mu(t)} = \lambda_0 \theta$$

$$e^{\theta \mu(t)} = \lambda_0 \theta t + 1$$

$$\mu(t) = \theta^{-1} \ln(\lambda_0 \theta t + 1)$$

$$\lambda(t) = \frac{\lambda_0}{\lambda_0 \theta t + 1}$$

Aceasta este expresia unei foarte lente scăderi a ratei de defectare, ceea ce duce la o cantitate apreciabilă de observatii experimentale.

Selectarea modelului si estimarea de parametri. Întrebări legitime:

(1) Care model este cel mai potrivit?

(2) Cum se estimează parametrii modelului ales?

Datele experimentale nu sunt foarte cuprinzătoare pentru a ghida utilizatorul. Se studiază rata de defectare ca functie de testare si se ghiceste mai curând modelul pe care-l urmează; apoi se estimează parametrii lui.

Se utilizează tehnicile de estimare statistică, de pildă metoda verosimilității maxime sau metoda celor mai mici pătrate.

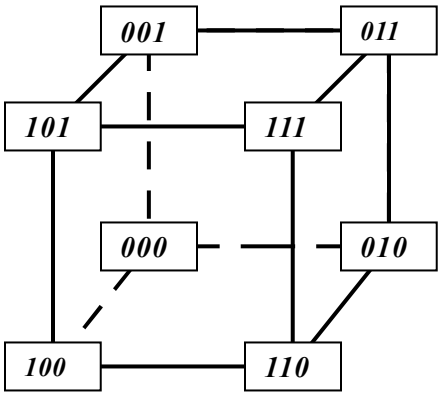
REDUNDANTE INFORMATIONALE

Redundanta informațională (codarea)

Un cuvânt cu *d* biti este codat într-un cuvânt de cod de *c* biti, $c > d$. Rezultă un număr de 2^c cuvinte binare, dar nu toate cele 2^c combinatii de biti sunt cuvinte de cod valide.
Înainte de a putea utiliza informatia este necesar a se extrage din cuvântul de cod cuvântul originar. Altfel zis cei *c* biti trebuie decodati.
Dacă cei *c* biti primiti nu constituie un cuvânt de cod valid, înseamnă că s-a detectat o eroare. Pentru anumiti algoritmi de codare, unele tipuri de erori pot fi nu numai detectate ci pot fi chiar corectate la destinatie, la utilizator.
Parametrii cheie în operatia de codare/decodare îl reprezintă numărul de biti eronati care pot fi detectati ca eronati si numărul de biti eronati care pot fi corectati. În general, aceste două numere nu coincid.
Într-o privire foarte generală, bitii suplimentari necesari pentru a crea posibilitatea de detectare/corectare a erorilor fac să crească timpul consumat, si nu numai prin operatiile de codare si de decodare ci si prin durata transmiterii unui număr mai mare de biti.

Distanta Hamming

Distanta Hamming între două cuvinte (de cod) este dată de numărul de pozitii binare prin care cele două cuvinte diferă. Evident, distanta Hamming se poate defini numai pentru cuvinte de lungimi identice.



În figura alăturată, distanta Hamming dintre două cuvinte care sunt separate (legate) de o muchie este 1. Două cuvinte separate prin două muchii (ca cel mai scurt parcurs de la unul la altul) sunt distantate Hamming cu 2. Cea mai mare distantă Hamming, 3, este între cuvintele situate în extremitătile unei diagonale a cubului.
Odată definită distanta Hamming, se poate defini distanta unui cod. *Distanta unui cod* este distanta Hamming minimă între două cuvinte valide distincte ale codului.
De exemplu, codul de patru cuvinte {001, 010, 100, 111} are o distantă egală cu 2. Acest cod poate detecta o eroare pe un singur bit (inversarea unui bit).
Un alt exemplu: codul cu două cuvinte {000, 111} are o distantă de 3. Acest cod poate detecta orice eroare de un bit sau de doi biti (unul sau doi biti inversati). Dacă o eroare dublă în acelasi cuvânt de cod este suficient de improbabilă, un alt mod de a spune "are o probabilitate foarte mică fată de eroarea unică, singulară", codul poate corecta un bit eronat.
Exemplele de mai sus sugerează vag, dar sugerează, conditiile necesare pentru a putea face o detectie sau o corectie a cuvâtului de cod receptionat. Regulile respective sunt:

- Pentru a detecta într-un cuvânt de cod până la t biti eronati, distanta acelui cod trebuie să fie de cel putin $t + 1$
- Pentru a corecta până la t biti eronati într-un cuvânt de cod, distanta codului trebuie să fie de cel putin $2t + 1$.

Aceste conditii sunt demonstrate riguros în multe surse, în particular în lucrarea Gh.M.Panaitescu *Transmiterea si codarea informatiei. Note de curs* (2007), p.81 (http://ac.upg-ploiesti.ro/gpanaitescu/tci.pdf).

Codare si redundantă

Codul {000, 111} poate fi utilizat pentru a coda date de un bit: 0 poate fi codat ca 000 si 1 ca 111. Codul acesta lucrează, într-un fel, identic cu TMR (sistemul Triple Modular Redundant) si anume prin "vot" majoritar. Sub acest aspect, multe tehnici redundante hardware pot fi considerate scheme de codare. Si duplexul poate fi considerat un cod ale cărui cuvinte constau în două cuvinte identice alăturate. Pentru datele de un bit, cuvintele de cod sunt 00 si 11.

Separabilitatea unui cod

Un cod este *separabil* dacă are câmpurile pentru bitii de date si bitii de codare în secvente separate. La decodare bitii de date sunt recuperati prin simpla ignorare a bitilor adăugati pentru codare. Asadar, bitii de codare pot fi ei însisi procesati separat pentru a verifica corectitudinea acelor de date.
Un cod *neseparabil* are bitii de date si bitii de codare integrati si extragerea datelor din cuvântul de cod necesită o procesare specifică.

Coduri prin paritate

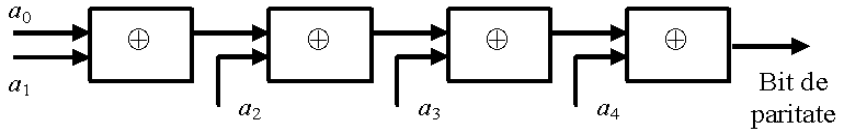
Cele mai simple coduri separabile sunt codurile cu paritate (constantă). Un cuvât al unui cod cu paritate (constantă) contine *n* biti informationali si un bit suplimentar care mentine paritatea. Pentru un cod cu paritate cu sot (sau fără sot), bitul suplimentar este fixat astfel ca numărul total de unităti binare din cuvântul de (*n* + 1) biti să fie totdeauna par (sau impar). Fractia suplimentară inevitabilă a unui cod cu paritate constantă este 1/*n*.

Proprietăti ale codurilor cu paritate constantă

Un cod cu paritate constantă are o distantă Hamming de 2 si este capabil a detecta erorile de un singur bit într-un cuvânt de cod. Dacă un bit se schimbă din 0 în 1 sau invers, paritatea se schimbă si eroarea poate fi detectată prin verificarea parităţii. Această paritate simplă nu poate însă corecta bitul eronat.

Circuite de codare si de decodare pentru coduri cu paritate constantă

Codorul este un sumator de biti modulo-2. Acesta generează un 0 dacă numărul de unităti binare din cuvântul-date este cu sot, un 1 dacă numărul de unităti binare a cuvântului-date este fără sot. Iesirea sumatorului este semnalul de paritate.



Codor

Decodorul re-generează bitul de paritate din bitii care constituie partea de date în cuvântul receptionat într-o schemă identică cu cea a codorului si îl compară cu bitul de paritate primit într-un circuit nici-exclusiv. Dacă există potrivire între cei doi biti de paritate, cel evaluat si cel primit, iesirea portii nici-exclusiv este un 1, ceea ce indică absenta erorii. Lipsa coincidentei produce în acelasi punct al schemei un 0 ceea ce indică o eroare.
Erorile pe doi biti nu pot fi detectate de un cod cu paritate constantă: paritatea nu este modificată prin inversarea a doi biti.
Toate erorile de trei biti într-un cuvânt, desigur o situatie mult mai rară, sunt detectate fără a putea discerne dacă este vorba de una sau trei erori în bitii cuvântului de cod.
După cum s-a amintit mai devreme, paritatea utilizată poate fi cu sot, fie fără sot. Când se alege una, când se alege cealaltă? Uzual este preferată paritatea cu sot dar decizia depinde uneori de tipul mai probabil de erori pe toti bitii cuvântului.

Pentru paritatea pară, bitul de paritate pentru toti bitii nuli este 0 si o cădere de tipul *toti-bitii*-0 trece neobservată, deoarece apare deghizată ca un cuvânt de cod valid. Prin alegerea tipului de paritate impar, eroarea de tipul *toti-bitii*-0 devine detectabilă.
Dacă eroarea de tipul *toti-bitii*-1 este mai probabilă, atunci, după caz, dacă numărul total de biti (n + 1) este par trebuie selectată paritatea impară, iar dacă (n + 1) este un număr impar trebuie retinută si utilizată paritatea pară.
Un sistem cu paritate constantă este si acela în care bitii de paritate sunt atribuiti separat unui byte sau oricărui alt grup de biti de lungime stabilită. Proportia de biti suplimentari se modifică de la 1/n la m/n (m este numărul de bytes-i sau de alte grupe egal dimensionate). Cu acest sistem se pot detecta până la m erori dacă erorile apar în grupe de biti (eventual bytes-i) diferite. Dacă penele de tipurile *toti-bitii*-0 si *toti-bitii*-1 sunt în linii mari la fel de posibile, atunci se selectează alternativ paritatea pară pentru un byte, paritatea impară pentru alt byte.

Codul cu paritate pentru bytes-i întretesuti

Cuvântul de cod de lungime n = 64, de pildă, arată astfel: $a_{63}a_{62}\ldots a_1a_0$. Sunt cuprinsi aici opt biti de paritate: primul bit al fiecărui octet, cel mai semnificativ, este bit de paritate. Pentru exemplul cu n = 64, bitii de paritate sunt a_{63}, a_{55}, a_{47}, a_{39}, a_{31}, a_{23}, a_{15}, a_7. Primul bit de paritate a_{63} este cel mai semnificativ bit în lantul de 8 bytes-i considerat. Ceilalti 7 biti de paritate se atribuie astfel ca grupele corespondente de biti să fie intretesute. Schema aceasta este benefică atunci când scurtcircuitarea unor biti adiacenti este un mod de defectare comun (de exemplu pe un bus). Dacă tipul de paritate (pară – impară) este alternat între grupe, erorile unidirectionale (*toti-bitii*-0 sau *toti-bitii*-1) vor fi si ele detectate.

Coduri cu paritate capabile a corecta erori

În schema cea mai simplă, informatia este organizată într-o matrice bidimensională

0	0	0	1	1	1	1
1	0	1	0	1	1	0
1	1	0	0	0	0	0
0	0	0	1	1	1	1
1	1	1	1	1	1	0
1	0	0	1	0	0	0

Bitii de la finele unei linii sunt biti de paritate pe acea linie, bitii de la baza unei coloane sunt de paritate pe acea coloană. O eroare pe un bit situată undeva, oriunde în matricea utilizată, afectează corectitudinea parităţii atât pe o linie cât

si pe o coloană. Identificarea liniei si coloanei care contin bitul eronat nu este o problemă, corectarea lui este imediată.
Acesta este un exemplu de *paritate suprapusă*: fiecare bit este acoperit de mai mult de un bit de paritate.

Modelul general pentru paritatea suprapusă

Scopul urmărit este identificarea vreunui bit eronat singular, presupus unicul bit eronat din cuvânt. În structura cuvântului de cod intră d biti de informatie si r biti de paritate, asadar cuvântul are un total de $d + r$ biti. Admitând că erorile sunt pe un bit si pe numai unul singur, sunt $d + r$ stări cu eroare si o stare fără eroare, ceea ce dă un total de $d + r + 1$ stări distincte ale cuvântului receptionat. În consecinta acestor observatii, pentru a distinge între stările enumerate sunt necesare $d + r + 1$ "semnături" de paritate distincte, tot atâtea configuratii de biti. Aceste r verificări de paritate generează 2^r semnături. Asadar, r este cel mai mic întreg care satisface inegalitatea $2^r \geq d + r + 1$.
Apare întrebarea firească: cum pot fi atribuiti bitii de paritate? Iată un exemplu: Fie $d = 4$ numărul bitilor de date si $r = 3$ numărul (minim) de biti de paritate. Într-adevăr, sunt $d + r + 1 = 8$ stări în care poate fi cuvântul de cod, sunt $2^3 = 8$ adaosuri distincte la secventa de biti de date.
Tabelul alăturat arată o atribuire posibilă, nu singura, de biti de paritate stărilor cuvântului receptionat. În cuvântul ($a_3a_2a_1a_0p_2p_1p_0$) pozitiile 0, 1 si 2 sunt consacrate bitilor de verificare a parității, restul sunt biti de informatie.

Starea	Biti de paritate
Fără eroare	000
Eroare pe bitul 0 (p_0)	001
Eroare pe bitul 1 (p_1)	010
Eroare pe bitul 2 (p_2)	100
Eroare pe bitul 3 (a_0)	011
Eroare pe bitul 4 (a_1)	101
Eroare pe bitul 5 (a_2)	110
Eroare pe bitul 6 (a_3)	111

Aceiasi parametri d, r îi are si un cod consacrat si foarte utilizat: codul Hamming (7,4) corector de o singură eroare (SEC – Single Error Corrector). După cum se poate observa, bitii de paritate în afară de faptul că numerotează în baza 2 cele opt variante posibile ale cuvântului receptionat, 000 pentru cuvântul fără eroare, cu o eroare pe unul din cei 7 biti pozitionată diferit de la caz la caz, acesti biti de paritate mai sunt si într-o anumită relatie cu bitii de informatie. Astfel, p_0 acoperă pozitiile 0, 3, 4, 6, adică $p_0 = a_0 \oplus a_1 \oplus a_3$, p_1 acoperă pozitiile 1, 3, 5, 6, adică $p_1 = a_0 \oplus a_2 \oplus a_3$ si p_2 acoperă pozitiile 2, 4, 5, 6, prin $p_2 = a_1 \oplus a_2 \oplus a_3$.

Un exemplu: pentru $a_3a_2a_1a_0 = 1100$ completarea cu biti de paritate este $p_2p_1p_0 = 001$. Dacă 1100001 devine 1000001, cu bitul eronat subliniat, se recalculează secventa de paritate din cuvântul eronat: $p_2p_1p_0 = 111$. Diferenta dintre seceventele de control al parității se obtine printr-o operatie "sau exclusiv" bit-cu-bit, care este 110. Diferenta aceasta se numeste *sindrom* si indică bitul eronat. Bitul a_2 este eronat si informatia corectă este $a_3a_2a_1a_0 = 1100$.
Pentru codul Hamming (7,4) sindromul poate fi calculat direct, într-un pas, din bitii receptionati a_3, a_2, a_1, a_0, p_2, p_1, p_0, prin operatia matricială următoare, cu toate operatiile făcute *modulo* 2.

$$\begin{matrix} a_3 & a_2 & a_1 & a_0 & p_2 & p_1 & p_0 \end{matrix}$$
$$\begin{bmatrix} 1 & 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} a_3 \\ a_2 \\ a_1 \\ a_0 \\ p_2 \\ p_1 \\ p_0 \end{bmatrix} = \begin{bmatrix} s_2 \\ s_1 \\ s_0 \end{bmatrix}$$

Matricea care (pre)multiplică cuvântul receptionat reprezentat aici ca un vector de valori binare este asa-numita matrice de verificare. Cuvintele de cod verifică relatia de mai sus cu un sindrom cu toate componentele nule. Prezenta erorii este semnalată de un vector sindrom nenul. Identificarea bitului eronat (si corectarea lui) se face prin potrivirea sindromului pe una din coloanele matricei de verificare. Dacă sindromul este $[1\ 1\ 0]^T$, cum s-a întâmplat în exemplul dat, acesta se potriveste (numai) pe coloana a_2. Consecinta: acela este bitul eronat si trebuie corectat (prin inversare).
Într-o pre-ordonare potrivită a bitilor de informatie si a bitilor de paritate, bitul eronat poatre fi localizat scăzând o unitate din indicele sindromului: rezultatul este indicele bitului eronat. În exemplul prezentat mai sus, ordinea aceea este $a_3a_2a_1a_0p_2p_1p_0$.
În general, dacă $2^r > d + r + 1$ este necesar a alege pentru a fi combinatii sindrom $d + r + 1$ combinatii binare din cele 2^r . Combinatiile cu multi de 1 trebuie evitate deoarece mai putini de 1 în matricea de verificare a parității înseamnă practic circuite mai simple pentru operatiile de codare si de decodare.
Selectarea matricei de verificare este de oarecare importantă. De exemplu, pentru $d = 3$ si $r = 3$, numai sapte din cele opt combinatii de 3 biti sunt necesare. Sunt posibile două matrici de verificare:

$$\begin{matrix} a_2 & a_1 & a_0 & p_2 & p_1 & p_0 \end{matrix}$$
$$\begin{bmatrix} 1 & 0 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 & 0 & 1 \end{bmatrix}$$
si
$$\begin{matrix} a_2 & a_1 & a_0 & p_2 & p_1 & p_0 \end{matrix}$$
$$\begin{bmatrix} 0 & 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 & 0 & 1 \end{bmatrix}$$

Prima are prea multi de 1 (111 pe prima coloană), a doua este mai potrivită. Pentru prima circuitele de codare necesită câte o poartă XOR (sau exclusiv)

pentru p_1 si p_2, dar două pentru bitul p_0. Pentru a doua matrice circuitele de codare necesită câte o poartă XOR pentru fiecare bit de paritate.

Îmbunătătirea detectiei este un obiectiv frecvent în toleranta la defecte. Codul descris mai devreme poate corecta un bit eronat si poate detecta erori pe doi biti. De exemplu, 1100001 poate deveni 1<u>01</u>0001, cu bitii subliniati, a_2 si a_1, eronati. Sindromul este în cazul respectiv 011. Acest sindrom indică gresit ca necesară o corectie a bitului a_0. O cale de a îmbunătăti capacitatea de detectie constă în a suplimenta cu un bit secventa de verificare. Acesta este un bit de verificare a parității pentru toti ceilalti biti, de informatie sau de paritate. Codul acesta este cunoscut ca un cod Hamming (8,4) de corectare a unui bit eronat/de detectare a doi biti eronati (SEC/DED – Single Error Correction/Double Error Detection).

Generarea sindromului pentru codul Hamming (8,4) este similară celei pentru codul Hamming (7,4): prin multiplicarea cuvâtului cu o matrice de verificare. Rezultatul este diferit: un vector sindrom cu 4 componente.

$$\begin{array}{c} a_3\ a_2\ a_1\ a_0\ p_3\ p_2\ p_1\ p_0 \\ \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 & 0 & 1 \end{bmatrix} \end{array} \begin{bmatrix} a_3 \\ a_2 \\ a_1 \\ a_0 \\ p_3 \\ p_2 \\ p_1 \\ p_0 \end{bmatrix} = \begin{bmatrix} s_3 \\ s_2 \\ s_1 \\ s_0 \end{bmatrix}$$

Bitul p_3 este, cum s-a spus, un bit de paritate pentru toti bitii de informatie si de control. O eroare de un singur bit schimbă paritatea generală si produce $s_3 = 1$. Ultimii trei biti ai sindromului indică bitul eronat, care trebuie corectat ca si mai devreme dacă $s_3 = 1$. Dacă $s_3 = 0$ si oricare alt bit al sindromului este nenul, se detectează o eroare dublă sau multiplă de ordin superior lui 2. Un exemplu:

Eroare unică – 11001001 devine 1<u>0</u>001001
Sindromul este 1110 – indică eroare pe a_2
Eroare dublă – 11001001 devine 1<u>01</u>01001
Sindromul este 0011 si indică o eroare (multiplă) care nu poate fi corectată

Iată acum un cod Hamming (8, 4) diferit. Codul Hamming (8, 4) prezentat si comentat mai devreme necesită calculul bitului de verificare suplimentar, ceea ce aduce un plus în consumul de timp la codare si la decodare. Pentru a evita această situatie, o solutie posibilă constă în atribuirea de sindromuri cu număr impar de unități binare.

$$\begin{array}{cccccccc} a_3 & a_2 & a_1 & a_0 & p_3 & p_2 & p_1 & p_0 \\ \left[\begin{array}{cccccccc} 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 \end{array}\right] \end{array}$$

O eroare dublă produce un sindrom cu un număr par de 1, ceea ce indică o eroare (multiplă) care nu poate fi corectată. În acest cadru, se utilizează numai 2^{p-1} combinatii din cele 2^p posibile. Este necesar, prin urmare, un bit de control suplimentar peste minimul necesar pentru un cod Hamming de tipul corector de o singură eroare (SEC). Numărul total de biti de control este acelasi cu cel necesar pentru codul Hamming original, capabil a corecta o eroare si de a detecta o dublă eroare (SEC/DED).

Ca structură, codurile cu *parităti suprapuse* sunt multe si variate. O comparatie între aceste coduri este interesantă. Astfel, la codurile Hamming, pe măsură ce d creste, overhead-ul parităţii r/d, raportul biti de protectie/biti de date, scade. Dar probabilitatea de a avea mai mult de un bit eronat în $d + r$ biti creste.

Fie f probabilitatea aparitiei unui bit eronat. Se presupune că erorile apar independent una de alta. Probabilitatea ca într-un câmp de $d + r$ biti succesivi să apară mai mult de un bit eronat este

$$\Phi(d, r) = 1 - (1 - f)^{d+r} - (d + r)f(1 - f)^{d+r-1} \approx 0{,}5(d + r)(d + r - 1)f^2$$

d	r	Overhead r/d	$\Psi(D, d, r)$
2	3	1,5000	0,5120E–16
4	3	0,7500	0,5376 E–16
8	4	0,5000	0,8448 E–16
16	5	0,3125	0,1344 E–15
32	6	0,1875	0,2250 E–15
64	7	0,1094	0,3976 E–15
128	8	0,0625	0,7344 E–15
256	9	0,0352	0,1399 E–14
512	10	0,0195	0,2720 E–14
1024	11	0,0107	0,5351 E–14

adică complementul la unitate al probabilităţii prezentei a cel mult o eroare, cu aproximarea valabilă pentru $f << 1$.

Dacă sunt D biti de date în total, probabilitatea de a avea mai mult de un bit eronat poate fi redusă prin partitionarea celor D biti în D/d portiuni egale, cu fiecare portiune codată separat. Este vorba aici de un compromis, un echilbru între probabilitatea erorilor nedetectabile si overhead-ul r/d. Probabilitatea de a avea o eroare necorectată în cel putin una din cele D/d portiuni este

$$\Psi(D, d, r) = 1 - [1 - \Phi(d, r)]^{D/d} \approx (D/d)\Phi(d, r)$$

cu aproximarea acceptată dacă $\Phi(d, r) << 1$.

Tabelul de mai sus cuprinde o comparatie numerică pentru cazul $D = 1024$ si $f = 10^{-11}$.

Codurile cu *sumă de control* sunt utilizate în principal pentru a detecta erori în transmisia de date prin retelele de comunicatie. Ideea de bază: se însumează blocul de date care se transmite si se transmite *si* această sumă. Receptorul însumează datele primite si compară suma obtinută cu suma de verificare trimisă odată cu datele. Dacă cele două sume de control nu coincid înseamnă că transmiterea s-a făcut cu eroare.

Sunt mai multe versiuni pentru *suma de control*.

Fie *d* (biti) lungimea cuvintelor de date. În versiunea *simplă precizie*, suma de control rezultă ca un număr modulo 2^d. În versiunea *dublă precizie*, suma de control rezultă ca un număr modulo 2^{2d}.

Deoarece în procesul de însumare sunt retinuti numai *d* biti, cei mai din dreapta, este de înteles că suma de control în simplă precizie este capabilă a cuprinde mai putine erori decât suma de control în dublă precizie. În cazul dublei precizii, sunt retinuti 2*d* biti din suma de control.

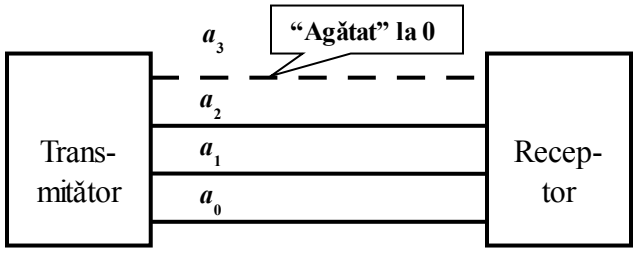
Suma de control *cu reziduu* face un transfer de la bitul al *d*-lea, cel mai semnificativ, la bitul cel mai putin semnificativ, un transfer circular, ciclic, si de aceea codul este întrucâtva mai fiabil.

Suma de control *Honeywell* concatenează cuvintele în perechi pentru calculul sumei de verificare (făcută modulo 2^{2d}) si prin aceasta codul protejează la erori repetate în aceeasi pozitie a secventei.

Imediat mai jos este dată o comparatie între cele patru versiuni ale sumelor de control:

Precizie simplă	Precizie dublă	Cu reziduu	Honeywell
0000	0000	0000	
0101	0101	0101	
1111	1111	1111	00000101
0010	0010	0010	11110010
0110	**00010110**	**0111**	**11110111**

Un exemplu este prezentat în figura alăturată.



Precizie simplă		Honeywell	
Transmis	Receptionat	Transmis	Receptionat
1000	0000		
1011	0011	10001011	00000011
0000	0000	00000100	00000100
0100	0100		
0111	**0111**	**10001111**	**00000111**

În cazul sumei de control în simplă precizie, sumele pentru secventa transmisă si pentru cea calculată se potrivesc: asadar, erori nedetectate. În cazul sumei Honeywell, sumele de control pentru secventa transmisă si pentru cea calculată nu se potrivesc si erorile sunt detectate.
Toate schemele cu sumă de control au un neajuns conceptual: atunci când fac detectia de erori nu le si localizează. Dacă sunt detectate erori, întregul bloc de date trebuie retransmis.
Codurile *Berger* sunt coduri separabile. Se numără unitătile binare din cuvânt, se exprimă rezultatul în binar, se complementează si se atasează rezultatul la bitii de informatie.
Exemplu: Codarea secventei 11101. Sunt patru de 1, în binar 100, 011 după complementare. Cuvântul de cod se obtine prin juxtapunerea rezultatului după bitii de informatie: 11101011.

d	*c*	*overhead*
4	3	0,7500
8	4	0,5000
16	5	0,3125
32	6	0,1875
64	7	0,1094
128	8	0,0625
256	9	0,0352
512	10	0,0195

Un cod *Berger* detectează toate erorile unidirectionale, cele care constau în faptul că una sau mai multe unităti binare devin zerouri si zerourile rămân zerouri (sau invers). Totusi, dacă numărul de biti care trec din 0 în 1 este egal cu cel al celor care trec din 1 în 0, eroarea devine nedetectabilă.
Se poate evalua un overhead pentru un cod Berger. Dacă bitii de informatie sunt în număr de d, sunt cel mult d de 1 si sunt necesari până la $\log_2(d + 1)$ biti pentru reprezentarea numărului de unităti binare. Asadar

$$overhead = [\log_2(d + 1)]/d$$

Cu notatia c pentru numărul de biti de verificare, se obtine tabelul de mai sus.

Coduri ciclice

Codurile ciclice au această denumire derivată din proprietatea cuvintelor de cod de a rămâne în cadrul codului dacă bitii componenti sunt permutati circular. Codurile ciclice sunt în acelasi timp coduri liniare: suma a două cuvinte de cod, bit-cu-bit modulo 2, este totdeauna un cuvânt de cod.

Aceste coduri sunt uzual neseparabile, dar există si coduri ciclice separabile. Codarea constă în multiplicarea (modulo 2) a cuvântului de date cu un număr constant. Produsul este cuvântul codat. Decodarea constă în diviziunea cu aceeasi constantă. Dacă restul nu este zero, de vină este o eroare.

Codurile ciclice sunt larg utilizate în stocarea si transmiterea datelor.

Teoria pe care se bazează codurile ciclice porneste de la un număr k de biti de informatie care trebuie codati. Cuvântul de cod este lung de n biti obtinuti prin multiplicarea celor k biti de informatie cu un număr care este ca lungime de până la $n - k + 1$ biti. Numărul multiplicator este prezentat ca un polinom, denumit si polinom generator. Unitătile si zerourile binare din multiplicatorul de $n - k + 1$ biti sunt considerate coeficienti ai unui polinom de gradul $n - k$.

Exemplu: multiplicatorul este 11001, polinomul generator este

$$G(X) = 1X^0 + 0X^1 + 0X^2 + 1X^3 + 1X^4 = 1 + X^3 + X^4$$

Un cod ciclic (n, k) este un cod ciclic care utilizează un polinom generator de gradul $n - k$ si are numărul total de biti în cuvintele de cod n. Un cod ciclic (n, k) poate detecta toate erorile singulare si toate secventele de biti adiacenti eronati mai scurte de $n - k$ biti.

Codurile ciclice sunt utile în aplicatii de genul comunicatiilor fără fir: canalele sunt uzual afectate de zgomot si de interferente care produc erori pe secvente de biti adiacenti.

Implementarea hardware execută multiplicarea (modulo 2) pe registre de deplasare si porti XOR (sau exclusiv). De exemplu, dacă polinomul generator este $1 + X^3 + X^4$ (care corespunde multiplicatorului 11001), circuitul de codare este alcătuit din două porti XOR si patru operatii de deplasare. Registrele de deplasare sunt elemente de întârziere care mentin la iesire intrarea lor pe durata unui ciclu.

În ceea ce priveste multiplicarea modulo 2, dacă multiplicatorul este alimentat serial si se adună multiplicatul deplasat (shifted), se obtine produsul

```
     1100101
       11001
------------
     1100101
    0000000
   0000000
  1100101
1100101
------------
10100011101
```

Al cincilea bit al produsului, de pildă, este suma modulo 2 a bitilor corespunzători ai multiplicatului (subliniati) deplasati spre stânga de 0 ori, de 3 ori si de 4 ori, deplasări de ordine egale cu ordinele bitilor nenuli ai mutiplicatorului. Deplasările se realizează cu elementele de întârziere. Codul ciclic nu este separabil: bitii de informatie si bitii de control nu apar separati în cuvântul de cod 10100011101.
Operatiile circuitului de codare sunt date în tabelul alăturat.

Tempo	Intrare	O_4	i_3	O_3, O_2, O_1	Iesire
1	1	0	1	000	1
2	0	1	1	100	0
3	1	0	1	110	1
4	0	1	1	111	1
5	0	0	0	111	1
6	1	0	1	011	0
7	1	1	0	101	0
8	0	1	1	010	0
9	0	0	0	101	1
10	0	0	0	010	0
11	0	0	0	001	1

Coloana i_3 contine intrarea în elementul de întârziere care produce pe O_3. Intrarea de biti informationali în cazul ilustrat este 1100101, iar iesirea codată este 10100011101.
Decodarea se face prin împărtirea la polinomul generator si urmează algoritmul prezentat imediat. Pentru cuvântul fără eroare si respectiv, pentru cuvântul cu eroare (eroarea este reprezentată diferit) se obtine:

```
10100011101 : 11001 = 1100101
11001
 11010
 11001
    11111
    11001
      11001
      11001
      00000

10100111101 : 11001 = 1100110
11001
 11011
 11001
    10111
    11001
     11100
     11001
       1011
```

asadar, o dată rest nul, altă dată rest nenul. Restul nul arată că nu a apărut nici o eroare. La împărtirea cu 11001 practicată mai sus, scăderea (aici modulo 2) cu care suntem obisnuiti la împărtirea a două numere, este identică cu adunarea.
Prezenta unei/unor erori scoate de obicei cuvântul din multimea de cuvinte ale codului. Semnalarea acestei situatii o face restul nenul: dacă cuvâtul eronat este 10100111101, restul este, cum s-a văzut, 01011, nenul.
Să vedem ce se poate întâmpla dacă trei biti sunt eronati. Sunt date ilustrativ două cazuri. Rezultatele împărtirii cu polinomul generator sunt respectiv:

```
101110101 : 11001 = 1101101
11001
 11100
 11001
   10110
   11001
    11111
    11001
      11001
      11001
      00000

10011011101 : 11001 = 1110011
11001
 10100
 11001
  11011
  11001
     10110
     11001
      11111
      11001
      00110
```

Situatia 10111010101 în loc de 10100011101 produce un rest nul, asadar erorile rămân nedetectate. Dacă cele trei erori sunt adiacente, 10011011101, restul nenul indică prezenta erorilor.
Implementarea unui circuit de împărtire tine seama de faptul că diviziunea poate fi făcută prin multiplicare în buclă feedback.
Exemplu: Fie polinomul $E(X)$ cuvântul receptionat; fie $G(X)$ polinomul generator; fie $D(X)$ cuvântul originar informational. Dacă nu există eroare, se receptionează $E(X)$ si se calculează $D(X)$ prin împărtirea $E(X)/G(X)$ si restul este polinomul nul. De exemplu

$$E(X) = D(X)G(X) = D(X)(1 + X^3 + X^4) = D(X) + D(X)(X^3 + X^4)$$

si

$$D(X) = E(X) + D(X)(X^3 + X^4)$$

deoarece în aritmetica modulo 2 scăderea coincide cu adunarea.
Circuitul divizor foloseste si el registre de deplasare si porti XOR. Apare în plus un circuit feedback pentru diviziunea $D(X) = E(X) + D(X)(X^3 + X^4)$. La pornire,

fiecare element de întârziere retine un 0. Circuitul produce mai întâi câtul de sapte biti (bitii de informatie) si apoi cei patru biti ai restului. Dacă restul este nenul, este prezentă cel putin o eroare.
Operatiile divizorului sunt prezentate în tabelul care urmează; i_3 este intrarea pentru elementul de întârziere $O_3 = i_4 \oplus O_4$.

Tempo	Intrare	i_4	O_4	i_3	O_3, O_2, O_1	Iesire
1	1	1	0	1	000	1
2	0	0	1	1	100	0
3	1	1	0	1	110	1
4	1	0	1	1	111	0
5	1	0	0	0	111	0
6	0	1	0	1	011	1
7	0	1	1	0	101	1
8	0	0	1	1	010	0
9	1	0	0	0	101	0
10	0	0	0	0	010	0
11	1	0	0	0	001	0

Dacă intrarea este 10100011101, iesire este 1100101 si restul este nul. Orice eroare singulară în secventa $E(X)$ receptionată produce un rest nenul.
Într-un cod ciclic (n, k), numărul de biti de control $n - k$ este independent de numărul k al bitilor de informatie; k poate fi crescut fără a creste si $n - k$, ceea ce lasă gradul polinomului generator neschimbat si complexitatea circuitelor de codare si de decodare rămâne la fel.
În ceea ce priveste detectarea erorilor în secventă (bursty errors), pe măsură ce k creste, abilitatea codului ciclic de a detecta erorile succesive se reduce; numai erorile în secventă de lungime până la $n - k$ sunt detectate garantat.
În multe aplicatii apare necesitatea de a face sigură detectarea tuturor erorilor succesive pe lungime de până la 16 biti. Pentru aceasta se utilizează codurile ciclice de tipul (16 + k, k). Cele mai frecvent utilizate:
CRC-16 (Cyclic Redundancy Code pe 16 biti), cu polinomul generator

$$G(X) = X^{16} + X^{15} + X^2 + 1$$

si CRC-CCITT cu polinomul generator

$$G(X) = X^{16} + X^{12} + X^5 + 1$$

Coduri aritmetice

Codurile aritmetice sunt coduri care se conservă printr-un anumit set de operatii aritmetice. Această proprietate permite detectarea erorilor care pot apărea în timpul executării unei operatii aritmetice din setul de operatii definit. O astfel de detectare concurentă a erorilor poate fi totdeauna atinsă prin duplicarea procesorului aritmetic. Această din urmă metodă este totdeauna costisitoare.

Relativ la conservarea codului, se spune că un cod este conservat printr-o operatie aritmetică * dacă pentru orice pereche de operanzi (X, Y) si pentru perechea de entităti codate corespunzătoare (X', Y') există o operatie ⊗ pentru operanzii codati astfel încât $X' \otimes Y' = (X * Y)'$. Altfel spus, operatia aritmetică executată când ⊗ este aplicat operanzilor codati X' si Y' produce acelasi rezultat ca cel obtinut prin codarea rezultatului operatiei originare * aplicat operanzilor X si Y înainte de codare. Rezultatul operatiei aritmetice este codat în acelasi mod.

Cum se face detectarea erorii? Este de asteptat ca un cod aritmetic să fie capabil a detecta toate erorile de un singur bit. O eroare pe un singur bit într-un operand sau într-un rezultat intermediar poate produce o eroare pe mai multi biti în rezultatul final. De exemplu, la adunarea a două numere în binar, dacă sectiunea i a sumatorului este defectă, toti ceilalti $n - i$ digiti de ordin mai înalt pot fi eronati.

Codurile aritmetice *neseparabile* sunt cele mai simple. Acestea se vor identifica în continuare sub denumirea de codurile AN. Sunt formate prin multiplicarea operanzilor cu o constantă A, $X' = A*X$ si operatiile * si ⊗ coincid: ambele sunt multiplicări.

De exemplu, dacă $A = 3$, fiecare operand este multiplicat cu 3. Rezultatul fiecărei operatii aritmetice este verificat dacă este un întreg multiplu de 3. Toate erorile care au magnitudinea multiplu de A nu vor putea fi detectate.

În codarea AN, constanta A nu trebuie să fie o putere a lui 2. Un A impar este cea mai bună alegere; codul va detecta orice eroare de un bit. O astfel de eroare are o magnitudine de 2^i.

Dacă $A = 3$, se obtine codul AN cel mai ieftin, care face posibilă detectarea tuturor erorilor de un bit.

Exemplu: Numărul $0110_2 = 6_{10}$. Reprezentarea în cod AN cu $A = 3$ este $010010_2 = 18_{10}$. O eroare în bitul cu pozitia 2^3 poate da rezultatul eronat $01\underline{1}010_2 = 26_{10}$. Eroarea este detectabilă deoarece 26 nu este un multiplu de 3.

Există si coduri aritmetice *separabile*. Cele mai simple sunt *codul rezidual* si *codul rezidual inversat*. În fiecare din acestea se atasează un simbol de control separat $C(X)$ pentru fiecare operand X. Pentru codul rezidual, $C(X) = X \bmod A = |X|_A$. A este numit modulul de verificare.

Pentru ambele coduri separabile $C(X) \otimes C(Y) = C(X*Y)$ cu operatiile * sau ⊗ reprezentând fie adunare, fie multiplicare

$$|X + Y|_A = ||X|_A + |Y|_A|_A$$
$$|X * Y|_A = ||X|_A * |Y|_A|_A$$

În împărtirea $X - S = Q * D$, X este dempărtitul, D este împărtitorul, Q este câtul, S este restul si verificarea este $||X|_A - |S|_A|_A = ||Q|_A * |D|_A|_A$.

Exemple: Dacă $A = 3$, $X = 7$ si $Y = 5$, resturile sunt $|X|_3 = 1$ si $|Y|_3 = 2$ si

$$|7 + 5|_3 = 0 = ||7|_3 + |5|_3|_3 = |1 + 2|_3 = 0$$
$$|7 * 5|_3 = 2 = ||7|_3 * |5|_3|_3 = |1 * 2|_3 = 2.$$

Dacă $A = 3$, $X = 7$ si $D = 5$, atunci $Q = 1$ si $S = 2$; verificarea prin rest este

$$||7|_3 - |2|_3|_3 = ||5|_3 * |1|_3|_3 = 2$$

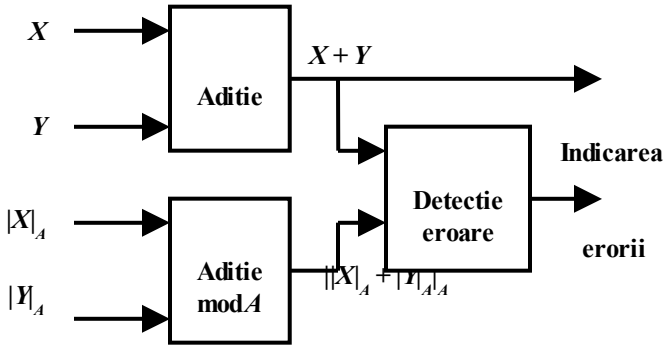
Scăderea se face prin adunarea complementului la modul (modulus):

$$|1 - 2|_3 = |1 + |3 - 2|_3|_3 = |1 + 1|_3 = 2$$

Urmează o comparatie între codul rezidual modulo A fată-n fată cu codul AN.

- Ambele au aceleasi erori nedetectabile.

Exemplu: Pentru $A = 3$ sunt nedetectabile numai erorile care modifică rezultatul cu un multiplu de 3. Erorile pe un singur bit sunt totdeauna detectate.



- Aceeasi algoritmi de verificare pentru ambele metode: se calculează restul modulo A al rezultatului.
- Aceeasi crestere în lungime a cuvântului codat, cu $|\log_2 A|$
- Diferenta majoră constă în separabilitate: Unitatea pentru $C(X)$ în codul rezidual este separabilă. Pentru codul AN codul este unitar.

Codurile AN si codurile reziduale cu $A = 3$ sunt exemplele cele mai simple dintr-o clasă de coduri aritmetice ieftine care folosesc o constantă A de forma $A = 2^a - 1$, cu exponentul a un întreg. Acestă alegere simplifică calculul restului la împărtirea cu A (în algoritmul de verificare). Calcularea restului este simplă deoarce $|z_i r^i|_{r-1} = |z_i|_{r-1}$ când $r = 2^a$.

Aceste coduri permit utilizarea sumei modulo $2^a - 1$ pentru grupele de a biti care compun numărul (fiecare grupă are o valoare $0 \le z_i \le 2^a - 1$).

Exemplu: Restul împărtirii lui $X = 1110101011$ prin $A = 7 = 2^3 - 1$.

Se face o partitie a lui X în grupe de câte 3 biti, începând cu bitul cel mai putin semnificativ. Se obtine $X = (z_3, z_2, z_1, z_0) = (11\ 110\ 101\ 011)$.

Se adună aceste grupe modulo 7; se elimină numerele 7 si se adună *end-around-carry* (rest circular) dacă este necesar.

Ponderea transferului (carry-out) este 8 si $|8|_7 = 1$: se adună end-around-carry.

Restul modulo 7 al lui X este 3. Restul corect al lui $X = 1963_{10}$ divizat cu 7 este $|z_i r^i|_{r-1} = |z_i|_{r-1}$; $r = 2^a$

```
    11    z3
+  110    z2
1  001
+    1    end-around-carry
   010
```

```
+  101    z1
   111
+  011    z0
1  010
     1    end-around-carry
+  011
```

Coduri aritmetice pentru operanzi cu semn

Codul trebuie să fie complementabil în raport cu R. $R = 2^n$ (complement fată de 2) sau $R = 2^n - 1$ (complement fată de 1), n este numărul de biti ai operandului codat.
Pentru un cod AN, $R - AX$ trebuie să se dividă cu A, adică A trebuie să fie factor al lui R. Dacă e imperativ necesar ca A să fie impar atunci $R = 2^n$ se exclude. Pentru A impar, se poate utiliza numai complementul fată de 1; A trebuie să fie un factor (divizor) al lui $2^n - 1$.
Exemplu: $n = 4$, $R = 2^n - 1 = 15$ pentru complementul fată de 1 si divizibil cu A pentru codul AN cu $A = 3$.
$X = 0110$ este reprezentat prin $3X = 010010$. Complementul la 1 este $101101 = 45_{10}$ care se divide cu 3.
Complementul la 2 al lui $3X$ este $101110 = 46_{10}$, care nu se divide cu 3.
Dacă n = 5 si se ia complementul la 1 atunci $R = 31$ care nu este divizibil cu $A = 3$. $X = 00110$ este reprezentat prin $3X = 0010010$ al cărui complement la 1 este $1101101 = 109_{10}$ – care nu este divizibil cu 3.
În cazul unui cod rezidual cu operanzi cu semn trebuie să fie satisfăcută relatia $A - |X|_A = |R - X|_A$. R trebuie să fie un întreg multiplu al lui A, care, din nou, să permită numai aritmetică pe complementul unitătii. Prin modificarea procedurii astfel încât si complementul la 2 (cu $R = 2^n$) să poată fi utilizat, se obtine

- $|2^n - X|_A = |2^n - 1 - X + 1|_A = |2^n - 1 - X|_A + |1|_A$
- Pentru formarea complementului la 2, este necesară adăugarea la codul rezidual a unui termen de corectie $|1|_A$
- A trebuie să fie un factor al lui $2^n - 1$.

Exemplu de cod rezidual: $A = 7$, $n = 6$, $R = 2^6 = 64$ pentru complementul la 2. $R - 1 = 63$ care este divizibil cu 7. $001010_2 = 10_{10}$ si are reziduul 3 modulo 7. Complementul la 2 al lui 001010 este 110110. Complementul lui $|3|_7$ este $|4|_7$ si prin adunarea termenului de corectie $|1|_7$ se obtine 5, reziduul corect modulo 7 al lui $110110 = 54_{10}$.
Corectia similară este necesară si când se adună operanzii prin complement la 2. Un transfer (carry-out) de pondere 2^n se generează si se pierde. Pentru a-l compensa, se scade $|2^n|_A$ la verificarea rezidualului. Deoarece A este un factor al lui $2^n - 1$, $|2^n|_A = |1|_A$.
Există o interdependentă între unitătile principale si de verificare. În adunarea de complemente ale lui 2, se generează un transfer la care se renunță.

```
110110 = X          101 = |X|7
```

```
+  001101 = Y        +  110 = |Y|7
   ----------           -------------
1  000011            1  011
                        1  transfer la coadă (end-around-carry)
                       -------------
                       100
                    -     1  termen de corectie
                       -------------
                       011
```

Aceasta rezultă într-o interdependentă între unitătile principale si de verificare. O eroare într-o unitate principală se poate propaga la unitatea de verificare si efectul erorii este mascat. O eroare pe un singur bit este totdeauna detectabilă.

Coduri bi-reziduale

Corectarea erorii poate fi obtinută prin utilizarea a două sau mai multe verificări de reziduu. Cazul cel mai simplu este cel al codului bi-rezidual.
Codarea bi-rezudual constă în două verificări de reziduu fată de A_1 si fată de A_2. Cele două numere sunt $A_1 = 2^{a-1}$ si $A_2 = 2^{b-1}$ si sunt două verificări de reziduu de cost redus cu n = c.m.m.c.(a, b) cu n numărul de biti în operanzi. Orice eroare pe un singur bit poate fi corectată.

REDUNDANȚE TEMPORALE

Conceptul de bază din redundanța temporală este repetarea calculului de două sau de mai multe ori si compararea rezultatelor pentru a observa existența unor anumite discrepanțe. Dacă se detectează o eroare, calculele pot fi executate din nou pentru a vedea dacă discordanta rămâne sau dispare. Aceste tratări sunt bune pentru detectarea erorilor datorate unor defecte tranzitorii dar nu oferă protectie la erorile care rezultă din defecte permanente.
O altă formă de redundanță temporală, de data aceasta pentru manipularea defectelor permanente modifică maniera în care se execută calculele a doua oară. O solutie utilizează o logică alternativă pentru circuite combinationale auto-duale care execută o functie pe un set de intrări la un tempo si execută aceeasi functie pe intrări complementate într-un tempo următor: iesirea a doua ar trebui să fie complementul rezultatului primar. Dacă valoarea secundară a functiei nu este complementara primei valori înseamnă că s-a detectat o eroare.
Tratarea următoare utilizează recalculul cu operanzi deplasati (*shifted*) si este aplicabilă pe organizări hardware detaliate la biti (*bit-sliced*). În primul pas se execută calculul normal pe operanzi si rezultatul este stocat într-un registru. În pasul următor operanzii sunt deplasati la stânga cu k biti, iesirea este deplasată la dreapta cu k biti si rezultatul se compară cu cel din calculul precedent. Orice eroare de natură aritmetică sau logică în sectiuni (*slices*) de $k - 1$ biti este astfel detectată. Necesarul de hardware aditional constă în trei unități de deplasare, registrul de stocare pentru rezultatul primului calcul si comparatorul.
O variantă a acestei metode face o recalculare cu operanzi fragmentati si cu fragmente inversate (*swapped*): operatia este executată în doi pasi, mai întâi în forma normală, apoi jumătătile superioare si inferioare ca magnitudine ale operanzilor sunt inversate (*swapped*) astfel că o sectiune (*slice*) de biti defectă operează pe jumătăti diferite ale operanzilor în cele două calcule. Cerintele de hardware suplimentar iau forma câtorva multiplexoare, un registru de memorare si un comparator.

Temporizatoare (timere) *watchdog*

Temporizatoarele *watchdog* au fost utilizate încă de timpuriu în sistemele digitale ca un mod ieftin de detectare a erorilor. În princpiu, separat de procesul monitorizat, este implementat un temporizator. Procesul supravegheat trebuie să reseteze timerul înainte ca acesta să expire; altminteri procesul este considerat afectat de defecte.
Traditional, timerele watchdog sunt utilizate pentru a detecta erorile în fluxul de control, care rezultă din absenta resetării la timp a timerului. Când un timer

expiră, sistemul este resetat. Alternativ, în locul resetării sistemului, se poate genera o întrerupere pentru a initia recuperarea din eroare. Timerele watchdog pot de asemenea să fie utilizate în cam aceiasi manieră cum *timeout*-urile sunt utilizate pentru monitorizarea comportamentului unui singur subsistem. Timeout-urile diferă de timerele watchdog prin aceea că ele asigură o verificare mai fină a fluxului de control.
Timerele de control pot fi implementate fie hardware (timerul este în general ceva extern care poate fi resetat cu un semnal) fie software (rulat adesea pe acelasi procesor pe care este rulat si procesul monitorizat dar timerul este întretinut ca un proces separat).
O implementare nouă a efectului unui timer watchdog fără a utiliza un timer propriu-zis este tehnica de verificare a fluxului de control bazată pe suma de control în dublă precizie. Sumele de control în dublă precizie sunt luate dintr-un bloc de instructiuni fără ramificatii ca suma totală a instructiunilor sau a unei transformate a instructiunilor. Înainte de fiecare bloc, valoarea sumei de control este trimisă unui buffer. Pe măsură ce instructiunile se execută ele sunt scăzute din buffer. Când blocul ajunge la final sau când apare o ramificatie, se trimite un semnal de verificare nul. Dacă bufferul devine zero sau negativ înainte de setarea semnalului, este un semn că a apărut o eroare în fluxul de control. Dacă bufferul este pozitiv când semnalul este setat, si atunci este un semn de eroare.

Exemple de aplicatii ale timerelor watchdog

Pluribus Reliable Multiprocessor. Un exemplu de sistem proiectat cu utilizarea extensivă a timerelor watchdog este multiprocesorul Pluribus. Pluribus a fost confectionat mai întâi pentru scopuri de cercetare; obiectivul principal – fiabilitate înaltă. Comportarea lui Pluribus ca un întreg nu este monitorizată dar timere hardware si software monitorizează aproape fiecare subsistem. Această tratare creste fiabilitatea generală a sistemului deoarece un subsistem care esuează din cauza unui defect intermitent sau tranzitoriu este repornit si nu este lăsat să producă căderea sistemului. Dacă Pluribus utilizează alte tehnici de detectare a erorilor, acele tehnici sunt uzual combinate cu un timer. Timerele acoperă durate de la cinci microsecunde la două minute.
Subsistemele Pluribus functionează ciclic cu o constantă de timp caracterisitică. În timpul fiecărui ciclu subsistemul execută un auto-test complet de consistentă. Traversarea întregului ciclu indică faptul că subsistemul operează corect; o scurgere de timp îndelungată fără ca un timer să fie resetat indică dimpotrivă faptul că subsistemul a suferit o cădere din care nu se poate recupera prin actiune proprie. Un exemplu de subsistem controlat astfel este lista de buffere de mesaj libere, unde sunt stocate bufferele de mesaj când nu sunt în uz. Bufferele părăsesc lista pentru cel mult două minute astfel că un timer de 2 minute este asociat cu bufferul. Dacă un timer depăseste 2 minute, faptul indică o disfunctie. Bufferul nu revine în lista de buffere libere prin actiune proprie si bufferul monitorizat este fortat să revină în listă. În acest caz, disfunctia produce o degradare a performantei de mai putin de două minute, în timpul

căreia sistemul operează cu mai putine buffere de mesaj. Cu toate acestea, eroarea apărută nu produce vreo pierdere de date deoarece timerul facilitează o recuperare completă.
Un alt exemplu este disfunctia unor blocaje (*locks*) de excludere mutuală care sunt prevăzute pe fiecare subsistem. Un lacăt disfunct poate mentine blocată o resursă desi nici un subsistem nu o utilizează. Un subsistem care încearcă să utilizeze acea resursă trece printr-o stare de asteptare. Deoarece lacătul este disfunct, resursa nu va (re)deveni niciodată liberă dar un timer de 1/15 secunde întrerupe procesul si descuie fortat resursa. Cu toată degradarea temporală (1/15 dintr-o secundă) a performantei sistemului, el nu este afectat de eroare. Ca si în primul exemplu, nu se pierd date si este posibilă recuperarea completă.
O eroare mai serioasă, din care timerele watchdog fac o recuperare partială, este disfunctia permanentă a unui procesor. Dacă un procesor cade, orice buffer de mesaje care nu a fost returnat listei de bufere libere va fi returnat prin operatia executată de timerul care monitorizează lista, cum s-a descris mai devreme. Ca si atunci, orice resursă încuiată de procesor este descuiată, deblocată. Desi recuperarea completă nu este imediat posibilă, deoarece procesorul însusi trebuie reparat sau substituit, sistemul rămâne în functie si eroarea poate fi limitată la acel unic procesor.
VAX-11/780. Un sistem multiprocesor proiectat pentru o varietate de aplicatii comerciale, care face uz de timere watchdog este VAX-11/780. Pe acest sistem procesorul consolă rulează un proces watchdog care este resetat când este explorată (*strobed*[7]) o linie de întrerupere. Dacă această explorare făcută de un procesor nu se încheie în 200 de microsecunde, faptul indică o disfunctie si procesorul de consolă încearcă a determina motivul acestei disfunctii.
Centralele Bell System Telephone. Încă un sistem care utilizează procesoare watchdog pentru a detecta erori este sistemul de comutare programată stocat pe telefon de Bell Systems. Timere watchdog externe monitorizează operarea adecvată a programului prin amorsarea recuperării când timerele nu sunt setate periodic. Aceasta permite detectarea preventivă (înainte ca eroarea să se propage si să producă pagube severe sistemului) a problemelor cauzate de erorile software si, în consecintă, cu recuperare mai usoară. Trebuie observat că în detrimentul detectiei erorilor bazate pe principul watchdog, tehnicile de auditare software cunoscute sunt linia principală de apărare în fata erorilor.
Sojournorul martian. Un exemplu de caz în care un timer watchdog a demonstrat abilitatea sa de a detecta erori este legat de misiunea NASA *Mars Pathfinder* care utilizează vehiculul Sojourner. Sistemul de calcul care controlează vehiculul utilizează un sistem de operare în timp real pe mai multe fire (*multithreaded*) cu preîntâmpinare (*preemptive*). Task-urile sunt planificate pe bază de priorităti care reflectă urgenta lor relativă. Datorită unei erori de proiectare, poate apărea o conditie cunoscută ca inversare de prioritate. Pentru a

[7] Cuvântul strobe (strobed) apare în limbajul celor care se ocupă de calculatoare în două prescurtări: RAS (**R**ow **A**ddress **S**trobe) si CAS (**C**olumn **A**ddress **S**trobe). În ambele cazuri este vorba de un semnal de ceas într-un cip de memorie, utilizat pentru a localiza cu precizie linia sau coloana unui bit particular într-o matrice organizată în linii si coloane.

ilustra inversarea de prioritate se consideră scenariul din exemplu următor: (1) un fir de prioritate scăzută obtine un blocaj (*lock*) mutual exclusiv al accesului la date în indiviziune, partajate, (2) în aceste conditii un task îndelungat cu prioritate mai înaltă decât a firului de prioritate joasă este planificat datorită unei întreruperi si (3) firul cu prioritate superioară are nevoie de acces la datele blocate de task-ul de prioritate inferioară. Ca rezultat (1) task-ul de prioritate inferioară este oprit de la executie de firul de prioritate mai înaltă si (2) task-ul de prioritate mai înaltă este si el oprit de la executie deoarece el se blochează asteptând ca firul de prioritate inferioară să descuie lacătul (*lock*).
Utilizarea timerului watchdog face ca scenariul de mai sus să fie detectat si sistemul să fie repornit. Cu toate acestea, restartul complet produce pierdere de date si resetările repetate limitează serios lucrul corect al sistemului de deplasare pe Marte. Problema a fost ulterior detectată si software-ul a fost remediat (*patched*) pentru a restabili comportarea potrivită.
În acest sistem metoda de recuperare s-a aplicat când timeout-ul de watchdog este un reset de sistem traditional, o măsură drastică dar robustă care reprezintă o practică inginerească bună. Disponibilitatea sistemului este mult mai importantă decât pierderea de date datorată resetării sistemului.

Limitările timerelor watchdog

Timerele watchdog nu sunt perfecte în ceea ce priveste detectarea erorilor în sistemele numerice. Motivele sunt patru:

1. Desi detectia erorilor nu este limitată la vreun model al defectelor precizat, timerele watchdog detectează numai erorile de un tip foarte special. Presupunerea este că orice eroare se va manifesta ca eroare pe fluxul de control astfel încât sistemul nu continuă a reseta timerul. Dacă apare o eroare pe fluxul de control dar programul resetează timerul la timp, eroarea va trece nedetectată.
2. Resetările timerului trebuie plasate cu grijă pentru a fi eficace. Ele nu pot fi plasate în interiorul rutinelor de întrerupere sau al buclelor (pentru a evita buclele infinite), dar pot apărea suficient de frecvent astfel ca timerul să nu poată expira pe durata unei operări normale.
3. Pot fi verificate numai procesele cu durate de executie relativ deterministe deoarece detectia erorilor se bazează pe de-a-ntregul pe timpul dintre resetările timerului. Dacă timpul de setare este mai scurt decât durata de executie cea mai lungă posibilă a procesului verificat, el poate expira chiar dacă nu există vreo eroare. Pe de altă parte, dacă timpul de setare este prea lung, atunci chiar dacă apare o eroare pe fluxul de control, procesul poate avea timp suficient de a reveni la punctul la care timerul este resetat si eroarea nu va fi detectată.
4. Un timer watchdog furnizează numai o indicatie de disfunctie posibilă a procesului; un proces esuat partial poate el însusi să reseteze timerul. Acoperirea este limitată, deoarece nu sunt verificate nici datele nici rezultatele. Când este utilizat pentru a reseta sistemul, un timer watchdog

poate îmbunătăti disponibilitatea (timpul mediu de recuperare se scurtează) dar nu fiabilitatea (aparitia disfunctiilor este la fel de probabilă si după resetare). Când disponibilitatea unui sistem digital este mai importantă decât pierderea de date în anumite conditii, utilizarea unui timer watchdog pentru resetarea sistemului la detectarea unei erori este o alegere potrivită.

Heartbeats

Heartbeats reprezintă o tratare comună a detectării disfunctiilor din procese si noduri în ambianta calculului distribuit (rețelizat). Periodic, o entitate monitorizantă trimite un mesaj (*a heartbeat* – o bătaie de inimă) la un nod sau la un proces monitorizat si asteaptă un răspuns. Dacă nodul monitorizat nu răspunde într-un interval de timp predefinit, este declarat disfunct si se initiază actiunea potrivită de recuperare.

Limitări ale heartbeat-ului traditional

Sunt două probleme majore asociate cu schema traditională de heartbeat:

- Perioada de timeout este prenegociată de cele două părti sau uneori chiar hard-coded[8] de programator. Valoarea de timeout predefinită nu se poate adapta la schimbările de trafic în retea sau la variabilitatea încărcării individuale pe noduri. În cazul traficului intens, al încărcării mari a nodurilor sau al prezentei unui nod lent, valoarea de timeout poate fi prea scurtă si face ca monitorizarea nodurilor să declare un nod sănătos ca fiind defect. O astfel de alarmă falsă este de nedorit într-un mediu distribuit, în special pentru aplicatii critice cum sunt cele din banking-ul comercial si sistemele de baze de date.
- Nodul monitorizat este presupus a fi sănătos dacă este capabil a răspunde la mesajul heartbeat. Acest fapt este acceptabil uzual pentru aplicatiile cu un singur fir (*single-threaded*). Cu toate acestea, într-o aplicatie multifir, un fir independent al executiei este răspunzător de răspunsul la mesajul heartbeat. Operarea sănătoasă a acestui fir nu implică în mod necesar operarea sănătoasă a întregii aplicatii multithread. Alte fire din proces pot fi în situatie de blocaj, situatie care previne progresul, avansul întregii aplicatii, altfel spus, alte fire ar putea opera într-o stare coruptă care interzice procesului producerea unui serviciu corect.

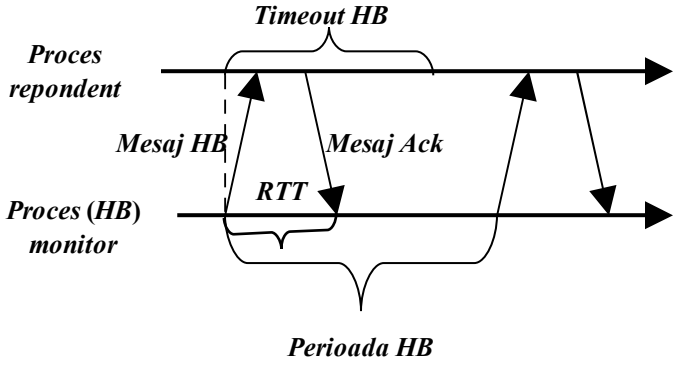
Algoritmii heartbeat *adaptivi* si *inteligenti* rezolvă aceste două probleme. Un algoritm heartbeat este numit *adaptiv* dacă valoarea de timeout utilizată de monitor nu este fixată ci este negociată periodic între cele două părti pentru adaptare la schimbările traficului din retea sau la variatiile de încărcare a nodului. Un algoritm heartbeat este denumit *inteligent* dacă entitatea care este

[8] **hard-coded** – în jargonul programării înseamnă valori de date sau comportamente (pre)scrise direct într-un program, posibil în mai multe locuri, unde nu pot fi modificate prea usor. Sunt mai multe posibilităti, depinzând de cât de des valoarea s-ar putea schimba.

monitorizată excită un set de verificări predefinite pentru a testa robustetea întregului proces si numai apoi răspunde procesului de monitorizare.

Proiectarea heartbeat-urilor adaptive, inteligente

Pentru a ilustra conceptele de *heartbeat adaptiv* si *heartbeat inteligent*, sunt create două procese multifir independente, unul care răspunde la mesajul heartbeat, altul heartbeat-ul care monitorizează.
Heartbeat-ul *monitor* este entitatea care monitorizează fiind răspunzătoare de trimiterea periodică de mesaje cerere de heartbeat către nodul tintă. *Heartbeat*-ul care *răspunde* este entitatea monitorizată si răspunde la mesajele cerere de heartbeat trimise de monitor. Schema *adaptivă* utilizează algoritmul Jacobson care permite ajustarea valorii de timeout potrivit performantei măsurate a retelei în termeni de timp dus-întors (RTT – round trip time) la transmiterea mesajului. Algoritmul heartbeat este făcut *inteligent* (adică capabil a verifica robustetea întregului proces) prin utilizarea unui mesaj de test *null* în interiorul procesului pentru a testa operarea sănătoasă a tuturor firelor din proces. În paragrafele următoare se prezintă implementări ale acestor două scheme.
Figura următoare reprezintă protocolul heartbeat. Periodic, monitorul heartbeat trimite un mesaj către procesul care răspunde la heartbeat, sterge contorul *ack_missed* si porneste timerul. Durata timerului este dictată de valoarea curentă a variabilei timeout asociată cu procesul care răspunde la heartbeat.



Protocolul heartbeat (HB) adaptiv

Pe de altă parte, procesul care răspunde la heartbeat răspunde cu un mesaj de confirmare a receptiei heartbeat-ului. Dacă acest mesaj de confirmare (acknowledge) este primit de monitorul heartbeat înainte de expirarea timpului potrvit, monitorul presupune că procesul îndepărtat este viu, altminteri contorul *ack_missed* este incrementat. Dacă contorul nu a atins valoarea sa maximă, poate fi trimis încă un mesaj heartbeat de la monitor la respondent, altminteri procesul monitorizat este presupus a fi defect.

Cruciale pentru acest protocol sunt valorile de timeout si perioda de heartbeat. În general perioada poate fi fixată ca un multiplu al valorii curente de timeout. Este totusi de dorit a avea o valoare de timeout care se adaptează la timpul curent de răspuns al procesului respondent. Timpul de răspuns, cum este văzut de monitor este o functie de încărcarea curentă pe masina de departe si de timpul necesar pentru a transmite mesajul si răspunsul de confirmare a heartbeat-ului, adică timpul de răspuns este o functie de RTT.
Pentru a calcula RTT este suficient a include în mesajul heartbeat o marcă de timp a cărei valoare este timpul de trimitere. Această marcă de timp (*timestamp*) va fi trimisă înapoi la monitor de respondent; astfel, când monitorul primeste o confirmare de heartbeat, poate calcula instantaneu RTT ca diferenta dintre timpul curent si acea marcă de timp. Totusi, se dovedeste că o astfel de solutie nu face fată cazului unei încărcări variabile. Problema principală este cea a variabilitătii valorii instantanee a RTT, care poate prezenta oscilatii substantiale. Este necesară o estimare a RTT mai precisă si mai netedă.
O abordare a acestei probleme se găseste în algoritmul lui Jacobson (care s-a dovedit eficace în cazul încărcărilor variabile si este implementat curent în protocolul TCP). Pentru fiecare respondent la heartbeat, monitorul heartbeat mentine un RTT variabil care este cea mai bună estimare curentă a timpului dus-întors la destinatia în discutie. Când este trimis un mesaj heartbeat, se porneste un timer. Dacă confirmarea revine înainte de expirarea timpului, monitorul măsoară cât timp consumă sosirea confirmării; fie M această valoare. Apoi se actualizează RTT potrivit formulei

$$\text{RTT} = \alpha\text{RTT} + (1 - \alpha)M$$

cu α un factor de netezire care exprimă ce pondere se acordă valorii vechi. Tipic, α este fixat la 7/8.
Chiar fiind dată o valoare bună pentru RTT, alegerea unei valori de timeout nu este tocmai simplă. O alegere ar putea fi β.RTT, dar este greu de ales β. Mai mult, experienta a arătat că o valoare constantă pentru β este inflexibilă deoarece ea nu reuseste să acopere dispersii mari ale valorilor RTT. Jacobson propune a lua β proportional cu deviatia standard a functiei densitate de probabilitate a timpului de sosire a confirmării. În acest mod, o dispersie mare face pe β mare si invers. În particular, el sugerează utilizarea deviatiei medii care este o estimare ieftină a deviatiei standard. Algoritmul lui cere retinerea secventială a unei alte variabile netezite, D, deviatia. Ori de câte ori soseste o confirmare heartbeat, se calculează diferenta între valoarea asteptată si cea observată, $|\text{RTT} - M|$. O valoare netezită a lui D este mentinuă după formula

$$D = \alpha D + (1 - \alpha)|\text{RTT} - M|$$

cu α acelasi sau diferit de cel din expresia de netezire a RTT. Cu toate că deviatia D nu este exact egală cu deviatia standard, aceasta este destul de bună si este posibil a calcula D si RTT în mod foarte eficace utilizând numai adunări, scăderi si deplasări de întregi. Valoarea initială a timeout-ului este calculată după cum urmează:

$$\text{RTT} = (7/8)\text{RTT} + (1/8)M$$

$$D = (3/4)D + (1/4)|\text{RTT} - M|$$

Utilizând aceste două numere, valoarea de timeout este calculată ca RTT + 4*D*. Mai mult, dacă timerul expiră înainte ca monitorul heartbeat să primească confirmarea de heartbeat, valoarea de timeout este dublată în maniera retractilă a pasului înapoi (*back-off*).

Implementarea si evaluarea de heartbeat inteligent

Schema heartbeat-ului inteligent este implementată în cadrul procesului heartbeat respondent. La primirea unui mesaj cerere de heartbeat de la monitorul de heartbeat, firul respondent initiază o rundă de mesje surogat (MSG_HB_NULL_TEST) către toate firele din proces. Pentru a procesa un mesaj, un anumit fir obtine mai întâi un *lock* pentru structura de date locală. Apoi incrementează un contor în mesaj si relaxează *lock*-ul elementului structură de date. Dacă procesul operează corect, adică nu există un blocaj (*deadlock*), contorul din mesajul *null* va atinge ulterior numărul total de fire curent active în proces. Ultimul fir din lant care vede această conditie satisfăcută trimite un MSG_HB_NULL_TEST_REPLY înapoi la respondentul de heartbeat (această transmisie este executată în cadrul aceluiasi proces). La primirea acelui MSG_HB_NULL_TEST_REPLY, elementul respondent presupune că întregul proces este într-o stare de sănătate liberă de blocaje si trimite apoi către monitor un mesaj de confirmare de heartbeat. Dacă, totusi, există fire în proces în stare blocată, mesajul de test *null* este blocat si respondentul nu trimite la monitor un mesaj de confirmare. Ulterior, monitorul intră în timeout si declară disfunct procesul monitorizat.

Checkpointarea

Disfunctii la executarea programelor

Azi calculatoarele sunt mult mai rapide, iar aplicatiile sunt din ce în ce mai complicate. Printre aplicatiile care sunt încă mari consumatoare de timp se pot mentiona:

- Actualizarea bazelor de date.
- Simulările curgerii fluidelor, necesare în particular pentru modelarea meteo si climatică.
- Optimizarea, alocarea optimă a resurselor economice (de pildă în utiliarea liniilor aeriene).
- În astronomie: simularea în problema celor *n* corpuri si modelarea universului.
- În biochimie: studierea compusilor proteinici.

Când timpul de executie a calculelor este foarte lung, atât probabilitatea de a claca în timpul executiei cât si costurile unui astfel de eveniment devin semnificative.

Un exemplu relativ la costul întreruperii executiei unui program:
Rularea completă a programului consumă T ore. Sistemul suferă disfunctii tranzitorii la o rată de λ disfunctii pe oră. Disfunctia este instantanee dar tot efortul de calcul anterior este pierdut.
E este timpul total de executie asteptat statistic (*expected*), timp care include si orice efort de calcul pierdut din cauza disfunctiilor apărute.
Dacă nu sunt întreruperi, cu alte cuvinte dacă nu apar disfunctii în timpul executiei (un caz cu probabilitatea $e^{-\lambda t}$), timpul total de executie asteptat (mediu) conditionat este T.
Probabilitatea aparitiei unei disfunctii la momentul τ al executiei este $\lambda e^{-\lambda\tau}\, d\tau$.
Într-un asemenea caz, τ ore sunt risipite, programul trebuie reluat si un timp suplimentar aditional E este de asteptat pentru a încheia calculul. Timpul mediu asteptat pentru încheierea calculului este asadar $\tau + E$.
Pentru a calcula costul disfunctiei se face o mediere pe toate cazurile:

$$E = Te^{-\lambda T} + \int_0^T (\tau + E)\lambda e^{-\lambda \tau} d\tau = E(1 - e^{-\lambda T}) + (1 - e^{-\lambda T})/\lambda$$

o ecuatie în E cu solutia $E = (e^{\lambda T} - 1)/\lambda$.
O măsură a overhead-ului (comparat cu T) este

$$\eta = E/T - 1 = (e^{\lambda T} - 1)/(\lambda . T) - 1$$

Overhead-ul relativ η depinde numai de produsul $\lambda . T$ care este numărul mediu de disfunctii pe durata executiei programului. Acest η creste foarte rapid (exponential) cu $\lambda . T$.
Este de preferat, desigur, a face în asa fel ca lucrul să nu fie reluat de la început dintr-o stare cu toate disfunctiile din nou posibile. Solutia este asa-numita *checkpointare* (utilizarea unor puncte de control).

Definitia checkpointării

Un checkpoint este un instantaneu (aproape în sensul fotografic) al întregii stări a procesului de calcul la momentul când acel checkpoint este prelevat: se retine toată informatia necesară pentru repornirea procesului din acel punct.
Checkpoint-ul este salvat într-o memorie stabilă, de o fiabilitate suficientă. Cele mai utilizate ca memorie stabilă sunt de obicei discurile. Discurile pot mentine date chiar dacă, de pildă, alimentarea cade (cădere care nu produce vătămări fizice ale suprafetei active). În plus, discurile pot retine cantităti enorme de date la un cost foarte scăzut. Si checkpoint-urile pot fi foarte cuprinzătoare: zeci sau chiar sute de megabytes.
Adesea este utilizată ca memorie stabilă si RAM cu o baterie de backup.
Nici un mediu nu este perfect fiabil; fiabilitatea trebuei să fie suficient de înaltă pentru aplicatia *at hand*.

Overhead-ul si latența unui checkpoint

Overhead-ul checkpointării este cresterea timpului de executie al aplicatiei datorată preluării de checkpoint-uri.
Latenta checkpointării este timpul necesar pentru salvarea checkpoint-urilor.
Într-un sistem simplu, overhead-ul si latenta coincid, sunt identice. Dacă parte din checkpointare poate fi suprapusă cu executia aplicatiei, latenta poate fi substantial mai mare decât overhead-ul. Dacă un proces face checkpointarea prin transcrierea stării sale într-un buffer intern, unitatea centrală (CPU) poate continua executia în timp ce checkpointul este transferat din buffer pe disc.

Latența checkpointării – un exemplu

```
for (i = 0; i < 1000000; i ++)
        if (f(i) < min) {min = f(i); imin = i;}

for (i = 0; i < 100; i ++) {
        for (j = 0; j < 100; j ++) {
                e[i][j] += i*j/min;
        }
}
```

Prima parte a acestui scurt program face calculul celei mai mici valori a unei functii $f(i)$ pentru $0 < i < 1000000$. Partea a doua execută o multiplicare urmată de o împărtire.
Ce se poate spune despre dimensiunea checkpointului din acest exemplu? Întrebarea este cât se poate de legitimă deoarece dimensiunea checkpointului determină latența acelui checkpoint. Dimensiunea poate varia de la program la program si chiar în timpul executiei unui aceluiasi program.
Un checkpoint în prima parte a exemplului dat poate fi redus: numai contorul programului si variabilele *min* si *imin*; multe alte registre sunt irelevante.
Un checkpoint prelevat în partea a doua trebuie să includă ceea ce s-a calculat din masivul $e[i][j]$ până la momentul respectiv.
În general, dimensiunea checkpointului este dependentă de program si, cum s-a mai spus, poate fi de la câtiva kilobytes la mai multi gigabytes.
Sunt câteva probleme în legătură cu checkpointarea. Iată-le enumerate:

- Câte checkpointuri sunt necesare?
- În ce puncte din executia programului se cuvine a se preleva un checkpoint?
- Cum se poate reduce overhead-ul de checkpointare?
- Cum se poate face checkpointarea în sisteme distribuite în care un control central poate exista sau poate să nu existe?
- La ce nivel (kernel/user/application) trebuie făcută checkpointarea?
- Cât de transparent fată de utilizator trebuie să fie procesul de checkpointare?

Checkpointarea la nivel de kernel

Procedurile de checkpointare sunt incluse în kernel, sunt transparente pentru utilizator si nu necesită schimbări în program.
Când sistemul reporneste după cădere, kernelul este responsabil pentru administrarea operatiei de recuperare.
Orice sistem de operare preia checkpointuri când un proces este întrerupt de un altul; starea procesului este înregistrată astfel că executia poate fi reluată din punctul de întrerupere fără pierdere de efort de calcul.
Multe sisteme de operare au putine checkpointuri orientate pe toleranta la defecte sau nu au deloc.

Checkpointarea la nivel de utilizator

În această variantă, biblioteca de nivel utilizator este cea pregătită să execute checkpointarea. Pentru a face checkpointarea, programele aplicatii sunt conectate la aceste biblioteci.
Ca si checkpointarea la nivel de kernel, această abordare nu cere în general vreo schimbare în codul aplicatiei dar legarea explicită cu biblioteca la nivelul utilizatorului este cerută în mod necesar.
Biblioteca la nivel de utilizator gestionează si recuperarea din starea de disfunctie.

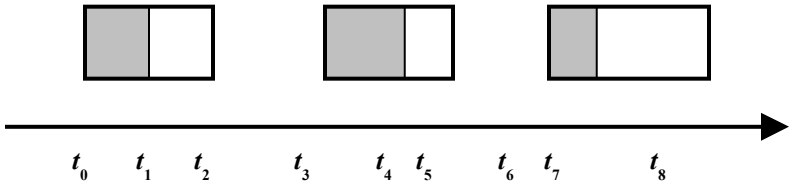
Checkpointarea la nivel de aplicatie

Aplicatia este responsabilă pentru toate functiile de checkpointare, codul pentru checkpointare si pentru recuperare este parte din aplicatie. Acesta este cel mai important control asupra procesului de checkpointare, dar este costisitor de implementat si de depanat.
Firele (threads) sunt invizibile la nivel de kernel, dar nivelurile de utilizator si de aplicatie nu au acces la informatia detinută la nivel de kernel; nivelurile de utilizator si de aplicatie nu pot atribui un identificator (ID) de proces particular pentru un proces de recuperare.
Nivelurilor de utilizator si de aplicatie s-ar putea să nu li se permită checkpointarea unor părti din sistemul de fisiere. Ar putea în schimb să memoreze nume de fisiere si pointeri la fisiere.

Model analitic pentru latență si overhead

Overhead-ul este partea din checkpointare care nu se execută în paralel cu aplicatia.
Latența este timpul între t_{start} (când operatia de checkpointare începe) si t_{end} (când operatia se încheie).
Overhead-ul are impact mai mare asupra performantelor decât latența.

Checkpointul preia si reprezintă starea sistemului la t_{start}.
Overhead-ul este partea din $[t_{start}, t_{end}]$ pe durata căreia aplicatia este oprită din executie din cauza checkpointării (CPU este ocupată cu checkpointarea).
Notarea overhead-ului cu t_c face intervalul de overhead $[t_{start}, t_{start} + t_c]$.



În figură, dreptunghiurile reprezintă latenta, iar partea umbrită reprezintă overhead-ul.
Dacă apare o disfunctie în intervalul $[t_{start}, t_{end}]$, checkpointul preluat este inutil si sistemul trebuie să revină la checkpointul precedent.
Exemplu: dacă disfunctia apare în intervalul $[t_3, t_5]$, se revine la checkpointul precedent, adică la starea procesului de la timpul t_0.
Dacă t_r exprimă timpul mediu de recuperare, acesta este timpul petrecut în starea cu defect plus timpul de recuperare la o stare functională (adică timpul de re-bootare completă a procesorului).
Dacă la momentul τ apare o disfunctie tranzitorie, procesul redevine activ la momentul $\tau + t_r$.
Câteva notatii suplimentare necesare pentru a scrie modelul analitic:
I – intervalul între checkpointuri, timpul între executarea checkpointului i si a checkpointului $i + 1$.
$E_{interval}$ – durata medie statistică (expected value) a lui I.
T – timpul cheltuit pentru executarea aplicatiei pe această perioadă; dacă nu apare vreo disfunctie, $I = T + t_c$.
t_l – latența, egală cu diferența $t_{end} - t_{start}$.
Dacă disfunctia apare la τ ore în intervalul I, lucrul pierdut include:

- Lucrul executat pe durata τ
- Lucrul util executat în timpul $[t_{start}, t_{end}] = t_l - t_c$.

t_r – timpul mediu de recuperare si de reluare a calculului.
Timpul suplimentar total datorat unei disfunctii la τ ore în intervalul I este

$$\tau + t_l - t_c + t_r$$

O aproximare de primul ordin pentru intervalul dintre checkpointuri

Se presupune că nu mai mult de o disfunctie poate lovi sistemul între două checkpointuri, ceea ce este o aproximare bună dacă $T + t_c$ este mic comparativ cu $1/\lambda$, timpul mediu între aparitia a două disfunctii.
Timpul mediu între două checkpointuri succesive se evaluează pe baza unei priviri asupra a două cazuri:

Cazul 1: între două checkpointuri succesive nu apare nici o disfunctie. Durata scursă este $T + t_c$, iar probabilitatea cazului este $e^{-\lambda (T+t_c)}$.
Cazul 2: între două checkpointuri succesive apare o disfunctie. Probabilitatea aproximativă este $1 - e^{-\lambda (T+t_c)}$.
Timpul aditional datorat disfunctiei: $\tau + t_r + t_l - t_c$.
Media timpului τ este $(T + t_c)/2$.
Timpul mediu (asteptat) aditional este $(T + t_c)/2 + t_r + t_l - t_c$.
La calculul lungimii medii a intervalului dintre checkpointuri avem:
contributia cazului 1:

$$(T + t_c)e^{-\lambda(T+tc)}$$

si contributia cazului 2:

$$(1 - e^{-\lambda(T+tc)})[(T + t_c)/2 + T + t_c + t_r + t_l - t_c] =$$
$$= (1 - e^{-\lambda(T+tc)})[3T/2 + t_c/2 + t_r + t_l]$$

Prin însumarea celor două contributii se obtine:

$$E_{interval} \approx 3T/2 + t_c/2 + t_r + t_l - (T/2 - t_c/2 + t_r + t_l)e^{-\lambda(T+tc)}$$

Sensibilitatea acestui $E_{interval}$ la t_l si t_c este dată de derivatele (partiale) în raport cu variabilele respective:

$$\partial E_{inter}/\partial t_c \approx \frac{1}{2} + \left[\frac{1}{2} + \lambda\left(\frac{T - t_c}{2} + t_r + t_l\right)\right] e^{-\lambda (T+t_c)}$$
$$\partial E_{inter}/\partial t_l \approx 1 - e^{-\lambda (T+t_c)}$$

Dacă $\lambda(T + t_c) << 1$ atunci derivata în raport cu t_c este mult mai mare decât cea în raport cu t_l. Speranta E este mult mai sensibilă la overhead decât la latență si, de aceea, t_c trebuie mentinut scăzut chiar dacă t_l creste.

Reducerea overheadului prin bufferizare

Se poate face ca procesul să scrie checkpointul în memoria principală si apoi să revină la executia aplicatiei. Memoria cu acces direct (DMA) este utilizată pentru a copia checkpointul din memoria principală pe disc. DMA reclamă implicarea CPU numai la începutul si la încheierea operatiei.
Este posibilă o rafinare a acestei posibilităti de copiere prin scriere bufferizată. Se are în vedere faptul că copierea unor portiuni din starea procesului care nu s-au schimbat de la ultimul checkpoint este pierdere de timp. Dacă procesul nu actualizează prea frecvent unele pagini din memoria principală, mult din efortul de copiere într-o zonă buffer poate fi evitat.

Copiere cu scriere bufferizată

Multe sisteme de memorare sunt prevăzute cu biti de protectie pentru memorie (pentru o pagină de memorie fizică principală) care indică fie că pagina este read-write, fie că este read-only, fie că este inaccesibilă.
Când checkpointul este prelevat, bitii de protectie ai paginilor care apartin procesului sunt setati ca read-only. Aplicatia continuă rularea în timp ce

paginile de checkpoint sunt transferate pe disc. Dacă aplicatia încearcă să actualizeze pagina, se amorsează o violare a tipului de acces. Sistemul bufferizează atunci pagina si tipul de acces se setează la read-write. Pagina bufferizată este copiată mai târziu pe disc.

Reducerea overhead-ului de checkpointare – excluderea de memorie

Două tipuri de variabile nu trebuie să fie checkpointate: cele care nu au fost actualizate si cele care sunt "moarte". O variabilă moartă este o variabilă a cărei valoare prezentă nu va mai fi utilizată nicodată în program.
Sunt două genuri de variabile moarte: acelea care nu vor mai fi niciodată adresate de program si acelea pentru care următorul acces este pentru scriere. Provocarea este a identifica corect astfel de variabile.

Identificarea de variabile moarte

Spatiul de adresare al unui proces are patru segmente: codul, datele globale, heap-ul si stack-ul.
Identificarea în cod a variabilelor moarte este facilă: codul cu automodificare nu se mai utilizează, astfel, segmentul de cod din memorie este read-only si nu mai trebuie checkpointat.
Segmentul stack este la fel de accesibil. Continutul de adrese retinute în locatii situate sub pointerul de stack sunt evident moarte (spatiul de adresare virtual are uzual segmentul de stack la vârf, crescător în jos).
Segmentul heap: multe limbaje de programare permit programatorului să aloce si să elibereze explicit memorie (exemple: apelurile *malloc*() si *free*() utilizate în C). Continutul listei *free* este mort prin definitie.
Unele pachete de checkpointare la nivel de utilizator (de pildă *libckpt*) furnizează programatorului apeluri la proceduri (de pildă *checkpoint_here*()) care specifică regiuni de memorie care ar trebui excluse din sau incluse în checkpointuri viitoare.

Reducerea latenței

Compresia la checkpointare are ca rezultat un volum mai mic de scris pe disc.
Cât de mult se câstigă prin compresie depinde de:

- extinderea compresiei, care este dependentă de aplicatie si poate varia între 0 si 50%.
- efortul cerut de algoritmul de comprimare, care este executat de CPU si se adaugă la overhead-ul checkpointării ca si la latență.

În checkpointarea simplă secventială unde $t_c = t_l$, compresia poate aduce avantaje.
În sistemele mai eficiente, în care $t_c << t_l$, utilitatea acestei solutii este discutabilă si trebuie evaluată cu grijă înainte de a fi utilizată.

Amplasarea checkpointurilor

Asezarea în timp (si în program) a checkpointurilor trebuie să aibă în vedere o echilibrare între costuri si beneficii. Se urmăreste uzual minimizarea duratei de executie a unei aplicatii de durată.
Costul, considerat a fi timpul pentru stocarea unui checkpoint poate fi mare. Câteva notatii:
t_x – durata executiei fără checkpointare.
t_c – timpul mediu consumat cu prelevarea unui checkpoint.
N – o variabilă de decizie cu semnificatia numărul de checkpointuri plasate uniform în job pentru minimizarea timpului total de executie $T_{tot}(N)$.
$\tau_x = t_x/N$ – timpul (mediu) între două checkpointuri succesive.
Disfunctiile apar cu rata λ. Disfunctiile sunt tranzitorii, ele dispar după o durată de existentă t_f.
La manifestarea unei disfunctii, sistemul revine la checkpointul cel mai recent.
Checkpointurile sunt presupuse a se afla într-o memorie sigură, care nu poate fi coruptă de vreo disfunctie.

Model analitic pentru amplasarea checkpointurilor

t_l – timpul total pierdut pentru fiecare disfunctie tranzitorie.
t_f – timpul cât sistemul este disfunct.
Dacă disfunctia apare pe durata checkpointului, se asociază aparitiei o probabilitatea $P_c = t_c/(t_c + \tau_x)$. Timpul irosit este în acest caz $\tau_x + t_c/2$.
Dacă disfunctia apare pe durata executiei se lucrează cu probabilitatea $P_x = \tau_x/(t_c + \tau_x)$. Timpul irosit este acum $\tau_x/2$.

$$t_l = t_f + P_c(\tau_x + t_c/2) + P_x(\tau_x/2) = t_f + (t_c + \tau_x)/2.$$

Rezultatul obtinut este intuitiv: $(t_c + \tau_x)/2$ este jumătate din intervalul $t_c + \tau_x$.

Amplasarea optimă a checkpointurilor

Se presupune că λ este suficient de mic astfel încât probabilitatea disfunctiei în timpul revenirii (*rollback*) este neglijabilă.
Numărul de disfunctii mediu statistic (*expected*) pe durata timpului total de executie de $t_x + N\,t_c$ este $\lambda(t_x + N\,t_c)$.
Timpul total consumat:

$$T_{tot}(N) = t_x + N\,t_c + \lambda(t_x + N\,t_c)[t_f + (t_c + t_x/N)/2]$$

Selectarea lui N pentru minimizarea timpului $T_{tot}(N)$

$$dT_{tot}(N)/dN = t_c + \lambda t_c(t_c/2 + t_f) - (\lambda t_x^2)/(2N^2)$$

si prin anularea derivatei se obtine

$$N_{opt} = \frac{t_x\sqrt{\lambda}}{\sqrt{t_c(2 + \lambda\,t_c + 2\lambda\,t_f)}}$$

Numărul N_{opt} trebuie să fie un întreg, cel care prin trunchiere sau prin rotunjire în plus să minimizeze $T_{tot}(N)$. Rezultă imediat intervalul optim între două checkpointuri succesive: $\tau_{opt} = t_x/N_{opt}$.
Exercitiu: Relaxati ipoteza că probabilitatea unor disfunctii suplimentare în timpul procesului de recuperare este neglijabilă.
Amplasarea uniformă este optimă dacă checkpointarea are un cost constant pe toată durata executiei.
Dacă dimensiunea (deci durata) checkpointului variază mult de la o parte a executiei la alta, timpul optim dintre checkpointuri nu mai are un pas constant.

Plasarea optimă a checkpointurilor: un model la nivelul instructiunilor

Probabilitatea unei defectiuni pe durata executării unei instructiuni depinde de unitătile functionale utilizate si de timpul de executie al acelei instructiuni.
Se defineste variabila de decizie M care reprezintă numărul de instructiuni între două checkpointuri consecutive.
Este vizată minimizarea lui W, timpul consumat cu o instructiune.
Multimea instructiunilor este partitionată în N submultimi de instructiuni similare. Pentru o instructiune de tipul i, timpul de executie este T_i, rata de defectare este λ_i, frecventa este f_i ($\Sigma f_i = 1$).
Se notează, de asemenea, cu s ($1 - s$) fractia de defecte permanente (tranzitorii).
Constanta μ_i reprezintă rata de "reparare" a disfunctiilor tranzitorii pentru cazul unei instructiuni de tipul i.
În continuare sunt date alte câteva notatii utilizate în scrierea modelului la nivelul de instructiune.
Evenimentele posibile în timpul executării unei instructiuni sunt:
H^C – instructiunea este executată cu succes când este executată prima dată; probabilitatea evenimentului este p^C.
H^{RB} – instructiunea esuează, disfunctia e identificată, programul este readus (*rolled-back*) la ultimul checkpoint, instructiunea este executată; probabilitatea asociată este p^{RB}.
H^{PF} – revenirea (*roll-back*) esuează si ea, programul esuează, programul este reîncărcat si repornit; probabilitatea este p^{PF}.
p_i^C, p_i^{RB}, p_i^{PF} sunt probabilităti conditionate pentru instructiunile de tipul i, trecut ca indice.
Aceste probabilităti conditionate se calculează si apoi se mediază

$$p^{(j)} = \sum_{i=1}^{N} f_i p_i^{(j)} \quad (j = C, RB, PF)$$

Sunt prezentate imediat alte notatii pentru un sistem cu rata de defectare λ si cu rata de reparare μ.
Probabilitatea lipsei defectelor în intervalul de timp $(0, t)$

$$P_0(\lambda, t) = e^{-\lambda t}$$

Probabilitatea tranzitiei de la starea lipsită de defecte la timpul 0 la starea lipsită de defecte de la momentul t

$$P_{00}(\lambda,\mu,t) = \frac{\mu}{\lambda+\mu} + \frac{\lambda}{\lambda+\mu} e^{-(\lambda+\mu)t}$$

Pentru $0 \le t_1 \le t$, probabilitatea tranzitiei de la starea lipsită de defecte la momentul 0 la starea lipsită de defecte la momentul t, cu cel putin o defectare în perioada $(0, t_1)$

$$\overline{P}_{00}(\lambda,\mu,t_1,t) = P_{00}(\lambda,\mu,t) - e^{-\lambda t} P_{00}(\lambda,\mu,t-t_1)$$

Probabilitătile din modelul la nivel de instructiune

$$P_i^{(C)} = P_0(\lambda_i, T_i) = e^{-\lambda_i T_i}$$

M – numărul de instructiuni dintre checkpointuri.

m – numărul de instructiuni dintre instructiunea cu disfunctie si ultimul checkpoint = 1, …, M, fiecare cu probabilitatea $1/M$.

$P_{i,m}^{RB}$ – probabilitatea conditionată a revenirii izbutite, fiind dat tipul i si numărul m de instructiuni executate de la ultimul checkpoint.

δ_1 – timpul de setup necesar pentru a initia revenirea programului, inclusiv timpul necesar pentru a încărca informatia salvată la ultimul checkpoint.

$$P_{i,m}^{(RB)} = \overline{P}_{00}[(1-s)\lambda_i, \mu_i, T_i, T_i] P_0(s\lambda_i, T_i) P_0(\lambda_i, \delta_1) [P^{(C)}]^{m-1} P_i^{(C)}$$

$$P_i^{(RB)} = \overline{P}_{00}[(1-s)\lambda_i, \mu_i, T_i, T_i] P_0(s\lambda_i, T_i) \frac{1-[P^{(C)}]^M}{1-P^{(C)}} \frac{P_i^{(C)}}{M}$$

$$P_i^{(PF)} = 1 - P_i^{(C)} - P_i^{(RB)}$$

Modelul la nivelul instructiunii serveste la calculul lui W.

Cu τ timpul mediu pentru a executa cu succes o instructiune si cu T_s timpul consumat de checkpointare se poate scrie expresia timpului cheltuit pe instructiune

$$W = \tau + T_s/M.$$

Prin cresterea lui M, numărul de instructiuni între checkpointuri, se obtin cresteri ale primului termen si descresteri ale termenului secund.

$T = \Sigma f_i T_i$ este durata medie de executie a instructiunii fără defecte.

δ_2 este timpul mediu necesar diagnozei si reparatiei.

L notează numărul mediu de instructiuni pe program si

$$\tau = \overline{T} + P^{(RB)}\left(\delta_1 + \frac{M+1}{2}\overline{T}\right) + P^{(PF)}\left(\delta_1 + \frac{M+1}{2}\overline{T} + \delta_2 + \frac{L+1}{2}W\right)$$

Acest τ include pe W ca unul din termeni. Dacă τ este substituit în expresia lui W se obtine o ecuatie care se rezolvă pentru W.

Rezolvarea pentru W produce rezultatul

$$W = \frac{1}{1-\frac{L+1}{2}P^{(PF)}}\left[\overline{T} + P^{(RB)}\left(\delta_1 + \frac{M+1}{2}\overline{T}\right) + P^{(PF)}\left(\delta_1 + \frac{M+1}{2}\overline{T} + \delta_2\right) + \frac{T_s}{M}\right]$$

Stabilirea valorii optime pentru M, care minimizează pe W, se face iterativ. Valoarea initială se obtine prin punerea valorii 1 pentru numitor si 0 pentru δ_2; se ia derivata în raport cu M si se anulează pentru a obtine o ecuatie în M.

Valoarea initială pentru *M* rezultată este aproximativ următoarea:

$$M_{opt} \approx \sqrt{\frac{2T_s}{\overline{T}(P^{(RB)} + P^{(PF)})}} = \sqrt{\frac{2T_s}{\overline{T}(1 - \sum_{i=1}^{N} f_i e^{-\lambda_i T_i})}}$$

Schema CARER (Cash-Aided Rollback Error Recovery)

Reducerea duratei checkpointării permite checkpointarea mai frecventă; se reduce si penalitatea la revenire după manifestarea disfunctiei si recuperarea din starea disfunctă.
Schema CARER reduce timpul cerut pentru a preleva un checkpoint prin marcarea urmei (*footprint*) procesului în memoria principală si în memoria cache ca parte a stării checkpointate. Se admite că memoria si memoria cache sunt candidate mult mai putin la disfuncii decât procesorul.
Checkpointarea constă în memorarea registrelor procesorului în memoria principală si includerea urmei proceselor în memoria principală plus în orice linie de cache marcată ca fiind parte din checkpoint.

Biti de checkpointare pentru fiecare linie de cache

Această tehnică necesită modificări de hardware: asocierea unui bit suplimentar de checkpointare asociat cu fiecare linie de cache.
Când acest bit este 1: linia corespunzătoare nu poate fi modificată, adică linia este parte din ultimul, cel mai recent checkpoint; nu poate fi actualizată fără a obliga la prelevarea imediată a unui checkpoint.
Dacă acel bit este 0: procesorul este liber să modifice cuvântul.
Urma procesului în memoria principală si liniile marcate din cache îndeplinesc dubla sarcină de memorie si de parte a checkpointului. Schema aceasta restrânge libertatea în a decide când trebuie prelevate checkpointurile.
Checkpointarea este fortată, obligatorie atunci când:

- O linie marcată ca nemodificabilă trebuie actualizată
- Ceva în memoria principală, nu importă ce, trebuie actualizat
- Se execută o instructiune I/O sau apare o întrerupere externă.

Prelevarea unui checkpoint implică salvarea registrelor procesorului în memorie si setarea la 1 a bitului de checkpoint asociat cu fiecare linie de cache validă.
Revenirea la checkpointul anterior este foarte simplă: se restaurează registrele si se marchează ca invalide toate liniile din cache cu bitul de checkpoint nul.
Costul implicat este după cum urmează:

- Un bit de checkpoint pentru fiecare linie de cache.
- Fiecare rescriere a unei linii de cache în memoria principală implică prelevarea unui checkpoint.

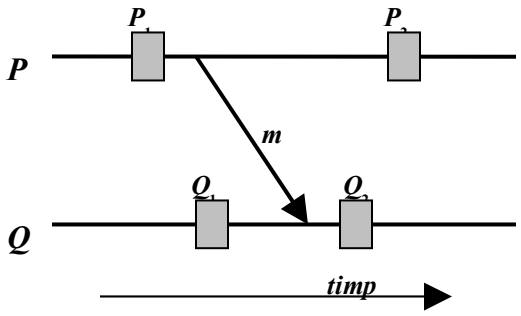
Checkpointarea în sistemele distribuite

Sistemele distribuite sunt caracterizate de faptul că procesoarele si memoriile asociate sunt conectate printr-o retea. Fiecare procesor poate avea discuri locale. Nu este exclusă existenta unui sistem de fisiere în retea accesibile tuturor procesoarelor.
Procesele conectate prin canale directionale: conexiuni punct-la-punct de la un proces la un altul. Canalele se presupun a fi libere de erori si programate pentru a livra mesajele în ordinea primirii.
Într-un sistem distribuit se petrec evenimente deteministe si non-deterministe.
Un eveniment non-determinist este un eveniment a cărui aparitie nu poate fi prezisă pe baza stării anterioare a sistemului. Dimpotrivă, un eveniment determinist poate fi prezis.
Executia procesului este o secventă de evenimente deterministe, întreruptă când si când de unele evenimente non-deterministe.
Exemplu: un program care controlează o supapă de presiune a unui reactor chimic este o buclă nesfârsită cu intrări furnizate cu o anumită frecventă de senzori de presiune; urmează de fiecare dată decizii de control. Valoarea unei intrări se constituie ca un eveniment non-determinist: ea nu poate fi prezisă pe baza rezultatelor observării anterioare.
O notiune cu care se lucrează uneori este aceea de proces determinist pe portiuni. Aceasta vrea să cuprindă o situatie reală în care odată intrarea cunoscută, restul este predicitibil (presupunând lipsa vreunei disfunctii).
O executie a unui proces poate fi privită ca deterministă pe portiuni. Se iau în considerare "bucăti" de timp. Fiecare din ele începe cu un eveniment non-determinist. Fiind date informatii asupra evenimentului non-determinist si asupra stării procesului la începutul intervalului de timp, se poate prezice fiecare eveniment care se produce pe durata intervalului.

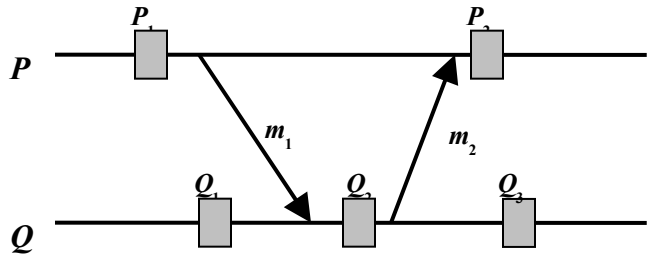
Proces/canal/starea sistemului

Starea unui proces are un sens evident: starea canalului la momentul t este descrisă prin multimea de mesaje transferate prin el până la momentul t (dar si ordinea receptionării lor).
Starea sistemului distribuit este agregatul de stări ale proceselor individuale si ale canalelor.
Starea se zice că este consistentă dacă pentru fiecare livrare de mesaj există un eveniment corespunzător de expediere a mesajului.
O stare care violează această conditie – un mesaj livrat care nu a fost încă trimis – violează cauzalitatea; un asemenea mesaj se numeste *orfan*.
Inversul este consistent: starea unui sistem reflectă trimiterea unui mesaj dar nu receptionarea lui.
Despre stări consistente si stări inconsistente, mai multe în discutia următoare.
Două procese P si Q au fiecare de prelevat checkpointuri. Mesajul m este trimis de P spre Q.

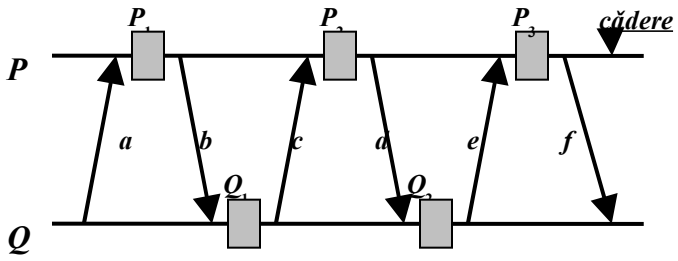
Iată multimile de checkpointuri care reprezintă stări consistente ale sistemului (v.figura):
$\{P_1, Q_1\}$ – nici un checkpoint nu are stire de mesajul m.
$\{P_2, Q_2\}$ – P_2 indică faptul că m a fost expediat; Q_2 indică receptionarea lui.
$\{P_2, Q_1\}$ – P_2 indică expedierea mesajului m; Q_1 nu are înregistrată receptia lui m.



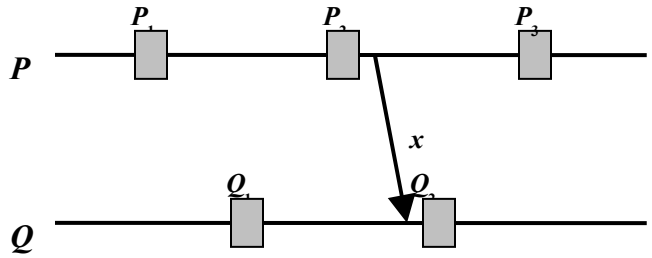
Dimpotrivă, multimea $\{P_1, Q_2\}$ reprezintă o stare inconsistentă; P_1 nu are înregistrată expedierea lui m, Q_2 stie de receptionarea lui m, ceea ce face din m un *mesaj orfan*.
Multimea de checkpointuri care reprezintă o stare consistentă a sistemului formează o linie de recuperare: sistemul poate fi făcut să revină (*roll-back*) prin ele si repornit de acolo.
$\{P_1, Q_1\}$: revenirea lui P la P_1 anulează trimiterea mesajului m si revenirea lui Q la Q_1 are semnificatia neprimirii lui m de către Q.
Repornirea din aceste checkpointuri are ca efect retrimiterea mesajului m de către P pe care-l va primi Q.
$\{P_2, Q_1\}$: revenirea lui P în P_2 nu include retransmiterea lui m; iar revenirea lui Q în Q_1 înseamnă că procesul Q nu are înregistrată deocamdată primirea lui m.
Procesul de recuperare trebuie să fie capabil a reproduce pe m pentru Q; acest fapt poate fi făcut prin adăugarea lui m la checkpointul lui P sau prin detinerea unui registru separat al mesajelor care să înregistreze tot ce primeste Q.
Uneori, checkpointurile pot fi inutile; ele nu vor face parte deloc din linia de recuperare astfel că prelevarea lor este o pierdere de timp.
Figura următoare arată ceea ce este un checkpoint inutil.
Q_2 este un checkpoint inutil. El înregistrează receptionarea lui m_1 dar nu expedierea lui m_2.
Multimea $\{P_1, Q_2\}$ nu poate fi consistentă (altminteri m_1 ar deveni un mesaj orfan). Similar $\{P_2, Q_2\}$ nu poate fi o multime de checkpointuri consistentă (deoarece altminteri m_2 ar deveni un mesaj orfan).



Dacă checkpointurile nu sunt coordonate (direct – prin transmitere de mesaje sau indirect – prin ceasuri sincronizate), o disfunctie singulară poate produce un *efect de domino*. Figura următoare ilustrează o asemenea situatie.

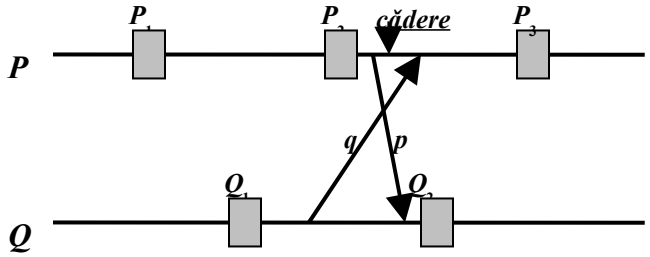


Când P suferă o disfunctie tranzitorie, el revine la checkpointul P_3. Deoarece mesajul f a fost trimis după preluarea checkpointului P_3, Q trebuie să revină (*roll-back*) si el (altminteri Q ar avea un mesaj care n-a fost trimis niciodată, un mesaj orfan). P va reveni la P_2 deoarece Q a trimis un mesaj e către P.
Această revenire pas cu pas continuă până când (toate) procesele au revenit la pozitiile lor de plecare.
Uneori, unele mesaje se pot pierde.



Mesajele se pot pierde datorită revenirii (*roll-back*).

Se presupune, de pildă, că procesul Q revine la Q_1 după primirea mesajului x de la P. Înregistrarea primirii lui x este pierdută.
Dacă P nu revine la P_2, este ca si când P a trimis un mesaj care n-a fost primit vreodată de Q.
Mesajele pierdute astfel nu violează cauzalitatea. Este ceva similar cu pierderea de mesaje din cauza unor probleme în retea si se rezolvă prin retransmitere.
Totusi, dacă Q trimite un ACK (*acknoledgement*) despre x la P înainte de revenire, atunci acel ACK va fi un mesaj care devine orfan cu exceptia cazului când P revine la P_2.
Este posibil si un blocaj, blocaj care se numeste *livelock*[9]. Livelock-ul este o altă problemă care poate apărea în sistemele distribuite checkpointate. Iată un exemplu:
Q trimite lui P un mesaj q, P trimite lui Q un mesaj p. P esuează înainte de primirea mesajului q. Pentru a preveni situatia în care p ar deveni orfan, Q trebuie să revină în Q_1.



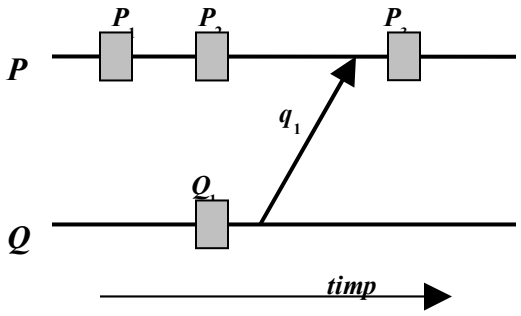
Între timp, P se recuperează, revine la P_2, trimite o copie nouă a lui p si apoi primeste copia lui q care a fost trimisă înainte ca orice revenire să înceapă.
Totusi, deoarece Q a revenit, copia lui q este acum orfană si astfel P trebuie să repete revenirea sa. La rându-i, face orfană copia secundă a lui p, fortând astfel pe Q să repete si el revenirea sa.
Acest dans de reveniri poate continua indefinit.

[9] *Livelock* – o buclă nesfârsită în executia unui program. Ea apare când un proces se repetă pe sine din cauză că el continuă să primească informatii eronate. Poate apărea si atunci când un proces care apelează alt proces este el însusi apelat de acel proces si nu există o logică detectoare a acestei situatii în vederea opririi ei. Un "livelock" diferă de un "deadlock" când procesul continuă să aibă loc si nu-i o asteptare într-o buclă fără efecte. Este deosebit si de *deadly embrace* care este un punct mort manifestat prin asteptarea reciprocă a răspunsului de către două elemente de program. De pildă, într-o retea, un utilizator lucrează cu fisierul A si necesită fisierul B pentru a continua, iar un alt utilizator lucrează cu fisierul B si are nevoie de A pentru a continua; fiecare îl asteaptă pe celălalt, amândoi sunt temporar blocati.

Algoritmi de checkpointare coordonati

Cum s-a văzut, checkpointarea necoordonată poate duce la efectul de domino sau la buclare (*livelock*). Alternativa este checkpointarea coordonată.
Sunt două moduri de bază pentru coordonarea checkpointării:

- Algoritmul Koo-Toueg care are un initiator al procesului de checkpointare pe întreg sistemul.



- Un algoritm care esalonează în timp checkpointuri: esalonarea checkpointurilor poate evita încărcarea excesivă cvasisimultană a sistemului de discuri.

Comunicarea generează proceduri de checkpointare induse.
O solutie mixtă cuprinde utilizarea simultană a algoritmilor de checkpointare coordonati si necoordonati, cei din urmă pentru a se ocupa de disfunctiile cele mai izolate.

Algoritmul Koo-Toueg

Se admite că P doreste să stabilească un checkpoint la P_3. Acesta va înregistra q_1 primit de la Q. Pentru a preveni orfanizarea lui q_1, Q trebuie să checkpointeze si el. Astfel, stabilirea de către P a unui checkpoint la P_3 obligă pe Q să preia un checkpoint care să înregistreze că mesajul q_1 a fost trimis.
Un algoritm pentru astfel de checkpointare coordonată cunoaste două tipuri de checkpointuri: checkpointuri tentative si checkpointuri permanente.
P înregistrează mai întâi starea lui curentă într-un checkpoint tentativă, apoi trimite un mesaj tuturor celorlalte procese de la care a primit un mesaj de când a prelevat ultimul său checkpoint. Se notează multimea acestor procese cu Π.
Mesajul înstiintează fiecare proces din Π (de pildă pe Q) de ultimul mesaj, m_{qp}, pe care P l-a primit de la el înainte ca checkpointul tentativă să fi fost prelevat.
Dacă m_{pq} nu a fost înregistrat într-un checkpoint de Q, pentru a preveni m_{pq} de a deveni orfan, Q este pus să preia un checkpoint tentativă pentru a înregistra expedierea lui m_{pq}.

Dacă toate procesele din multimea Π, care sunt necesare, confirmă preluarea unui checkpoint cum se cere, atunci toate checkpointurile tentativă pot fi convertite în permanente.
Dacă unele elemente ale lui Π sunt incapabile de un checkpoint cum se cere, *P* si toate elementele din Π abandonează checkpointurile tentativă si niciunul nu este transformat în permanent.
Aceasta ar putea dezamorsa (*set off*) o reactie în lant de checkpointuri.
Fiecare element al lui Π poate depune un set de checkpointuri între procese în propria sa multime corespunzătoare.

Checkpointarea esalonată (*staggered*)

Algoritmul Koo-Toueg si altele asemenea pot conduce la un număr mare de procese care prelevă checkpointuri aproape simultan. Dacă ele scriu toate într-o memorie stabilă partajată, de pildă un set de discuri comune, transferul poate conduce la congestionări ale accesului la disc sau ale retelei sau concomitent ale amândurora.
Oricare din cele două tratări poate fi utilizată pentru a ne asigura că în orice moment cel mult un proces îsi preia checkpointul său.
(1) Se scrie checkpointul într-un buffer local, apoi se trec esalonat (*stagger*) scrierile din buffer în memoria stabilă; se presupune că bufferul are o capacitate suficientă.
(2) Se încearcă esalonarea checkpointurilor în timp.
Checkpointurile esalonate ar putea deveni inconsistente; este posibil a exista mesaje orfane în sistem.
Acest fenomen poate fi evitat printr-o fază coordonată în care fiecare proces înregistrează în memoria stabilă toate mesajele pe care le trimite urmând checkpointului precedent. Faza de înregistrare a mesajelor lansate de procese se va suprapune (*overlap*) în timp.
Dacă volumul de mesaje este mai redus decât dimensiunea checkpointurilor individuale, discurile si reteaua vor cunoaste un flux redus.

Recuperarea din starea de disfunctie

Dacă un proces clachează, el poate fi repornit după revenire la ultimul checkpoint si toate mesajele stocate în registru sunt citite.
Această combinatie de checkpoint cu registru de mesaje este denumită *checkpoint logic*.
Acest algoritm de checkpointare esalonată garantează că toate checkpointurile logice formează o linie de recuperare consistentă.
Algoritmul esalonat, *faza primă*, faza checkpointării:
pentru ($i = 0$; $i \leq n - 1$; i++) {
    P_i preia un checkpoint

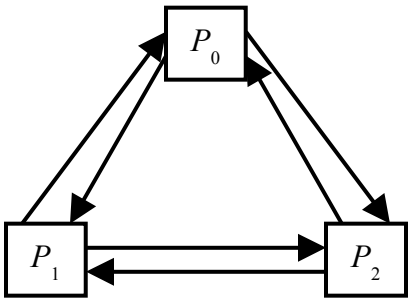
P_i trimite un mesaj la $P_{\{(i+1) \bmod n\}}$ cu ordinul ca acesta din urmă să preia un checkpoint
}

Dacă P_0 primeste un mesaj de la $P_{\{n-1\}}$ care îi ordonă să preia un checkpoint, acesta este pretextul pentru P_0 de a initia faza a doua (de înregistrare de mesaje). El trimite un mesaj marker pe fiecare dintre canalele sale de plecare. Când procesul P_i receptionează un mesaj marker, el trece la faza a doua.

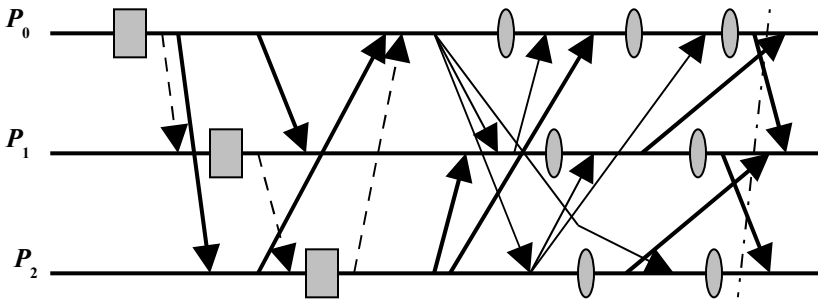
Faza a doua a algoritmului esalonat, faza înregistrării de mesaje:

if (nici un mesaj marker anterior nu a fost primit în această rundă de P_i)
then {
P_i trimite un mesaj marker pe fiecare din canalele sale de plecare
P_i înregistrează toate mesajele primite de el după checkpointul precedent
}
else
P_i actualizează registrul său de mesaje prin adăugarea tuturor mesajelor primite de el de la ultima actualizare a registrului
end if.

Exemplu de algoritm esalonat – faza 1:



Sistem cu trei procese.



Legenda figurii:

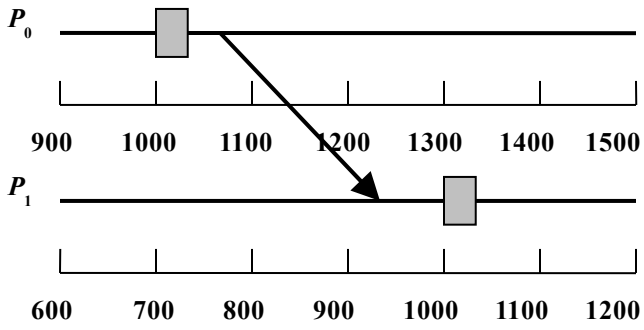
Dreptunghi: prelevare de checkpoint
Elipsă: înregistrare de mesaje
Linie continuă îngrosată: mesaje schimbate de procese
Linie continuă simplă: ordin de înregistrare de mesaje
Linie întreruptă: ordin de prelevare de checkpoint (take_checkpoint)
Linie din linii si puncte: pozitie de checkpoint logic
Mesajele schimbate de procese sunt notate m_0, m_1, m_2 etc. în ordinea temporală a generării lor.
P_0 prelevă un checkpoint si trimite mesajul si ordinul take_checkpoint la P_1.
P_1 trimite un ordin la fel către P_2 după ce prelevă propriul său checkpoint.
P_2 trimite un mesaj imperativ take_checkpoint înapoi la P_0.
La acest punct, fiecare proces a prelevat un checkpoint si faza a doua poate începe.
Faza a doua pentru acelasi exemplu:
P_0 trimite ordinul de înregistrare de mesaje, message_log la P_1 si P_2 care înregistrează mesajele pe care le-au primit după cel mai recent checkpoint.
P_1 si P_2 trimit mesaje similare cu ordinul message_log.
De fiecare dată când se primeste un astfel de mesaj, procesul înregistrează mesajele. Dacă este pentru prima oară când procesul primeste un astfel de ordin message_log, procesul trimite mesaje marker pe fiecare din canalele sale emergente.

Recuperarea

Ipoteză: fiind date checkpointul si mesajele primite, un proces se poate recupera. Putem avea mesaje orfane în raport cu checkpointurile fizice prelevate în prima fază. Nu vor exista mesaje orfane în raport cu ultimele (temporal) checkpointuri logice care sunt generate utilizând checkpointurile fizice si registrul de mesaje.

Sincronizarea temporală

Mesajele orfane pot să nu apară dacă fiecare proces checkpointează exact în aceleasi momente. Practic acest lucru este imposibil: deviatiile de ceas si timpii consumati de comunicarea mesajelor nu se pot reduce la zero.
Sincronizarea temporală poate fi încă utilizată pentru a facilita checkpointarea: trebuie însă să luăm în calcul diferentele nenule pe ceasuri.
Sincronizarea temporală (time-based): procesele sunt checkpointate la momente agreate în prealabil.
Exemplu: se indică fiecărui proces a checkpointa când ceasul local indică un multiplu de 100 de secunde.
O astfel de procedură nu este suficientă în sine pentru evitarea aparitiei mesajelor orfane. Iată un exemplu de creare a unui mesaj orfan (v.figura).



Fiecare proces prelevă un checkpoint la timpul 1000 (ceasul local). Neconcordanta dintre cele două ceasuri este astfel că procesul P_0 preia chckpointul său mult mai devreme (în timp real universal) decât P_1. Ca rezultat, P_0 trimite un mesaj către P_1 după checkpointul său, mesaj primit de P_1 înainte de a checkpointa el însusi. Acest mesaj este un potential orfan.

Cum se poate preveni crearea unui mesaj orfan?

Să presupunem că abaterea de temporalitate între oricare două ceasuri ale sistemului este mărginită de δ si fiecare proces este programat să checkpointeze când ceasul lui local indică timpul τ.

Urmând checkpointului său, un proces P_x ar putea să nu trimită mesaje la vreun alt proces P_y până când nu este sigur că ceasul local al lui P_y arată τ. P_x poate să rămână "tăcut" pe durata $[\tau, \tau + \delta]$ cu timpii măsurati pe ceasul lui local.

Dacă timpul de livrare al mesajelor între procese are o limită inferioară β, pentru a preveni mesajele orfane, procesul P_x trebuie să rămână tăcut un interval mai scurt $[\tau, \tau + \delta - \beta]$. Dacă $\beta > \delta$, acest interval este de lungime nulă si necesitatea ca P_x să păstreze tăcerea dispare.

Alte metode de prevenire, diferite sunt discutate imediat.

Să presupunem că mesajul m este primit de procesul P_y când ceasul său indică momentul t. Mesajul m trebuie să fi fost trimis (de P_x) nu mai târziu de β de mai sus, înainte ca ceasul lui P_y să indice $t - \beta$.

Deoarece desincronizarea ceasurilor este de cel mult δ, la acest moment, ceasul lui P_x ar fi trebuit să indice cel mult $t - \beta + \delta$.

Dacă $t - \beta + \delta < \tau$, trimiterea lui m va fi înregistrată în checkpointul lui P_x si m nu poate fi orfan.

Un mesaj m receptionat de P_y când ceasul său indică cel putin $\tau - \delta + \beta$ nu poate fi orfan.

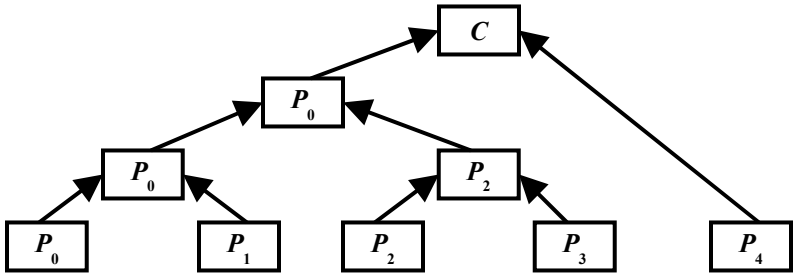
Mesajele orfane pot fi evitate de P_y prin neutilizarea si neincluderea în checkpointul său la momentul τ a nici unui mesaj primit în intervalul $[\tau - \delta + \beta, \tau]$ (după ceasul P_y) până după prelevarea checkpointului său de la momentul τ.

Checkpointarea fără disc

Memoria principală este volatilă si de aceea nepotrivită pentru depozitarea unui checkpoint; totusi, cu procesoare suplimentare se poate face checkpointarea în memoria principală. De ce? Pentru că prin evitarea scrierii pe discuri, checkpointarea poate fi mai rapidă.
Checkpointarea fără discuri este cel mai bine utilizată ca un nivel într-o checkpointare pe două niveluri. În cazul utilizării tehnicilor de tipul RAID, pentru tratarea disfunctiilor există procesoare redundante.
Exemplu: un sistem distribuit cu cinci procesoare care lucrează si unul suplimentar. Fiecare procesor lucrător detine checkpointul său depus în memoria proprie; procesorul suplimentar stochează informatia paritară a acestor checkpointuri. Dacă un proces lucrător devine disfunct checkpointul său poate fi reconstituit din cele patru rămase plus checkpointul paritar.

Checkpointare fără discuri în stilul RAID

Reteaua care legă procesoarele trebuie să aibă o bandă de trecere suficientă pentru expedierea checkpointurilor.
Exemplu: cazul cu *n* procesoare executante si un procesor consacrat checkpointării. Dacă toate procesoarele executante trimit checkpointurile lor la procesorul dedicat checkpointării pentru calculul parității, acesta poate deveni un *hotspot*, un loc de strangulare.
Solutia constă în distribuirea calculelor de paritate.



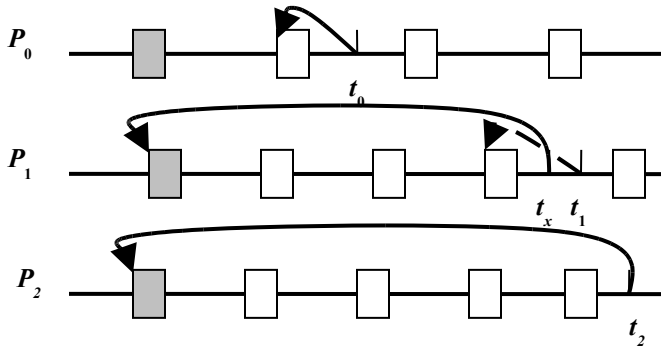
Recuperarea se poate face pe două niveluri. Coordonarea checkpointurilor previne aparitia mesajelor orfane dar impune un overhead.

- Nu afectează corectitudinea dacă disfunctiile sunt izolate, adică la orice moment cel mult un procesor este în starea de disfunctie (în recuperare).
- Vasta majoritate a disfunctiilor sunt izolate.
- Aceasta face recuperarea din disfunctii izolate rapidă.
- Se acceptă durate mai mari de recuperare pentru disfunctii simultane.

Aceasta sugerează o schemă de recuperare pe două niveluri.

- Primul nivel: fiecare proces prelevă propriul checkpoint fără coordonare (util, potrivit numai pentru recuperarea din disfunctie izolată).
- Checkpointul nu trebuie scris pe disc, poate fi scris în memoria altui procesor.
- Nivelul al doilea: ocazional, întregul sistem face o checkpointare coordonată (cu overhead mai mare) care protejează la disfunctii care nu sunt izolate.

Un exemplu de recuperare pe două niveluri: P_0 cade la momentul t_0; sistemul revine la ultimul checkpoint de prim nivel; recuperarea este cu sigurantă reusită.



Dreptunghiuri albe: checkpointuri de primul nivel
Dreptunghiuri umbrite: checkpointuri de nivelul al doilea

P_1 cade la momentul t_1; initiază revenirea; în punctul t_x (pe durata recuperării), esuează si P_2. Disfunctii care nu mai sunt izolate: sistemul revine în ambele procese la ultimul checkpoint de nivelul secund.
În general, cu cât sunt mai dese disfunctiile neizolate, corelate, cu atât mai mare trebuie să fie frecventa de prelevare a checkpointurilor de nivelul doi.

Înregistrarea mesajelor

Pentru a continua calculul dincolo de ultimul checkpoint, procesul de recuperare poate cere toate mesajele primite până atunci, citite în ordinea originară.
Pentru checkpointarea coordonată, fiecare proces poate fi făcut să revină la ultimul lui checkpoint si repornit: aceste mesaje vor fi reexpediate în timpul executiei repetate.
Pentru a evita overheadul de coordonare si a lăsa checkpointurile proceselor independente, o optiune o reprezintă înregistrarea mesajelor.
Sunt folosite două modalităti de înregistrare:

- Înregistrarea pesimistă: dă siguranta că revenirea nu va difuza, adică dacă un proces esuează, nici un alt proces nu va trebui să facă revenirea pentru a asigura consistenta.

- Înregistrarea optimistă: esecul unui proces poate declansa si revenirea altor procese.

Înregistrarea pesimistă a mesajelor reprezintă metoda cea mai simplă: primitorul unui mesaj opreste orice activitate când primeste acel mesaj, înregistrează mesajul într-o memorie stabilă, apoi reia executia.
Recuperarea procesului din starea de esec: procesul este făcut să revină la ultimul său checkpoint si îsi reciteste mesajele pe care le-a primit de la acel checkpoint, în ordinea corectă. Nu va exista nici un mesaj orfan, fiecare mesaj va fi ori primit înainte de ultimul checkpoint, ori salvat explicit în registrul de mesaje. Revenirea unui proces nu va declansa revenirea altor procese.
Înregistrarea mesajelor după expeditor tine seamă de faptul că înregistrarea mesajelor într-o memorie stabilă poate aduce un overhead semnificativ.
Înregistrarea mesajelor pe baza expeditorului este potrivită vis-à-vis de căderile izolate. Fazele acestui gen de înregistrare sunt următoarele:

- Expeditorul unui mesaj îl înregistrează într-un registru si, la cerere, registrul este citit pentru a reproduce mesajul.
- Fiecare proces are contoare de trimiteri si de primiri care sunt incrementate de fiecare dată când procesul trimite sau primeste un mesaj.
- Fiecare mesaj are un SSN (*Send Sequence Number*), valoarea contorului de trimiteri la momentul transmiterii.
- Unui mesaj primit i se alocă un RSN (*Receive Sequence Number*), valoarea contorului de primiri la momentul primirii.
- Primitorul trimite totodată un ACK (*acknowledge*) expeditorului, care include RSN-ul pe care l-a alocat mesajului.
- La primirea acestui ACK, expeditorul confirmă ACK-ul într-un measaj către primitor.
- În timpul cât receptorul primeste mesajul si trimite acel ACK si cât primeste ACK-ul expeditorului despre ACK-ul propriu, receptorului îi este interzisă expedierea de mesaje către alte procese, ceea ce este esential pentru functionarea corectă la recuperare.
- Un mesaj se numeste deplin-înregistrat dacă nodul expeditor stie atât SSN-ul cât si RSN-ul lui. El este partial-înregistrat dacă nodul expeditor nu stie încă RSN-ul lui.
- Când un proces revine si reporneste calculul de la ultimul checkpoint, el trimite celorlalte procese un mesaj care listează SSN-ul ultimului lui mesaj pe care l-a înregistrat în checkpointul său.
- Când acest mesaj este primit de un proces, acel proces stie care mesaje trebuie retransmise si se conformează.
- Procesul de recuperare trebuie acum să utilizeze aceste mesaje în aceeasi ordine în care ele au fost utilizate înainte de cădere; este usor de făcut aceasta pentru mesajele deplin-înregistrate deoarece RSN-urile lor sunt cunoscute si pot fi sortate după aceste numere.

Care este soarta mesajelor partial-înregistrate? Mesajele partial-înregistrate sunt cu RSN-uri necunoscute. Ele sunt trimise, dar ACK-urile lor nu vor fi primite

niciodată de expeditor. Receptorul a esuat înainte ca mesajul să-i fie livrat sau a esuat după receptia mesajului dar înainte de a trimite ACK-ul. Receptorul este oprit a trimite mesaje proprii către alte procese între primirea mesajului si trimiterea ACK-ului asociat. Ca rezultat, receptia mesajului partial-înregistrat într-o ordine diferită a doua oară nu poate afecta nici un alt proces din sistem, corectitudinea este prezervată. Este limpede, această metodă este garantată a lucra numai dacă la un moment dat, altfel oarecare, există cel mult un nod căzut.
Înregistrarea de mesaje optimistă are overhead mai redus decât înregistrarea pesimistă, dar recuperarea din starea de disfunctie este mult mai complexă.
Înregistrarea optimistă este mai curând de interes teoretic.
Când mesajele sunt primite, ele sunt scrise într-un buffer volatil care la momentul potrivit este copiat în memoria stabilă. Derularea procesului nu este întreruptă si astfel overheadul înregistrării este redus.
La cădere, continutul bufferului poate fi pierdut conducând la necesitatea ca mai multe procese să revină (*multiple roll-back*). Este necesară o schemă specială pentru a rezolva această situatie.

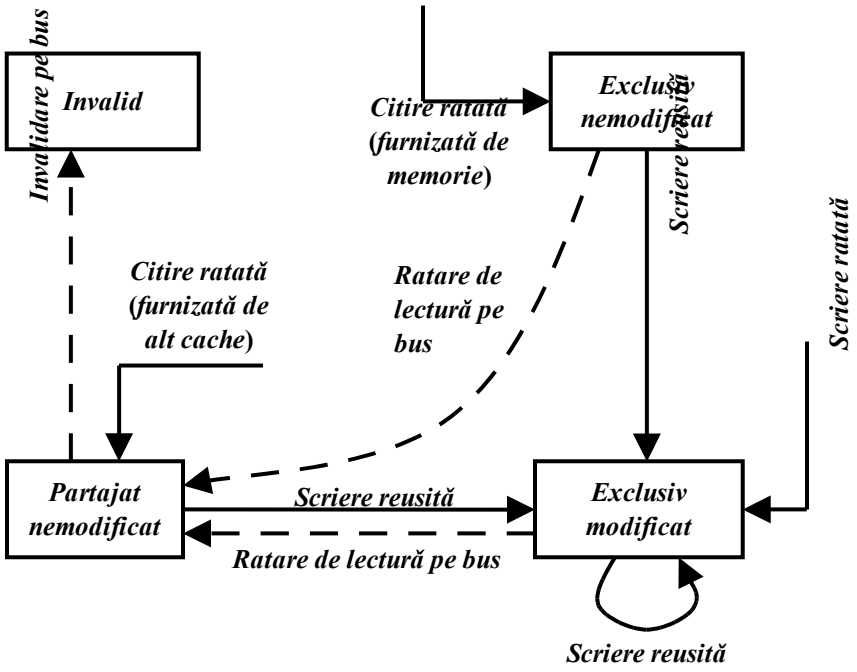
Checkpointarea în sistemele cu memorie partajată

Într-o variantă de CARER (*Cash-Aided Roll-back Error Recovery*) pentru sisteme multiprocesor pe bus cu memorie partajată, fiecare procesor are propriul său cache.
Este necesară o schimbare a algoritmului pentru a mentine coerenta de cache între memorii cash multiple. În locul unui singur bit care să marcheze o linie ca imuabilă, avem un identificator multi-bit: un identificator de checkpoint *C_id* pe fiecare linie, un contor de checkpointuri pe fiecare procesor, *C_count*, care retine numărul checkpointului curent. La fiecare prelevare de checkpoint, se incrementează contorul.
O linie modificată înainte va avea *C_id*-ul ei inferior contorului. Când o linie este actualizată, se pune *C_id* = *C_count*.
Dacă o linie a fost modificată de când a fost adusă în cache si *C_id* < *C_count*, linia este parte din starea checkpointului si este prin urmare inaccesibilă scrisului. Orice scrieri într-o astfel de linie trebuie să astepte până când linia este scrisă mai întâi în memoria principală.
Dacă contorul are *k* biti, el revine la 0 după atingerea valorii 2^{k-1}.

Un protocol de coerentă bus-based

Modifică un algoritm de coerentă a cache-ului pentru a tine cont de checkpointare. Întregul trafic între cache-uri si memorie trebuie să uzeze de bus, altfel spus, toate cache-urile pot urmări traficul pe bus.
O linie de cache poate fi în una din următoarele stări: invalidă, nemodificată si partajată, modificată si exclusivă si nemodificată si exclusivă.
Exclusivă: aceasta este singura copie validă în vreun cache.

Modificată: linia a fost modificată de când a fost adusă din memorie în cache.



Dacă un procesor are de actualizat o linie aflată în starea nemodificată partajată, ea se schimbă în starea modificată exclusivă. Alte cache-uri care detin aceeasi linie trebuie să invalideze cópiile lor care nu mai sunt actuale.
Când e vorba de stări exclusiv modificate sau exclusiv nemodificate si un alt cache pune pe bus o cerere de citire, cache-ul solicitant trebuie să facă service-ul acestei cereri (numai copia curentă a acelei linii).
Produs secundar: memoria este si ea actualizată dacă este necesar. Apoi se schimbă la starea partajată nemodificată.
Ratarea de scriere a unei linii în cache conduce la starea modificată exclusivă.
Coerenta bus-based si protocolul de checkpointare nu sunt lipsite de posibile contradictii. Cum se poate modifica protocolul pentru a tine seamă de checkpointare?
Starea originară exclusiv modificată se divide acum în două:

- Exclusivă modificată
- Inaccesibilă pentru scris (*unwritable*).

Protocolul directory-based

În această tratare este mentinut ca piesă centrală un director care înregistrează starea fiecărei linii. Acest director poate fi privit ca unul controlat de un anume *controller* al memoriei partajate. Controllerul manipulează toate ratările la citire si la scriere si toate celelalte operatii care schimbă starea liniei.

Exemplu: dacă o linie este în starea exclusivă-nemodificată si cache-ul care detine acea linie vrea s-o modifice, el notifică controllerul de intentia sa. Controllerul poate schimba starea la exclusivă-modificată. A implementa această schemă de checkpointare pe un astfel de protocol este atunci o problemă simplă.

Alte utilizări ale checkpointării

(1) Migrarea proceselor
Un checkpoint reprezintă starea procesului; migrarea procesului de la un procesor la un altul înseamnă deplasarea checkpointului si calculul poate fi reluat pe acel nou procesor; permite recuperarea din erori permanente sau intermitente.
Natura checkpointului determină dacă noul procesor trebuie să fie de acelasi model si trebuie să ruleze acelasi sistem de operare.
(2) Echilibrarea încărcării
O utilizare mai bună a unui sistem distribuit prin asigurarea că încărcarea cu calcule este distribuită potrivit între procesoare.
(3) Depanare (*debugging*)
Fisiere de miez (*core files*) sunt basculate (*dumped*) când un program se încheie anormal; acestea sunt în esentă checkpointuri care contin întreaga informatie de stare despre procesul afectat; depanatorii (programele depanatoare) pot citi fisierele *core* care pot ajuta la depanarea procesului.
(4) Instantanee
Observarea stării programului la momente discrete: întelegere mai adâncă a comportării programului.

SISTEME DE DISCURI TOLERANTE LA DEFECTE

Pentru a spori siguranta în functionare, sistemele de memorii pot fi structurate în asa manieră încât prin utilizarea unor redundante ele să manifeste o anumită tolerantă la defecte. În alti termeni, prin măsuri prealabile, anumite subsisteme pot suplini alte subsisteme care dintr-un motiv sau altul se defectează. Iată în continuare un exemplu semnificativ în ceea ce priveste toleranta la defecte a sistemelor de memorare, în particular cele pe disc.

Memorii ieftine exploatate în conditii de sigurantă

Fie n dispozitive de memorare, D_1, D_2, …, D_n. Fiecare dintre ele contine k octeti si sunt dispozitive de stocat date. Fie alte m dispozitive de memorare C_1, C_2, …, C_m. Si acestea contin fiecare tot câte k octeti si sunt denumite dispozitive de verificare. Continutul fiecărui dispozitiv de verificare se calculează din continutul dispozitivelor de date. Problema este a calcula continutul dispozitivelor C_i în asa mod încât oricare m dispozitive din D_1, D_2, …, D_n, C_1, C_2, …, C_m s-ar defecta, continutul dispozitivelor defecte să poată fi reconstituit din continutul dispozitivelor rămase în functie.

Strategia generală

Formal, modelul defectului este acela al unei pierderi de informatie prin stergere. Dacă un dispozitiv se defectează el iese din joc si sistemul recunoaste această situatie de inutilitate a acelui dispozitiv. Pierderea aceasta diferă de aparitia unor erori, caz în care defectarea se manifestă prin stocarea si restituirea unor valori incorecte care pot fi recunoscute printr-un anumit gen de codare intrinsecă.
Calculul continutului fiecărui dispozitiv de verificare C_i necesită o functie F_i aplicată pe continutul tuturor dispozitivelor de date. Formulele următoare sunt un exemplu pentru $n = 8$ si $m = 2$. Continutul dispozitivelor de verificare C_1 si C_2 se obtine prin evaluarea functiilor F_1, respectiv F_2.

$$C_1 = F_1(D_1, D_2, D_3, D_4, D_5, D_6, D_7, D_8)$$

$$C_2 = F_2(D_1, D_2, D_3, D_4, D_5, D_6, D_7, D_8)$$

Metoda de codare RS-RAID (RS – de la Reed-Solomon, RAID – de la Redundant Arrays of Inexpensive/Independent Disks) divide fiecare dispozitiv de memorare în cuvinte. Fiecare cuvânt este alcătuit din w biti, număr ales de programator, dar raportat la anumite restrictii. Asadar, fiecare dispozitiv contine

$$l = (k \ octeti)\left(\frac{8 \ biti}{octet}\right)\left(\frac{1 \ cuvânt}{w \ biti}\right) = \frac{8k}{w} \ cuvinte$$

Functiile de codare F_i operează pe cuvinte cu rezultatul tot în cuvinte, ca în relatiile următoare unde $x_{i,j}$ reprezintă cuvântul j stocat pe dispozitivul de memorare X_i.

D_1	D_2	C_1	C_2
$d_{1,1}$	$d_{2,1}$	$c_{1,1} = F_1(d_{1,1}, d_{2,1})$	$c_{2,1} = F_2(d_{1,1}, d_{2,1})$
$d_{1,2}$	$d_{2,2}$	$c_{1,2} = F_1(d_{1,2}, d_{2,2})$	$c_{2,2} = F_2(d_{1,2}, d_{2,2})$
$d_{1,3}$	$d_{2,3}$	$c_{1,3} = F_1(d_{1,3}, d_{2,3})$	$c_{2,3} = F_2(d_{1,3}, d_{2,3})$
.	.	.	.
$d_{1,i}$	$d_{2,i}$	$c_{1,i} = F_1(d_{1,i}, d_{2,i})$	$c_{2,i} = F_2(d_{1,i}, d_{2,i})$

Pentru a avea notatii mai simple, cu un indice mai putin, se admite că fiecare dispozitiv retine un cuvânt si numai unul. Pe calea aceasta problema se reduce la n cuvinte-date, $d_1, d_2, \ldots, d_n$ si la m cuvinte de verificare $c_1, c_2, \ldots, c_m$ calculate din cuvintele-date în asa mod încât să fie tolerată pierderea oricăror m cuvinte din cele $n + m$. Pentru calculul unui cuvânt de verificare c_i depus în dispozitivul C_i se aplică functia F_i cuvintelor-date

$$c_i = F_i(d_1, d_2, ..., d_n)$$

Dacă un cuvânt-dată din dispozitivul D_j este actualizat de la d_j la d_j' atunci fiecare din cuvintele de verificare c_i trebuie recalculat prin utilizarea unei functii $G_{i,j}$ astfel încât

$$c_i' = G_{i,j}(d_j, d_j', c_i)$$

Când m dispozitive de memorare clachează se reconstruieste sistemul după cum urmează. Mai întâi, pentru fiecare dispozitiv defect D_j se construieste o schemă care să recupereze continutul lui D_j din cuvintele depuse în dispozitivele functionale. Când operatia aceasta este încheiată se reconstituie continutul unor eventuale dispozitive de verificare disfuncte C_i, cu ajutorul functiilor F_i.

De exemplu, presupunând că $m = 1$, paritatea $n + 1$ se poate descrie în termenii generali de mai sus. Există numai un dispozitiv de verificare C_1 si lungimea cuvântului este de 1 bit ($w = 1$). Pentru calculul cuvâtului de verificare c_1 se ia paritatea prin SAU EXCLUSIV (XOR) a cuvintelor de date

$$c_1 = F_1(d_1, d_2, ..., d_n) = d_1 \oplus d_2 \oplus ... \oplus d_n$$

Dacă cuvântul de pe suportul D_j se schimbă din d_j în d_j' atunci c_1 se recalculează din paritatea vechiului cuvânt si din cele două cuvinte-date

$$c_1' = G_{1,j}(d_j, d_j', c_1) = c_1 \oplus d_j \oplus d_j'$$

Dacă un dispozitiv se defectează atunci fiecare cuvânt poate fi reconstituit prin paritatea cuvintelor de pe dispozitivele rămase în functie

$$d_j = d_1 \oplus ... \oplus d_{j-1} \oplus d_{j+1} \oplus ... \oplus d_n \oplus c_1$$

Sistemul este rezistent la defectarea oricărui (unic) suport.

O reformulare a problemei sună astfel: sunt date n date d_1, d_2, …, d_n, toate de dimensiunea w. Se definesc functiile F si G care sunt utilizate pentru a calcula si pentru a întretine, a mentine actuale cuvintele de verificare c_1, c_2, …, c_m. Se face o descriere a modului cum se reconstituie cuvintele de pe orice suport esuat când numărul de dispozitive de memorare defecte nu depăseste m. Cu cuvintele-date reconstituite se recalculează cuvintele de verificare din cuvintele-date cu ajutorul functiilor F. Sistemul este refăcut în întregime.

Algoritmul RS-RAID

Trei aspecte sunt deosebite în aplicarea algoritmului. Primul constă în utilizarea matricilor Vandermonde (Alexandre-Théophile Vandermonde, 1735-1796) pentru calculul si mentinerea cuvintelor de control, al doilea este utilizarea eliminării Gauss pentru recuperarea din starea de nefunctionare si al treilea, utilizarea aritmeticii specifice câmpurilor Galois. Acestea toate sunt detaliate în continuare.

Calculul si întretinerea cuvintelor de verificare

Functiile F_i sunt prin definitie combinatii liniare ale cuvintelor-date

$$c_i = F_i(d_1, d_2, ..., d_n) = \sum_{j=1}^{n} f_{i,j} d_j$$

Cu alte cuvinte, dacă se adoptă o reprezentare matricială cu D si C vectori si F_i linii într-o matrice F

$$FD = C$$

Matricea F este definită ca o matrice Vandermonde $m \times n$ cu $f_{i,j} = j^{i-1}$, ceea ce face din relatia de mai sus

$$\begin{bmatrix} f_{1,1} & f_{1,2} & \cdots & f_{1,n} \\ f_{2,1} & f_{2,2} & \cdots & f_{2,n} \\ \vdots & \vdots & & \vdots \\ f_{m,1} & f_{m,2} & \cdots & f_{m,n} \end{bmatrix} \begin{bmatrix} d_1 \\ d_2 \\ \vdots \\ d_n \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & 2 & 3 & \cdots & n \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & 2^{m-1} & 3^{m-1} & \cdots & n^{m-1} \end{bmatrix} \begin{bmatrix} d_1 \\ d_2 \\ \vdots \\ d_n \end{bmatrix} = \begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_m \end{bmatrix}$$

Când unul din cuvintele-date d_j se schimbă în d_j' cuvintele de verificare trebuie schimbate în consecintă. Prin scăderea portiunii din cuvântul de verificare care corespunde lui d_j si adunarea cantitătii necesare pentru d_j' se obtine pentru $G_{i,j}$ definitia din relatia de mai jos

$$c_i' = G_{i,j}(d_j, d_j', c_i) = c_i + f_{i,j}(d_j' - d_j)$$

Asadar, calcularea si întretinerea cuvintelor de verificare pot fi făcute printr-o aritmetică simplă, dar după regulile date mai departe.

Recuperarea din *crash*

Pentru a explica recuperarea aceasta se definesc matricea $A = \begin{bmatrix} I \\ F \end{bmatrix}$ si vectorul $E = \begin{bmatrix} D \\ C \end{bmatrix}$. Apoi se scrie ecuatia

$$AD = E$$

care în formă detaliată apare ca

$$\begin{bmatrix} 1 & 0 & 0 & \cdots & 0 \\ 0 & 1 & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & & \vdots \\ 0 & 0 & 0 & \cdots & 1 \\ 1 & 1 & 1 & \cdots & 1 \\ 1 & 2 & 3 & \cdots & n \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & 2^{m-1} & 3^{m-1} & \cdots & n^{m-1} \end{bmatrix} \begin{bmatrix} d_1 \\ d_2 \\ \vdots \\ d_n \end{bmatrix} = \begin{bmatrix} d_1 \\ d_2 \\ \vdots \\ d_n \\ c_1 \\ c_2 \\ \vdots \\ c_m \end{bmatrix}$$

Se observă că fiecare suport din sistem are o linie în matricea *A* si în vectorul *E*. Dacă un dispozitiv esuează, esecul se materializează în relatia de mai sus prin ignorarea/stergerea liniei care corespunde acelui dispozitiv. Rezultă o matrice *A*' si un vector *E*' cu linii mai putine, dar care verifică o ecuatie asemănătoare cu cea de mai sus

$$A'D = E'$$

Dacă exact *m* dispozitive sunt inutilizabile atunci matricea *A'* este o matrice $n \times n$. Deoarece matricea *F* este de tipul Vandermonde orice submultime de linii este liniar independentă. Matricea *A'* este asadar nesingulară si valorile care compun vectorul *D* pot fi calculate din ecuatia matricială de mai sus prin eliminare Gauss. Prin urmare toate datele pot fi recuperate.

Cu *D* odată obtinut, valorile oricărui suport cu date de verificare C_i esuat pot fi reconstituite. Dacă sunt mai putin de *m* dispozitive cu probleme, alegerea la întâmplare a unui număr de exact *n* linii din *A'* permite eliminarea gaussiană si continuarea este evidentă. Sistemul tolerează până la *m* dispozitive devenite de neutilizat.

Aritmetica în câmpurile Galois

O preocupare majoră în algoritmul RS-RAID o constituie domeniul calculelor care este o multime de cuvinte binare de lungime fixă *w*. Recuperarea dintr-o eroare comisă obisnuit ar putea consta în efectuarea unor calcule modulo 2^w. Maniera aceasta însă nu functionează deoarece împărtirea nu-i definită pentru orice pereche de elemente. De exemplu, 3:2 *modulo* 4 nu este definită. Această situatie face eliminarea Gauss imposibilă în foarte multe cazuri.

Câmpurile cu 2^w elemente sunt câmpuri Galois – notate $GF(2^w)$ – un subiect fundamental în algebra abstractă. Mai jos se definesc moduri eficiente de a aduna, scădea, multiplica si împărti elemente apartinând unui câmp Galois.
Elementele unui câmp Galois $GF(2^w)$ sunt întregi de la zero la $2^w - 1$. Adunarea si scăderea sunt aplicatii simple: operatii XOR (SAU EXCLUSIV). De pildă, în $GF(2^4)$

$$11 + 7 = 1011 \oplus 0111 = 1100 = 12$$
$$11 - 7 = 1011 \oplus 0111 = 1100 = 12$$

Multiplicarea si diviziunea sunt mai complexe. Când w este mic, 16 sau mai mic, se utilizează tabele de logaritmi lungi de $2^w - 1$. Tabelul contine indici, o functie *gflog* si o functie *gfilog*. Ambele functii sunt functii cu valori întregi. Prima este listată pentru indici de la 1 la $2^w - 1$ si este o listă de logaritmi în câmpul Galois, a doua este definită pentru indici de la 0 la $2^w - 2$ si contine rezultatul operatiei inverse logaritmării. Evident, compunerea celor două functii, în orice ordine produce functia identitate, $gflog[gfilog(i)] = i$, $gfilog[gflog(i)] = i$. Cu aceste functii se pot executa operatiile de multiplicare si de împărtire prin luarea logaritmilor factorilor, prin calculul sumei sau a diferentei valorilor obtinute (*modulo* $2^w - 1$) si revenirea la rezultat prin operatia inversă logaritmării. Iată un tabel de logaritmi pentru $w = 4$.

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
gflog(i)		0	1	4	2	8	5	10	3	14	9	7	6	13	11	12
gfilog(i)	1	2	4	8	3	6	12	11	5	10	7	14	15	13	9	

Evident, numai numerele nenule au logaritmi. Logaritnul invers al unui număr i este egal cu logaritmul invers al numărului $[i \bmod (2^w - 1)]$.
Exemple de calcul în aritmetica din $GF(2^4)$:

$$3*7 = gfilog[gflog(3) + gflog(7)] = gfilog[4 + 10] = gfilog[14] = 9$$
$$13*10 = gfilog[gflog(13) + gflog(10)] = gfilog[13 + 9] = gfilog[7] = 11$$
$$13/10 = gfilog[gflog(13) - gflog(10)] = gfilog[13 - 9] = gfilog[4] = 3$$
$$3/7 = gfilog[gflog(3) - gflog(7)] = gfilog[4 - 10] = gfilog[9] = 10$$

Asadar, o multiplicare sau o diviziune necesită trei apeluri la tabel – două pentru logaritmi si unul pentru inversul logaritmului –, o adunare sau o scădere si o operatie de tip *modulo*.
Aritmetica unui câmp Galois are fundamentele date în continuare.
Un câmp $GF(n)$ este o multime de n elemente închisă la operatiile de adunare si multiplicare, cu un invers (opus) raportat la adunare pentru fiecare element, cu un invers raportat la operatia de înmultire pentru fiecare element nenul. Câmpul $GF(2)$ de exemplu, contine două elemente, adunarea si înmultirea se execută *modulo* 2 (operatorii XOR si AND, respectiv). Analog, dacă n este prim atunci $GF(n)$ este multimea $\{0, 1, \ldots, n - 1\}$ în care adunarea si înmultirea se execută *modulo* n.
Dacă $n > 1$ nu este prim, atunci multimea $\{0, 1, \ldots, n - 1\}$ cu adunarea si multiplicarea *modulo* n nu este câmp. De exemplu, dacă $n = 4$ atunci multimea

{0, 1, 2, 3}, închisă la adunare si înmultire nu este câmp deoarece 2 nu are un invers la înmultire, nu există a astfel încât $2a = 1 \pmod 4$. Asadar, nu se poate face codarea cu cuvinte binare de lungime $w > 1$ cu operatiile de adunare si înmultire modulo 2^w. În loc trebuie utilizat câmpul Galois corespunzător.
Explicatiile relative la câmpurile Galois uzează de polinoamele într-o nedeterminată cu coeficienti în $GF(2)$. Adică, dacă $r(x) = x + 1$ si $s(x) = x$ atunci $r(x) + s(x) = 1$ doarece $x + x = (1 + 1)x = 0x = 0$. Mai mult, se pot lua polinoame de acest gen *modulo* alte polinoame conform cu identitatea: dacă $r(x) \bmod q(x) = s(x)$, atunci $s(x)$ este un polinom de grad inferior lui $q(x)$ si $r(x) = q(x)t(x) + s(x)$ cu $t(x)$ un polinom în x.
Dacă, de exemplu, $r(x) = x^2 + x$ si $q(x) = x^2 + 1$ atunci $r(x) \bmod q(x) = x + 1$.
Fie acum $q(x)$ un polinom *primitiv* de gradul w cu coeficienti în $GF(2)$. Primitiv înseamnă că polinomul nu are divizori cu coeficienti în $GF(2)$ si polinomul x este generatorul câmpului $GF(2^w)$. Cum generează x câmpul? Se consideră initial elementele obligatorii 0, 1 si x si se continuă enumerarea elementelor obtinute prin multiplicarea ultimului element cu x si retinerea rezultatului modulo $q(x)$. Enumerarea se încheie cu elementul pentru care rezultatul *modulo* $q(x)$ este egal cu 1.
Dacă, de exemplu, $w = 2$ si $q(x) = x^2 + x + 1$ atunci primele elemente sunt 0, 1 si x, iar $x^2 \bmod q(x) = x + 1$ si cele patru elemente ale câmpului $GF(4)$ sunt {0, 1, x, $x + 1$}. Un al cincilea element nu există deoarece $x(x + 1) = x^2 + x$ care luat *modulo* $q(x)$ produce 1, element deja existent.
Câmpul general $GF(2^w)$ se construieste prin găsirea unui polinom primitiv $q(x)$ de gradul w peste $GF(2)$ urmată de enumerarea elementelor generate de x. Adunarea si multiplicarea elementelor câmpului se fac după regulile adunării si multiplicării polinoamelor cu grija de a lua rezultatul totdeauna *modulo* $q(x)$. Un asemenea câmp se mai scrie ca $GF(2^w) = GF(2)[x]/q(x)$.
Acum, pentru a uza de un câmp $GF(2^w)$ în algoritmul RS-RAID este necesară definirea unei aplicatii a elementelor din acest câmp pe cuvinte binare de lungime w. Un polinom $r(x)$ din $GF(2^w)$ poate fi aplicat pe un cuvânt binar b de lungime w prin punerea celui de al i-lea bit din b egal cu coeficientul puterii x^i din polinom. Pentru $GF(4) = GF(2)[x]/x^2 + x + 1$ se obtine tabelul următor

Elemente generate	Elemente polinomiale	Elemente binare	Reprezentarea zecimală
0	0	00	0
x^0	1	01	1
x^1	x	10	2
x^2	$x + 1$	11	3

Adunarea elementelor se realizează prin operatia SAU EXCLUSIV (XOR) bit-cu-bit. Multiplicarea este mai complicată: se iau elementele sub forma polinomială, se multiplică ca polinoame si se ia rezultatul *modulo* $q(x)$. Tabelele

de logaritmi ca acela de mai sus se bazează pe o tabelă de compunere ca aceea dată pentru cazul $GF(4)$.
Pentru alte valori w, polinoame primitive $q(x)$ se găsesc în literatură. Iată câteva:

$w = 4$: $x^4 + x + 1$
$w = 8$: $x^8 + x^4 + x^3 + x^2 + 1$
$w = 16$: $x^{16} + x^{12} + x^3 + x + 1$
$w = 32$: $x^{32} + x^{22} + x^3 + x + 1$
$w = 64$: $x^{64} + x^4 + x^3 + x + 1$

Cu elementul de pornire $x^0 = 1$, $GF(2^w)$ se completează prin enumerarea elementelor obtinute prin multiplicarea cu x a ultimului element enumerat si luarea rezultatelor *modulo* $q(x)$. Tabelul care urmează cuprinde cazul câmpului $GF(2^4)$ cu polinomul primitiv $q(x) = x^4 + x + 1$. În acelasi tabel se observă si modul cum se generează tabelele de logaritmi si de invers-logaritmi prezentate mai devreme.

Element generat	Polinom	Exprimare binară	Exprimare zecimală
0	0	0000	0
x^0	1	0001	1
x^1	x^1	0010	2
x^2	x^2	0100	4
x^3	x^3	1000	8
x^4	$x + 1$	0011	3
x^5	$x^2 + x$	0110	6
x^6	$x^3 + x^2$	1100	12
x^7	$x^3 + x + 1$	1011	11
x^8	$x^2 + 1$	0101	5
x^9	$x^3 + x$	1010	10
x^{10}	$x^2 + x + 1$	0111	7
x^{11}	$x^3 + x^2 + x$	1110	14
x^{12}	$x^3 + x^2 + x + 1$	1111	15
x^{13}	$x^3 + x^2 + 1$	1101	13
x^{14}	$x^3 + 1$	1001	9
x^{15}	1	0001	1

Sumarul algoritmului

Fiind date n dispozitive pentru date si m dispozitive de control, algoritmul RS-RAID care le face tolerante la cel mult m defecte se aplică astfel:

1. Se alege o valoare pentru w astfel ca $2^w > m + n$. Este convenabil a se alege $w = 8$ sau $w = 16$ ceea ce conduce la cuvinte numărate în octeti (bytes). Pentru $w = 16$ suma $m + n$ poate fi până la 65535.

2. Se stabilesc tabelele cu functiile *gflog* si *gfilog* după metoda dată mai devreme
3. Se calculează matricea F care este o matrice Vandermonde $m \times n$: $f_{i,j} = j^{i-1}$ (pentru $1 \le i \le m$, $1 \le j \le n$), cu operatiile în câmpul Galois $GF(2^w)$.
4. Matricea F se foloseste la calculul si la întretinerea dispozitivelor de verificare pe baza cuvintelor depuse pe dispozitivele de date. Iarăsi, operatiile se fac în $GF(2^w)$.
5. Dacă un număr de dispozitive, mai putine de m clachează atunci ele se reconstituie în maniera care urmează. Se aleg oricare n dispozitive din cele rămase în functie si se construiesc matricea A' si vectorul E' ca mai sus. Se rezolvă apoi pentru D ecuatia $A'D = E'$. Prin aceasta datele de pe dispozitivele de stocare a datelor sunt recuperate. Acum se pot reconstitui si dispozitivele de verificare esuate, prin utilizarea matricii F.

Un exemplu. Se presupune că sunt trei suporturi de date si trei suporturi de verificare si fiecare dintre ele detine un megaoctet. Asadar, $n = 3$ si $m = 3$. Se alege $w = 4$, asa încât $2^w > m + n$. Pentru multiplicări se foloseste tabelul dat mai devreme pentru $GF(2^4)$. În aceste conditii matricea F este

$$F = \begin{bmatrix} 1^0 & 2^0 & 3^0 \\ 1^1 & 2^1 & 3^1 \\ 1^2 & 2^2 & 3^2 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 2 & 3 \\ 1 & 4 & 5 \end{bmatrix}$$

Se pot calcula acum cuvintele de verificare prin relatia $C = FD$. Se admite că primele cuvinte stocate pe cele trei dispozitive de date sunt, respectiv, 3, 13, 9. Calculul cuvintelor de control C_1, C_2, C_3 produce valorile următoare:

$$C_1 = 1*3 + 1*13 + 1*9 = 3 + 13 + 9 = 0011 \oplus 1101 \oplus 1001 = 0111 = 7$$

$$C_2 = 1*3 + 2*13 + 3*9 = 3 + 9 + 8 = 0011 \oplus 1001 \oplus 1000 = 0010 = 2$$

$$C_3 = 1*3 + 4*13 + 5*9 = 3 + 1 + 11 = 0011 \oplus 0001 \oplus 1011 = 1001 = 9$$

Dacă, de pildă, continutul dispozitivului de date cu indicele 2 se modifică si primul număr devine 1, atunci fiecare din dispozitivele de verificare primeste valoarea $(1 - 13) = (0001 \oplus 1101) = 1100 = 12$, care este utilizată pentru recalcularea valorilor de verificare

$$C_1 = 7 + 1*12 = 0111 \oplus 1100 = 1011 = 11$$

$$C_2 = 2 + 2*12 = 2 + 11 = 0010 \oplus 1011 = 1001 = 9$$

$$C_3 = 9 + 4*12 = 9 + 5 = 1001 \oplus 0101 = 1100 = 12$$

Dacă D_2, D_3 si C_3 se pierd atunci, din matricea A si din vectorul E se sterg liniile care corespund dispozitivelor disfuncte pentru a obtine ecuatia $A'D = E'$

$$\begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 1 \\ 1 & 2 & 3 \end{bmatrix} D = \begin{bmatrix} 3 \\ 11 \\ 9 \end{bmatrix}$$

Prin eliminare Gauss se poate inversa matricea A' si se obtine

$$D = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 3 & 1 \\ 3 & 2 & 1 \end{bmatrix} \begin{bmatrix} 3 \\ 11 \\ 9 \end{bmatrix}$$

si valorile reconstituite sunt

$$D_2 = 2*3 + 3*11 + 1*9 = 6 + 14 + 9 = 1$$

$$D_3 = 3*3 + 2*11 + 1*9 = 5 + 5 + 9 = 9$$

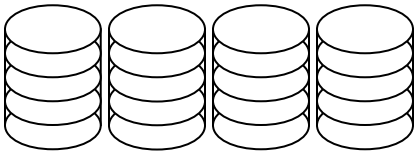
Cu matricea *F* se poate reconstitui si

$$C_3 = 1*3 + 4*1 + 5*9 = 3 + 4 + 11 = 12$$

si sistemul este recuperat în întregime.
Există si alte sisteme RAID la care se fac scurte referiri în continuare.

RAID nivelul 0 (fără redundantă)

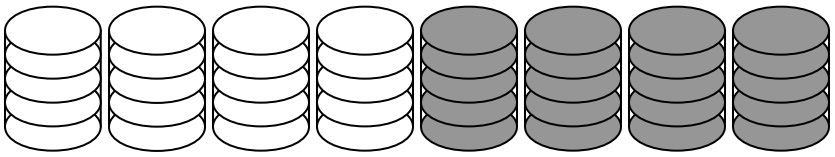
Un sistem de discuri non-redundant (sau de nivel 0) are costul cel mai scăzut din cauza lipsei oricărei redundante. Schema aceasta oferă cea mai bună performantă la scriere deoarece nu necesită vreodată actualizarea vreunei informatii redundante. Surpinzător, nu are cea mai bună performantă la citire. Schemele redundante (ca aceea numită "în oglindă" sau "oglindită", care creează duplicate ale datelor) pot executa mai bine citirile prin planificarea selectivă a cererilor pe discul cu timpul de căutare mediu cel mai scurt si întârzierea rotativă cea mai mică. Fără redundante, orice cădere a unui disc va produce pierderi iremediabile de date. Sistemele de discuri fără redundanțe sunt larg utilizate în supercalcul unde performanta si capacitatea trec ca importanță înaintea fiabilității.



Sistem de discuri fără redundanțe

RAID nivelul 1 (sisteme "în oglindă")

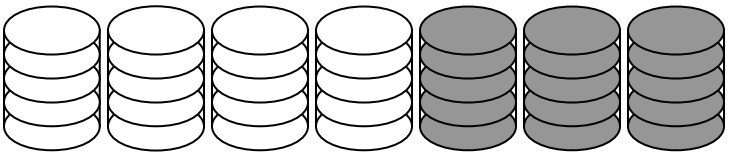
Sistemul traditional denumit "în oglindă" sau "cu umbră" utilizează de două ori mai multe discuri decât un sistem de discuri fără redundante. Ori de câte ori un articol-dată este scris pe un disc el este scris si pe un disc redundant, astfel încât există totdeauna două cópii ale informatiei, două exemplare. Când articolul trebuie citit, el poate fi recuperat de pe disc cu întârzieri de asteptare, de căutare si rotationale mai scurte. Dacă un disc se defectează, copia este utilizată pentru serviciul cerut. Oglindirea este utilizată frecvent în aplicatii cu baze de date, când disponibilitatea sistemului si viteza tranzactiilor sunt mai importante decât eficienta stocării.



Sistem de discuri "în oglindă"

RAID nivelul 2 (memorie în stilul codurilor corectoare de erori)

Sistemele de memorii asigură recuperarea din starea de disfunctie a unor componente la costuri mai reduse decât prin oglindire, dacă folosesc codurile Hamming. Codurile Hamming fac verificări de paritate pe submultimi de componente distincte si suprapuse. În una din variantele acestei scheme, patru discuri de date necesită trei discuri redundante, unul mai putin decât în cazul sistemului oglindă. Deoarece numărul de discuri redundante este proportional cu logaritmul numărului total de discuri din sistem, eficienta memorării creste odată cu numărul discurilor de date.

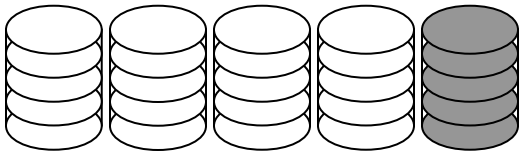


Sistem de discuri în stilul codurilor corectoare de erori

Dacă unul (si numai unul) din discuri cade, componentele mai multor discuri de paritate devin inconsistente cu datele si componenta defectă este identificată: este componenta comună tuturor subseturilor incorecte. Informatia pierdută este recuperată prin regulile obisnuite ale codului Hamming utilizat.

RAID nivelul 3 (paritate cu biti intercalati)

Sistemului din paragraful anterior i se pot aduce îmbunătătiri prin observarea faptului că spre deosebire de căderea elementelor de memorie, controlerele discurilor pot identifica usor care disc este cel cu defect. Astfel, pentru recuperarea informatiei pierdute se poate utiliza un singur disc de paritate si nu un set de discuri de paritate.

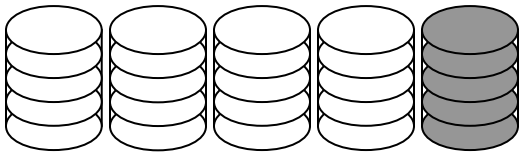


Sistem cu paritate prin biti intercalati

Într-un sistem de discuri cu paritate de biti intercalati, datele sunt conceptual intercalate bit-cu-bit pe discurile de date si se adaugă un singur disc de paritate pentru a tolera căderea unui disc (si numai a unuia). Fiecare cerere de citire accesează toate discurile de date si fiecare cerere de scriere accesează toate discurile de date si discul de paritate. Astfel numai o cerere poate fi servită la un moment dat. Deoarece discul de paritate contine numai informatia de paritate si nu date, discul de paritate nu poate participa activ la citiri, ceea ce produce o usoară scădere în performanta la citire fată de sistemele cu redundante care distribuie informatia de paritate si datele pe toate discurile. Sistemele de discuri cu paritate pe biti intercalati sunt utilizate frecvent în aplicatii care cer o lărgime de bandă mare dar nu viteze intrare-iesire mari. Mai sunt si simplu de implementat.

RAID nivelul 4 (paritate pe blocuri intercalate)

Există o similitudine între sistemele de discuri cu intercalare de biti si cele cu intercalare de blocuri. Deosebirea constă în obiectul operatiei de intercalare: nu biti, ci blocuri de dimensiune arbitrară. Dimensiunea acestor blocuri este denumită unitatea de realizare de benzi (*striping*). Citirile cerute, mai mici decât unitatea de *striping* accesează numai un disc de date. Cererile de scriere trebuie să actualizeze blocurile de date cerute si trebuie totodată să calculeze si să actualizeze blocul de paritate. Pentru scrieri de mare întindere care ating blocuri pe toate discurile, paritatea este calculată observînd cum diferă datele noi de cele vechi si aplicând acele diferente pe blocul de paritate. Scrierile de mică întindere cer asadar patru operatii de intrare-iesire pe disc: una pentru a scrie articolul nou, apoi două pentru a citi vechiul articol si vechea paritate pentru a calcula noua informatie de paritate si una de scriere a noii parităti. Această operatie este cunoscută ca o procedură citeste-modifică-scrie. Deoarece un sistem de discuri cu paritate cu bloc intercalat are numai un disc de paritate, care trebuie actualizat la toate operatiile de scriere, discul de paritate poate deveni cu usurintă un loc îngust, o strangulare. Din cauza acestei posibile limitări sunt de preferat sistemelor de discuri cu paritate pe blocuri, sistemele de discuri cu paritate pe blocuri distribuite.

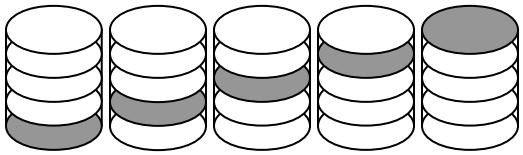


Sistem de discuri cu paritate pe blocuri intercalate

RAID nivelul 5 (paritate pe blocuri intercalate distribuite)

Sistemele de discuri cu paritate pe blocuri distribuite elimină strangularea de pe discul de paritate, care se constată la sistemele cu paritate pe blocuri intercalate,

prin distribuirea informatiei de paritate uniform pe toate discurile. Un avantaj suplimentar al distribuirii informatiei de paritate uniform pe toate discurile, frecvent trecut cu vederea, constă în distribuirea datelor pe toate discurile si nu pe toate discurile cu exceptia unuia. Aceasta permite tuturor discurilor să participe la servirea operatiilor de citire spre deosebire de schemele redundante cu discuri de paritate dedicate exclusiv în care discurile de paritate nu pot participa la servirea solicitărilor de citire. Sistemele cu paritate pe blocuri intercalate distribuite au una dintre ele mai bune performante pentru citiri restrânse, citiri lungi si scrieri lungi, între toate sistemele de discuri cu redundante. Cu toate acestea, cererile de citire de lungime restrânsă sunt întrucâtva ineficiente comparativ cu alte scheme redundante, cum sunt cele cu oglindire, datorită necesitătii de a executa operatii citeste-modifică-scrie pentru actualizarea parităţii. Aceasta este partea slabă majoră a sistemelor RAID de nivelul 5, în ceea ce priveste performantele.



Sistem de discuri cu paritate pe blocuri intercalate distribuite

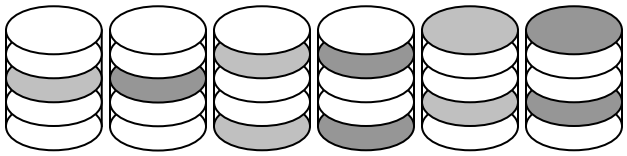
Metoda exactă utilizată pentru a distribui paritatea în sistemele cu paritate distribuită în blocuri intercalate are impact asupra performantelor. Figura alăturată ilustrează cea mai bună distributie a informatiei de paritate (discurile gri), numită distributie de paritate simetrică la stânga. O proprietate utilă a acestui gen de distributie constă în faptul că ori de câte ori sunt traversate secvential unitătile de *striping*, fiecare disc este accesat în succesiune o dată înainte de a fi accesat a doua oară. Această proprietate reduce conflictele de disc atunci când sunt servite solicitările mari.

0	1	2	3	**P_0**
5	6	7	**P_1**	4
10	11	**P_2**	8	9
15	**P_3**	12	13	14
P_4	16	17	18	19

Sistem RAID de nivelul 5 cu simetrie la stânga

RAID nivelul 6 (redundante P + Q)

Paritatea este verificată printr-un cod redundant capabil a corecta orice defectare singulară care se autoidentifică. Dacă sunt luate în considerare sisteme cu discuri mai numeroase, este necesar a utiliza coduri mai puternice, capabile să tolereze defecte multiple. Mai mult, când un disc într-un sistem protejat prin paritate cade, recuperarea continutului discului defect reclamă lectura cu succes a continutului tuturor discurilor functionale. În aceste cazuri, probabilitatea de a întâlni în cursul recuperării o eroare de citire necorectabilă poate fi semnificativă. Asadar, aplicatiile cu cerinte de fiabilitate mai severe trebuie tratate cu coduri corectoare de erori mai puternice.



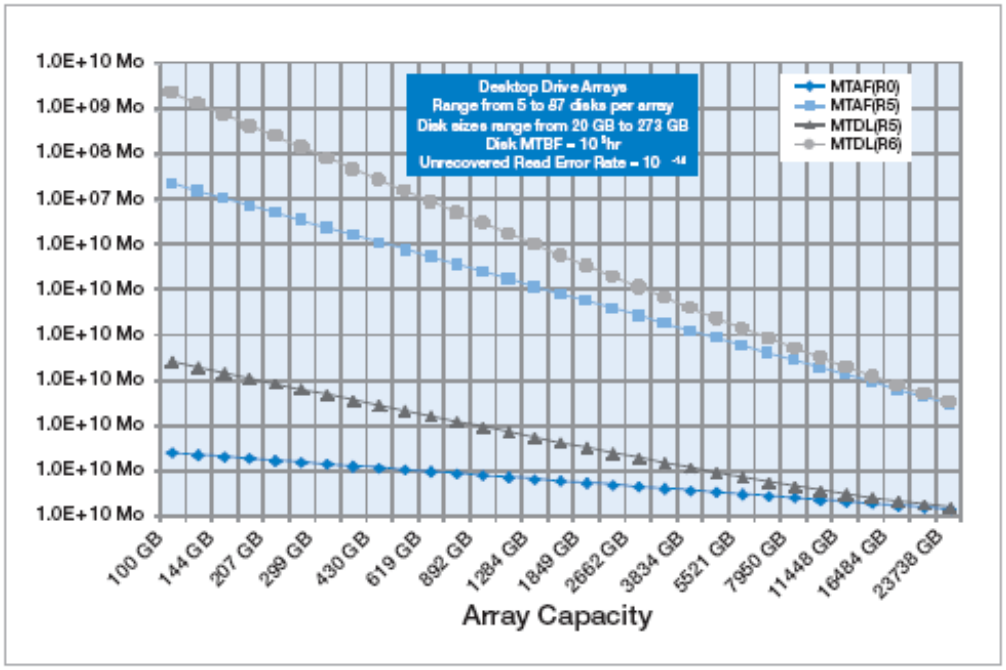
Sistem de discuri cu redundante P + Q

O astfel de schemă, denumită adesea schemă cu redundantă P + Q, foloseste codurile Reed-Solomon pentru protectia fată de căderea a până la două discuri utilizând cel putin două discuri redundante. Sistemele de discuri cu redundante P + Q sunt structural foarte asemănătoare sistemelor cu paritate distribuită bloc-intercalată si operează în mare măsură în acelasi mod. În particular, sistemele de discuri cu redundante P + Q execută operatii scurte de scriere în maniera citeste-modifică-scrie cu deosebirea că în loc de patru accesări de disc sunt aici necesare sase accesări pentru a actualiza atât informatiile P cât si cele Q.
Sistemele RS-RAID prezentate cu mai multe detalii în prima parte a acestei sectiuni se încadrează în această clasă de sisteme de nivelul 6.

Alte sisteme RAID

Literatura mai mentionează:

a) Sistemele de **nivel 0+1** – oglindă si *striping* – cu două subsisteme 0 cu *striping* si un subsistem 1 suprapus acestora. Se utilizează si în cazul replicării datelor pentru partajarea lor.
b) Sistemele de **nivel 1+0** – *striping* pe oglinzi – în care sunt create subsisteme RAID 1 si peste acestea un subsistem RAID 0 de *striping*.
c) Sisteme de **nivelul 7** – un sistem brevetat de Storage Computer Corporation care adaugă sectiuni cache la sistemele de nivelul 3 si 4.
d) Sisteme **RAID S** – proprietar EMC Corporation – sisteme cu paritate si *striping* utilizate în sistemele proprii de memorie Symmetrix.

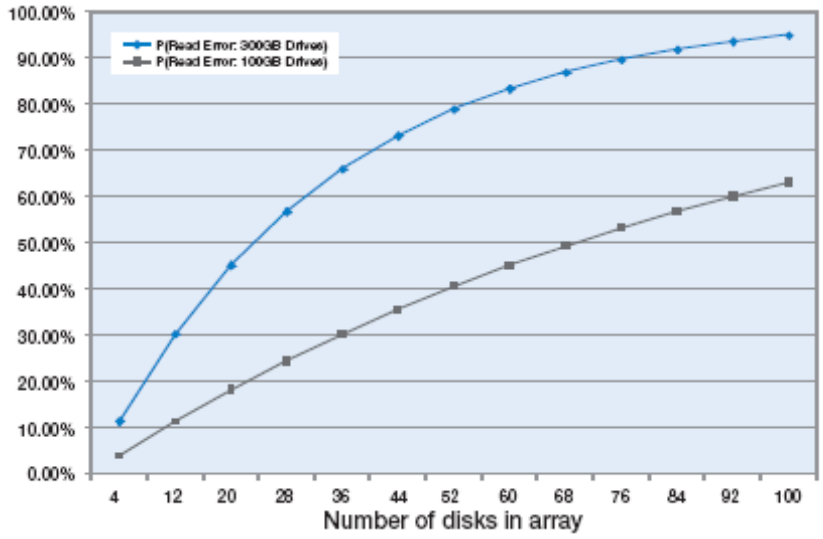


Timpul mediu până la defectare în functie de capacitatea de memorare
Structuri de discuri De la 5 la 87 de discuri pe structură.
Capacitatea discurilor de la 20 la 273 GB. MTBF pentru un disc 10^5 ore.
Rata erorilor de citire nerecuperate 10^{-14}.

Comparatii de costuri si de performanțe

Primele trei măsuri prin care se evaluează un sistem de discuri sunt fiabilitatea, performanța si costul. Sistemele RAID de la 0 la 6 acoperă o gamă largă de compromisuri între aceste trei măsuri. Este necesar a lua în considerare toate aceste trei măsuri pentru a întelege deplin valoarea si costul fiecărei organizări a sistemelor de discuri. Despre fiabilitate – în graficul alăturat.
Linia cea mai de jos (romburi) reprezintă timpul mediu până la defectare pentru un singur disc. Într-un sistem fără redundante (RAID 0) defectarea produce pierderea datelor. Linia următoare (triunghiuri) arată timpul mediu până la pierderea de date, MTDL (Mean Time to Data Loss) pentru un sistem RAID 5 cu probabilitatea de a găsi un defect latent după reconstituirea informației. De observat că un sistem RAID 5 cu capacitatea totală mai mare de 5 TB ar putea pierde date de mai multe ori în decursul unui an. Pentru a ilustra impactul defectelor latente asupra calculului MTDL, sau timpul mediu până la eroarea aditională, MTAF (Mean Time to Additional Failure), curba următoare (pătrate) arată probabilitatea căderii din cauza a două erori de disc, care ignoră defectele latente, pentru un sistem RAID 5. Ignorarea impactului defectelor latente arată că MTDL pentru un sistem RAID 5 rămâne destul de bun chiar la capacități de peste 5 TB. Linia cea mai de sus (cercuri) arată MTDL pentru un sistem RAID

6 cu luarea în considerare a probabilitătii de a apărea si defecte latente. Linia arată MTDL pentru sistemele RAID 6, chiar tinând seamă de impactul defectelor latente, care este cu ordine de mărime mai bun decât cel pentru un sistem RAID 5 comparabil.

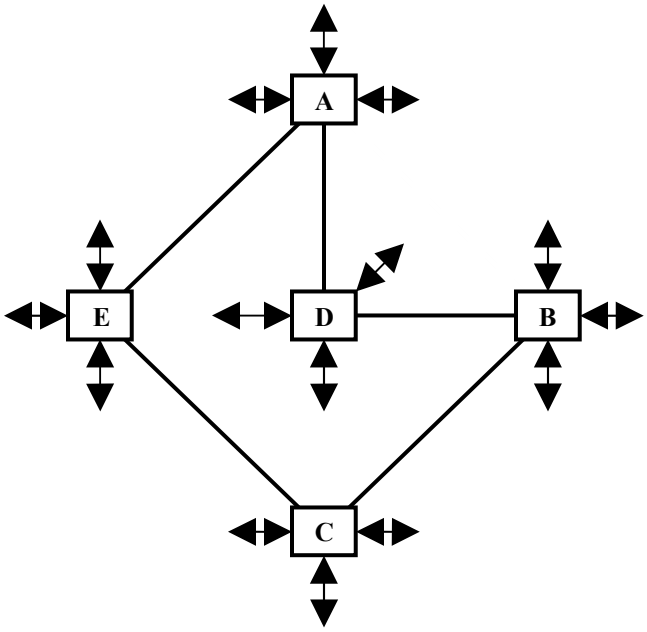


Probabilitatea erorilor de citire irecuperabile în timpul reconstructiei
Rata erorilor nerecuperabile ale discului: 1 la 10^{14} biti cititi

Pentru întelegerea mai exactă a modului în care defectele latente din sistemele RAID 5 afectează MTDL să examinăm probabilitatea de a întâlni un defect latent în timpul operatiei de reconstituire. Dacă un *controller* de RAID 5 întâlneste un defect în timpul reconstituirii, datele utilizatorului sunt pierdute deoarece discul defect si sectorul defect reprezintă două elemente lipsă, ceea ce depăseste capacitatea sistemului RAID 5 de a recupera date pierdute. Figura alăturată arată probabilitatea de a găsi un defect latent în timpul reconstructiei sistemului la capacităti ale discurilor variate, din ce în ce mai mari. Pentru sisteme foarte mari de discuri de mare capacitate, ar fi surprinzător a nu găsi un defect latent în timpul reconstructiei. Graficul presupune o rată a erorilor tipică pentru discuri din clasa desktop. Probabilitatea este un ordin de mărime mai mică pentru discuri din clasa *enterprise*.

REPLICAREA DATELOR PENTRU TOLERANȚA LA DEFECTE

Cópii identice ale datelor sunt detinute în mai multe noduri ale unui sistem distribuit. O asemenea replicare a datelor poate oferi performante îmbunătătite în ceea ce priveste toleranța la defecte.



Replicile datelor trebuie mentinute consistente în pofida oricăror căderi din sistem.
Exemplul din figura de mai sus poate retine cinci cópii în cinci noduri. Dacă nodul *A* este deconectat si un *write* actualizează copia din *A*, celelalte cópii nu vor mai fi consistente cu *A*. Orice *read* al datelor lor va rezulta în utilizarea de date vechi, depăsite. Se pune întrebarea: Câte cópii ar trebui citite (scrise)? Răspunsul este dat în continuare pe baza unor scheme consacrate.

O schemă simplă cu voting neierarhizată

Se atribuie v_i voturi copiei i a datelor.
S – multimea tuturor nodurilor cu cópii ale datelor.
V – suma tuturor voturilor cu $V = \sum_{i \in S} v_i$.

r, w – variabile astfel ca $r + w > V$; $w > V/2$.

$V(X)$ – numărul total de voturi atribuite cópiilor din multimea X – $V(X) = \sum_{i \in X} v_i$.

Este necesară o strategie care asigură că fiecare *read* utilizează date de ultimă versiune. Pentru a face lectura completă, se citesc noduri dintr-o multime $R \subseteq S$ astfel încât $V(R) \geq r$. Pentru a face scrierea completă, se scrie în fiecare nod al unei multimi $W \subseteq S$ astfel încât $V(W) \geq w$.

Justificarea procedurii este prezentată în continuare.

O multime R pentru care $V(R) \geq r$ este numită cvorum de lectură. O multime W pentru care $V(W) \geq w$ se numeste cvorum de scriere.

Pentru oricare două multimi R si W pentru care $V(R) \geq r$ si $V(W) \geq w$, $R \cap W \neq \emptyset$ (deoarece $r + w > V$).

Orice operatie de citire (*read*) este garantată a citi valoarea a cel putin unei cópii care a fost actualizată prin cel putin un *write*. Mai mult, pentru oricare două multimi W_1, W_2 astfel încât $V(W_1)$, $V(W_2) \geq w$, are loc $W_1 \cap W_2 \neq \emptyset$.

Exemplu: Dacă se atribuie un vot fiecărui nod (v. figura de mai sus), atunci suma tuturor voturilor este $V = 5$. Este necesar ca $w > 5/2$ si $r > 5 - w$. Combinatiile permise de r si w sunt (1, 5), (2, 5), (3, 5), (4, 5), (5, 5), (2,4), (3, 4), (4, 4), (5, 4), (3, 3).

Se consideră spre exemplificare perechea (r, w) = (1, 5). Atunci există un *read* posibil al oricăreia din (cele) cinci cópii, iar un *write* trebuie să actualizeze fiecare din cele cinci cópii.

Fiecare operatie de citire obtine cele mai actuale date.

Dacă $w = 5$, retinerea unui $r > 1$ încetineste operatia de citire.

Dacă nodul A este (accidental) deconectat, se poate citi încă din fiecare nod dar nu se pot actualiza toate nodurile.

Se consideră acum (r, w) = (3, 3); numai trei cópii de scris dar cititul consumă timp mai îndelungat.

Dacă nodul A este (accidental) deconectat, scrierea si lectura în/din A devine imposibilă dar cele patru noduri rămase pot continua cu scrieri si citiri în modul uzual.

Performantă si disponibilitate

Alegerea celor două numere r si w afectează performanta si disponibilitatea sistemului. Dacă sunt de făcut mai multe citiri decât scrieri se alege un r mai mic pentru a accelera operatiile de citire.

Alegerea $r = 1$ impune în exemplul discutat $w = 5$; scrierea nu poate fi executată dacă fie si numai un singur nod este (accidental) deconectat.

Alegerea $r = 2$ permite $w = 4$ si scrierea poate fi încă executată dacă patru din cinci noduri sunt conectate în retea. Este vorba aici de un *trade-off* între performante si disponibilitate.

Analiza fiabilitătii si disponibilitătii

Disfunctiile apar în orice nod potrivit unui proces Poisson de rată λ (se admite că legăturile nu cad). Când un nod cade, el este reparat si îi sunt încărcate date actuale. Timpul consumat cu reparatia este aleator, distribuit exponential cu media $1/\mu$.
Exemplu: $(r, w) = (3, 3)$, ambele operatii de citire si de scriere pot avea loc dacă cel putin trei din cinci noduri sunt functionale. Pentru a calcula fiabilitatea si disponibilitatea se folosesc pentru modelare lanturile Markov.
Fiabilitatea pentru cazul $(r, w) = (3, 3)$ urmează etapele de mai jos.
Starea: se defineste ca numărul de noduri căzute, disfuncte. Se scriu următoarele ecuatii diferentiale:

$$\frac{dp_0(t)}{dt} = -5\lambda p_0(t) + \mu p_1(t)$$

$$\frac{dp_1(t)}{dt} = 5\lambda p_0(t) - (4\lambda + \mu) p_1(t) + 2\mu p_2(t)$$

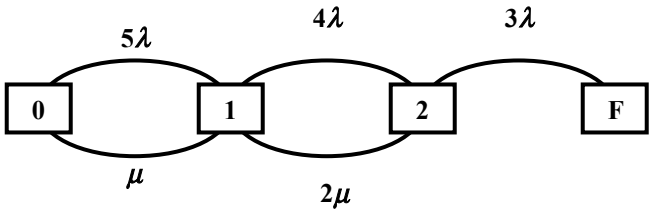
$$\frac{dp_2(t)}{dt} = 4\lambda p_1(t) - (3\lambda + 2\mu) p_2(t)$$

$$\frac{dp_3(t)}{dt} = 3\lambda p_2(t)$$

Fiabilitatea la momentul t este

$$R(t) = 1 - P_F(t)$$

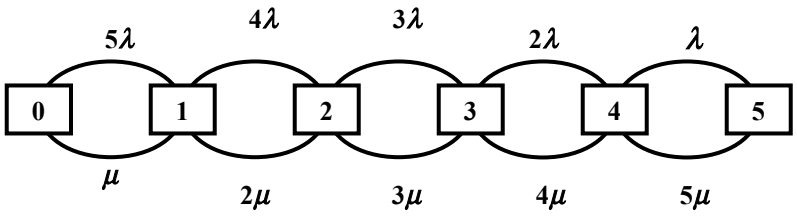
(indicele F se referă la disfunctia totală – Failure).



Disponibilitatea pentru acelasi caz, $(r, w) = (3, 3)$ se calculează pe baza ecuatiilor pentru regimul stationar:

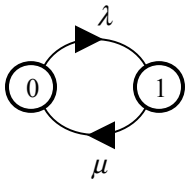
$$\begin{aligned}
5\lambda\ \pi_0 - \mu\ \pi_1 &= 0 \\
-5\lambda\ \pi_0 + (4\lambda + \mu)\pi_1 - 2\mu\ \pi_2 &= 0 \\
-4\lambda\ \pi_1 + (3\lambda + 2\mu)\pi_2 - 3\mu\ \pi_3 &= 0 \\
-3\lambda\ \pi_2 + (2\lambda + 3\mu)\pi_3 - 4\mu\ \pi_4 &= 0 \\
-2\lambda\ \pi_3 + (\lambda + 4\mu)\pi_4 - 5\mu\ \pi_5 &= 0 \\
\pi_0 + \pi_1 + \pi_2 + \pi_3 + \pi_4 + \pi_5 &= 1
\end{aligned}$$

Sistemul este disponibil dacă se află în una din stările 0, 1 sau 2.
Disponibilitatea stationară este $\pi_0 + \pi_1 + \pi_2$



Disponibilitatea poate cunoaste si o a doua tratare. Fiecare nod este considerat separat. Starea unui nod este 0 sau 1, functional, respectiv căzut. Pentru un nod:

$$\pi_0 = \mu/(\lambda + \mu);\ \pi_1 = \lambda/(\lambda + \mu)$$



Disponibilitatea sistemului *A* este dată de probabilitatea ca cel putin 3 noduri să fie functionale. Nodurile sunt presupuse a fi statistic independente si atunci

$$A = \sum_{i=3}^{5} C_5^i \left(\frac{\mu}{\lambda + \mu}\right)^i \left(\frac{\lambda}{\lambda + \mu}\right)^{5-i}$$

Exercitiu: Să se calculeze disponibilitatea punctuală (momentană) $PA(t)$ în cazul aceleiasi retele.

În sistemele cu reparatii, interesul principal este legat de disponibilitate (sau de disponibilitatea punctuală) si mai putin de fiabilitate.

Asignarea votului pentru maximizarea disponibilităţii

În general, nodurile au fiabiltăti si disponibilităti diferite, variate. În plus, si legăturile pot cădea.

Disponibilitatea punctuală este probabilitatea ca *la momentul t* sistemul să fie functional, să existe cvorumuri de citire si de scriere.

O problemă care comportă discutie o constituie atribuirea de voturi nodurilor pentru a maximiza disponibilitatea punctuală.

Asignarea optimală riguroasă este dificilă. În loc este necesară o euristică.

Mai întâi câteva notatii. Pentru un anumit moment *t* fixat (*t* omis pentru simplitatea scrierii):

- Disponibilitatea punctuală pentru nodul i – $a_n(i)$
- Disponibilitatea punctuală pentru legătura j – $a_l(j)$
- $L(i)$ – multimea de conexiuni (legături) la nodul i.

Euristica asignării voturilor

Euristica 1: se pune

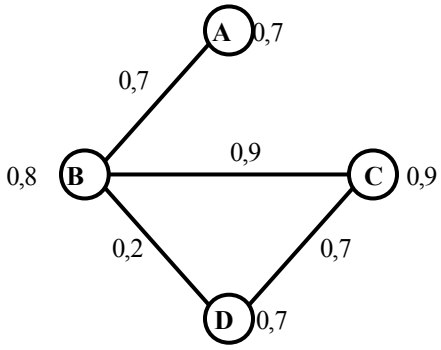
$$v(i) = a_n(i) \sum_{j \in L(i)} a_l(j)$$

(cu rotunjire la cel mai apropiat întreg).
Dacă suma tuturor voturilor este pară, se dă un vot în plus unuia din nodurile cu număr maxim de voturi.
Euristica 2: $k(i, j)$ – nod conectat la nodul i prin conexiunea j.
Se pune

$$v(i) = a_n(i) + \sum_{j \in L(i)} a_l(j) a_n[k(i, j)]$$

(cu rotunjire la cel mai apropiat întreg).
Din nou, dacă suma tuturor voturilor este pară, se dă un vot în plus unuia din nodurile cu număr maxim de voturi.

Exemplu pentru euristica 1



v(A) = round(0,7x0,7) = 0
v(B) = round(0,8x1,8) = 1
v(C) = round(0,9x1,6) = 1
v(D) = round(0,7x0,9) = 1

Nodul A este nesigur, nefiabil comparativ cu celelalte noduri si de aceea nu primeste voturi.
Suma voturilor este 3, cvorumurile trebuie să satisfacă $r + w > 3$; $w > 3/2 \Rightarrow w = 2$ sau $w = 3$.
Dacă $w = 2$, $r = 2$ este cel mai redus cvorum. Cvorumuri posibile pentru *read* (sau *write*): BC, CD, BD.
Dacă $w = 3$, $r = 1$ este cvorumul cel mai redus. Cvorumuri posibile pentru citire: B, C, D. Un singur cvorum pentru scriere: BCD.

Exemplu pentru euristica 2

v(A) = round(0,7 + 0,7x0,8) = 1
v(B) = round(0,8 + 0,7x0,7 + 0,9x0,9 + 0,2x0,7) = 2
v(C) = round(0,9 + 0,9x0,8 + 0,7x0,7) = 2
v(D) = round(0,7 + 0,2x0,8 + 0,7x0,9) = 1

Suma voturilor este pară si de aceea B primeste un vot suplimentar astfel că asignarea finală este v(A) = 1, v(B) = 3, v(C) = 2, v(D) = 1.
Suma voturilor este 7, cvorumurile pentru citire si scriere trebuie să satisfacă relatiile $r + w > 7$, $w > 7/2$, $w \in \{4, 5, 6, 7\}$.
Cvorumuri rezultate din euristica 2, cvorumuri posibile pentru $r + w = 8$.

r	w	Cvorumuri la citire	Cvorumuri la scriere
4	4	AB, BC, BD, ACD	AB, BC, BD, ACD
3	5	B, AC, CD	BC, ABD
2	6	B, C, AD	ABC, BCD
1	7	A, B, C, D	ABCD

Fiecare pereche (r, w) are asociată o disponibilitate, probabilitatea ca cel putin un cvorum de citire si unul de scriere să existe indiferent de căderile unor noduri si/sau ale unor legături.
Listele de cvorumuri sunt identice pentru citire si scriere în cazul (r, w) = (4, 4).
Alte combinatii (r, w) au listele de cvorumuri *read*/*write* diferite.
Calculul disponibilitătii pentru cazul (r, w) = (4, 4):
Disponibilitatea A este probabilitatea ca cel putin unul din cvorumurile AB, BC, BD, ACD să poată fi utilizat.
Marcăm evenimentele E_1, E_2, E_3, E_4 ca fiind respectiv AB, BC, BD, ACD functionale deci utilizabile.
Evenimentele astfel definite nu sunt mutual exclusive.
$A = \Pr(E_1 \cup E_2 \cup E_3 \cup E_4) =$
$= \Sigma_i \Pr(E_i) - \Sigma_{i<j} \Pr(E_i \cap E_j) + \Sigma_{i<j<k} \Pr(E_i \cap E_j \cap E_k) - \Pr(E_1 \cap E_2 \cap E_3 \cap E_4)$
Calcule (patiale):
$\Pr(E_1)$ = Pr(cvorumul AB utilizabil) = 0,7 × 0,7 × 0,8 = 0,392
$\Pr(E_2 \cap E_3)$ = Pr(cvorumurile BC si BD utilizabile) = 0,8 × 0,9 × 0,9 × 0,2 × 0,7 = 0,091.
Exercitiu: Să se completeze calculele pentru (r, w) = (4, 4).
Urmează o metodă diferită de a calcula disponibilitatea.
Sistemul are 8 module, 4 noduri si 4 conexiuni, fiecare modul poate fi functional sau disfunct. Rezultă un total de $2^8 = 256$ de stări mutual exclusive.
Probabilitatea fiecărei stări este un produs de 8 factori, fie $a_n(i)$, fie $[1 - a_n(i)]$, fie $a_l(i)$, fie $[1 - a_l(i)]$.
Un mod sistematic (dar cu consum de timp mare) de a calcula disponibilitatea: se listează toate stările si se adună probabilitătile acelora pentru care există un cvorum.

Pentru orice alte valori (r, w) cvorumurile pentru lectură si pentru scriere sunt diferite.
Se calculează disponibilitatea prin aflarea stărilor în care există atât cvorum pentru scriere cât si cvorum pentru citire.

Asignarea dinamică de voturi

Dacă repararea nu este suficient de rapidă, sistemul se poate degrada. Dacă sistemul se degradează suficient, nu mai există un cluster conex cu o majoritate totală de voturi. Solutia este cea a unor cvorumuri ajustabile în locul celor satice.
Se presupune că fiecare nod are atribuit exact un vot. Pentru fiecare articol-dată sunt mentinute numere de versiune, incrementate cu fiecare actualizare. Acestă incrementare poate fi executată numai dacă se poate aduna un cvorum pentru scriere.
Notatiile pentru asignarea dinamică a voturilor este dată în continuare.
VN_i – numărul versiunii datelor în nodul i.
SC_i – cardinalitatea site-urilor de actualizare la nodul i – numărul de noduri care au participat la actualizarea a VN_i-a a acestor date. Când sistemul începe operatia, SC_i este initializat la numărul total de noduri din sistem.
S_i – multimea de noduri cu care nodul i poate comunica.
M – numărul maxim al versiunii din S_i.
I – multimea partială de multimi S_i cu nodurile cu numărul de versiune M.
N – cardinalitatea maximă a site-urilor de actualizare (SC_i) de noduri din I.
Algoritmul de asignare dinamică a voturilor este următorul:

1. Dacă la nodul i soseste o cerere se actualizare, nodul i calculează următoarele cantităti:
- $M = \max\{VN_j, j \in S_i\}$ (S_i este multimea nodurilor cu care nodul i poate comunica, inclusiv el însusi), adică numărul de versiune maxim al datei de interes peste toate nodurile cu care nodul i poate comunica.
- $I = \{j | VN_j = M, j \in S_i\}$, adică multimea tuturor nodurilor cu numărul de versiune egal cu cel maxim.
- $N = \max\{SC_j, j \in I\}$, adică maxima cardinalitate a site-urilor de actualizare asociate cu toate nodurile din I.

2. Dacă $||I|| > N/2$ atunci nodul i poate creste un cvorum de scriere si este admis a executa actualizarea tuturor nodurilor din I. Dacă $S_i - I \neq \emptyset$ atunci toate elementele din $S_i - I$ trebuie să aibă versiunea lor si numerele de versiune aduse la actualitate înainte ca ele să poată face această actualizare. Suplimentar, numărul de versiune al fiecărei cópii a acelei date din S_i este incrementat, adică VN_j este incrementat pentru fiecare $j \in S_i$. Totodată, pentru fiecare $j \in S_i$, se pune $SC_j = ||I||$. Acest pas în totalitate trebuie executat atomic: toate aceste operatii trebuie făcute în fiecare nod din S_i sau deloc, în nici unul dintre ele.

Exemplu de atribuire dinamică a voturilor: cazul cu sapte noduri, aceleasi date, cu starea la momentul t_0

	A	*B*	*C*	*D*	*E*	*F*	*G*
VN	5	5	5	5	5	5	5
SC	7	7	7	7	7	7	7

O disfunctie face multimea nodurilor neconexă: cele două submultimi disjuncte sunt $\{A, B, C, D\}$ si $\{E, F, G\}$.
E primeste o solicitare de actualizare – $SC_E = 7$. *E* trebiue să găsească încă cel putin 7/2 noduri (inclusiv el însusi) dar nu poate găsi mai mult de 3: cererea de actualizare este refuzată.
A primeste o cerere de actualizare. Cererea poate fi onorată si *A*, *B*, *C*, *D* sunt actualizate.
Starea nouă rezultată este

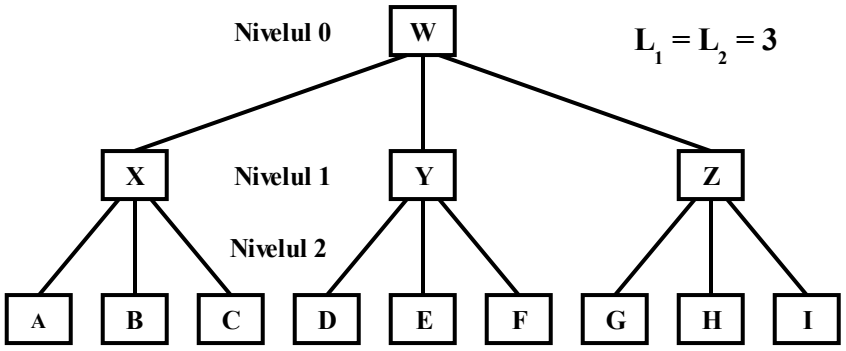
	A	*B*	*C*	*D*	*E*	*F*	*G*
VN	6	6	6	6	5	5	5
SC	4	4	4	4	7	7	7

Apare o altă disfunctie si submultimile disjuncte devin $\{A, B, C\}$, $\{D\}$, $\{E, F, G\}$.
O cerere de actulizare soseste la *C*, cvorumul de scriere este 3 în nodul *C*, actualizarea se face cu succes.
Rezultă o stare nouă:

	A	*B*	*C*	*D*	*E*	*F*	*G*
VN	7	7	7	6	5	5	5
SC	3	3	3	4	7	7	7

Votarea poate fi organizată ierarhic. Dacă *V* este mare, $r + w$ este de asemenea mare, operatiile făcute cu datele consumă timp îndelungat. O solutie posibilă este adoptarea unei scheme de votare ierarhice.
Pentru aceasta se construieste un arbore cu *m* niveluri. Toate nodurile care detin cópii ale datelor sunt frunze la nivelul $m - 1$. Se adaugă noduri virtuale la nivelurile mai înalte până la nivelul 0, cel al rădăcinii. Nodurile adăugate sunt grupări virtuale de noduri reale. Fiecare nod din nivelul *i* va avea exact L_{i+1} descendenti.
Un exemplu de arbore pentru generare ierarhică de cvorumuri este dat în figura alăturată.
Algoritmul de generare a cvorumurilor este următorul:
Se atribuie câte un vot fiecărui nod din arbore.

Se fixează cvorumul de citire si cvorumul de scriere la nivelul i, r_i si w_i astfel încât $r_i + w_i > L_i$; $w_i > L_i/2$.
Mai departe algoritmul se utilizează recursiv.
Se marchează "de citire" rădăcina la nivelul 0.
La nivelul 1 se marchează "de citire" r_1 noduri.
Se trece de la nivelul i la nivelul $i + 1$, se marchează "de citire" r_{i+1} descendenti ai fiecărui nod marcat "de citire" la nivelul i.
Nu se poate marca "de citire" un nod care nu are cel putin r_{i+1} descendenti fără defecte.
Se continuă până când $i = m - 1$.
Frunzele marcate "de citire" alcătuiesc cvorumul de citire.
Formarea unui cvorum de scriere este similară.



Exemplu numeric pentru $i = 1, 2$, $w_i = 2$, $r_i = L_i - w_i + 1 = 2$.
Pornind de la rădăcină se marchează "de citire" doi din descendentii săi, să spunem X si Y.
Se marchează "de citire" doi descendenti pentru X si Y , de pildă A, B pentru X si D, E pentru Y.
Cvorumul de citire este multimea frunzelor marcate "de citire": A, B, D, E.
Se presupune acum că D este defect. În această stare D nu poate fi parte a cvorumului de citire.
Trebuie ales un alt descendent al lui Y, de pildă F, pentru un nou cvorum de citire.
Dacă doi din descendentii lui Y sunt defecti, Y însusi nu poate fi marcat "de citire"; se revine si se încearcă marcarea "de citire" a lui Z.
Exercitiu: Să se listeze cvorumurile generate de $r_1 = 1$, $w_1 = 3$, $r_2 = 2$, $w_2 = 2$.
Relativ la tratarea ierarhică si non-ierarhică se pot face unele observatii. Astfel:
Cvorumul de citire constă în numai 4 cópii. Similar, putem avea un cvorum de scriere cu 4 cópii.
Pentru tratarea lipsită de ierarhizare, cu un vot pe nod, $r + w > 9$, $w > 9/2$. w este cel putin 4, comparativ cu exact 4 în tratarea ierarhizată pe arbore.
Pentru a dovedi că tratarea ierarhică lucrează, se poate arăta că orice cvorum de citire posibil trebuie să intersecteze orice cvorum de scriere în cel putin un nod.

Solutia cu backup primar

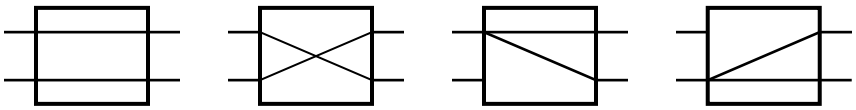
Un nod este desemnat ca nod primar; toate accesările se produc prin acel nod. Celelalte noduri sunt desemnate ca noduri de backup.
În operarea normală, toate scrierile pe nodul primar sunt repetate si pe backup-urile functionale. Dacă nodul primar cade, unul din nodurile de backup este ales pentru a-l înlocui.

TOLERANȚA LA DEFECTE ÎN REȚELE

O particularitate a retelelor este aceea că aproape todeauna există căi multiple care conectează sursa unui mesaj cu destinatia lui. Totodată există noduri de rezervă care pot fi conectate în retea pentru a înlocui unitătile disfuncte.
Literatura mentionează câteva topologii tolerante la defecte:

- Retelele în trepte multiple (*multi-stage*), unele dintre ele suplimentare
- Sitele (*meshes*) interstitiale
- *Crossbar*-ul cu redundante
- Hipercuburile
- Retelele punct-la-punct

Retelele multi-stage lipsite de tolerantă la defecte (retele fluture) sunt alcătuite tipic din comutatoare 2×2, comutatoare cu două intrări si două iesiri.



(a) (b) (c) (d)

Un comutator poate avea una din cele patru setări figurate mai sus:

(a) Directă – linia de intrare superioară conectată la linia superioară de iesire, linia de intrare inferioară conectată la linia inferioară de iesire.
(b) Încrucisată – linia de intrare superioară conectată la linia inferioară de iesire, linia de intrare inferioară conectată la linia superioară de iesire.
(c) Cu difuzare (broadcast) superioară – linia de intrare superioară conectată la ambele linii de iesire.
(d) Cu difuzare (broadcast) inferioară – linia de intrare inferioară conectată la ambele linii de iesire.

Retelele fluture sunt retele în k trepte, cu $k \geq 3$. Au 2^k intrări si 2^k iesiri. Treptele au fiecare 2^{k-1} comutatoare. Conexiunile urmează o anumită regulă recursivă, de la intrare către iesire.
În stratul de intrare, linia superioară a fiecărui comutator este conectată la intrările unui fluture $2^{k-1}\times 2^{k-1}$ si linia de iesire inferioară a fiecărui comutator este conectată la intrările unui alt fluture $2^{k-1}\times 2^{k-1}$. În particular, un fluture cu două straturi, pentru stratul de intrare linia de iesire superioară a fiecărui comutator este conectată la un comutator 2×2 si linia de iesire inferioară este conectată la alt comutator 2×2. Fluturele cel mai simplu, cu un strat constă dintr-un singur comutator 2×2.

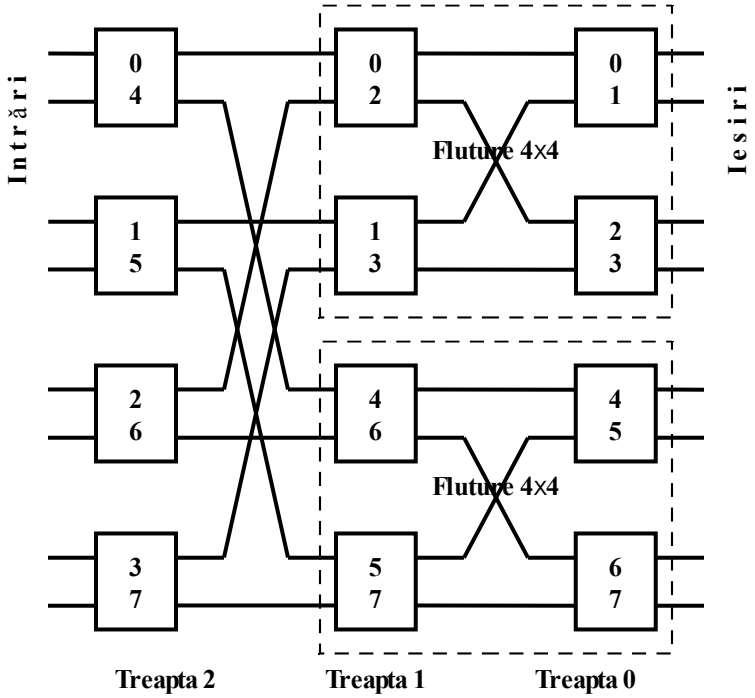


Figura alăturată reprezintă o retea fluture mai complexă, pe care se pot observa unele detalii si se pot preciza unele notatii.

Un comutator din treapta i are liniile numerotate cu numere care diferă cu 2^i. Linia de iesire j a fiecărei trepte merge la linia de intrare j a stratului următor ($j = 0, \ldots, 2^{k-1}$).

Numerele asociate oricărui comutator (*switchbox*), cu exceptia celor din stratul de iesire, sunt ambele de aceeasi paritate (pare sau impare).

O retea fluture nu este tolerantă la defecte: există o singură cale de la oricare dintre intrări la o anumită iesire. Dacă un comutator din stratul i clachează, 2^{k-i} intrări sunt deconectate de la 2^{i+1} iesiri.

Sistemul poate încă opera dar într-o manieră degradată.

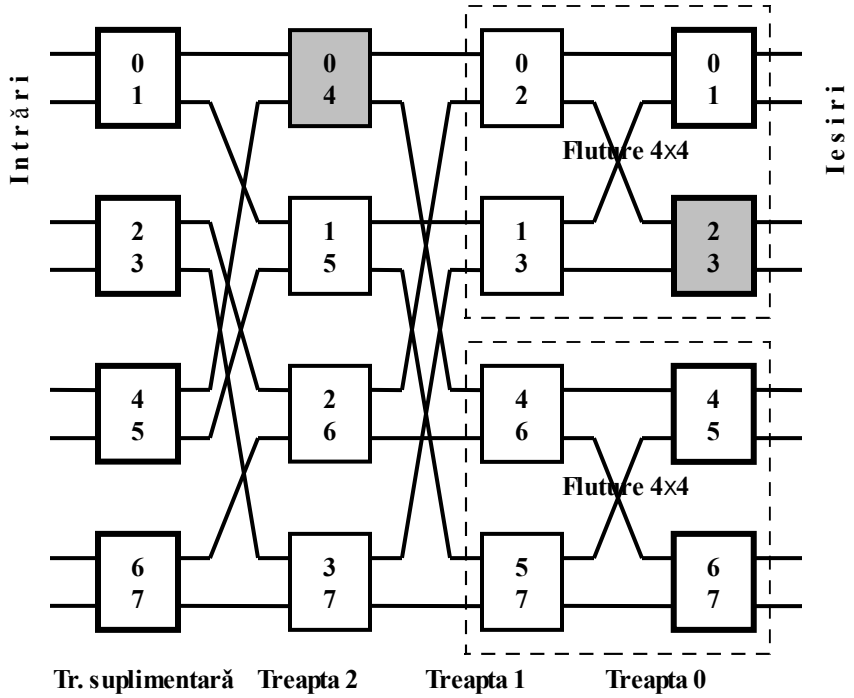
Pentru a crea o tolerantă la defecte se utilizează retele cu trepte suplimentare.

O posibilitate constă în a adăuga o extra-treaptă prin duplicarea treptei de intrare. Este necesară si o multiplexare în scopul *bypass*-ării comutatoarelor din straturile/treptele de la intrare si de la iesire. Prin adoptarea acestei solutii, un comutator disfunct poate fi ocolit prin rutarea pe *bypass*.

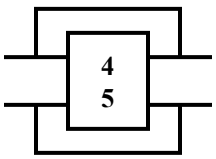
Exemple:

Comutatorul din stratul/treapta 0 cu liniile 2, 3 căzute este duplicat printr-o treaptă suplimentară. Comutatorul disfunct este ocolit (*bypass*-at) cu concursul unui multiplexor.

Comutatorul din stratul 2 cu liniile 0, 4 căzute: extra-stratul este setat astfel ca linia de intrare 0 să fie comutată la linia de iesire 1 si linia de intrare 4 la linia de iesire 5, prin bypassarea cutiei de comutare disfuncte.



Comutatoarele din treapta suplimentară si din treapta ultimă (0), reprezentate cu contur îngrosat, trebuie "citite" fiecare cu posibilitătile de ocolire (bypass), ca înfigura imediat următoare.



Se propune ca exercitiu demonstrarea faptului că reteaua cu o treaptă suplimentară poate rămâne conexă în pofida disfunctiei unei cutii de comutare situată oriunde în sistem.

Măsuri ale siguranței (*dependability*) unei retele cu mai multe straturi.

Retelele cu interconectare în mai multe trepte conectează N procesoare la N unități de memorie într-o arhitectură cu memorie partajată ($N = 2^k$). În prezența elementelor cu defecte, sistemul poate opera într-un mod degradat.

Rezilienţa sistemului în degradare progresivă poate fi măsurată. Iată măsurile uzuale pentru rezilientă:

- Lărgimea de bandă (sau banda de trecere)
- Numărul mediu de căi operationale
- Metrici ale conectivitătii între procesoare si memorii

Toate măsurile sunt functii de timp si presupun că defectele apar si pot fi reparate în intervalul [0, t].

Definiţiile complete si detaliate ale măsurilor enumerate mai sus sunt:

Banda de trecere $BW(t)$ – numărul mediu statistic (*expected*) de procesoare la momentul t, care comunică cu (parte din) memorie.

Conectivitatea $Q(t)$ – numărul mediu statistic (*expected*) de căi procesoare-memorii operationale la momentul t; o cale operatională include un procesor, o memorie si legăturile dintre ele, toate lipsite de defecte.

Un procesor (memorie) este accesibil(ă) (la momentul t) dacă este lipsit(ă) de defecte si este conectat(ă) la cel putin o memorie (un procesor) lipsit(ă) de defecte.

O măsură suplimentară a conectivitătii este cuplul $(A_r(t), A_m(t))$ alcătuit din numărul mediu de procesoare si de memorii accesibile la momentul t.

O altă măsură este dată de perechea $(N_m(t), N_r(t))$ formată din numărul mediu de procesoare (memorii) fără defecte, la care o memorie (un procesor) accesibil este conectat(ă) la timpul t.

Sunt de observat câteva imperfecţiuni ale acestor măsuri.

Banda de trecere, $BW(t)$ nu depinde numai de conditiile retelei ci si de volumul solicitării de memorie din partea procesoarelor.

Conectivitatea $Q(t)$ prin numărul de căi nu spune câte procesoare si câte memorii distincte sunt încă accesibile.

Perechea $(A_r(t), A_m(t))$ nu cuprinde faptul că există o retea de interconectare total conexă si fără defecte $A_r(t) \times A_m(t)$; nu indică nici numărul de memorii fără defecte care sunt conectate în medie la un procesor accesibil.

Prin combinarea măsurilor $Q(t)$ si $(A_r(t), A_m(t))$ se obtine o caracterizare mai completă a sistemului.

Dacă avem în vedere relatiile

$$N_m(t) = Q(t)/A_r(t) \text{ si } N_r(t) = Q(t)/A_m(t)$$

cuplul $(N_m(t), N_r(t))$ reprezintă o margine superioară pentru sistemul operaţional mediu maximal deplin conex la momentul t.

Analiza siguranţei (*dependability*)

O premisă importantă: timpul mediu între căderile (MTBF) componentelor (si posibilele reparatii) este mult mai mare decât durata medie de comunicare între un procesor si o memorie.

O altă premisă importantă: starea componentelor sistemului (cu defecte sau fără) este constantă pentru o perioadă de timp suficient de lungă, atât de lungă cât să permită analiza comportării sistemului într-o stare statistic stationară.

Sistemul este observat la un moment arbitrar t fixat pentru întreaga analiză. Toate măsurile sunt functii de t, inclusiv probabilitătile $p_r(t)$, $p_m(t)$, $p_l(t)$ de bună functionare pentru procesoare, memorii, legături. De regulă, pentru simplitate, timpul t este omis din notatii (se scrie doar p_r, p_m, p_l).
Se notează cu p_q probabilitatea ca un procesor să aibă nevoie de o legătură cu memoria.

Analiza benzii de trecere

Banda de trecere BW este, cum s-a mai spus, numărul mediu statistic (*expected*) de procesoare în comunicare activă cu o (parte din) memorie. Se admite o ipoteză simplificatoare: destinatiile cererilor de memorie din partea procesoarelor sunt independente statistic si uniform distribuite pe cele N memorii.
În conditiile specificate, banda de trecere a retelei este produsul numărului de memorii N cu Ψ_m, probabilitatea ca o memorie dată (de pildă memoria 0) să fie lipsită de defecte si să aibă o cerere la intrare ei.
Probabilitătile Ψ_m se calculează iterativ, urmând calea care duce la acea memorie desemnată de indicele m. Probabilitatea unei cereri pe o legătură de iesire a unui comutator se calculează din probabilitatea ca o solicitare să fi fost acceptată la legăturile de intrare ale acestui comutator (*switch*).
Calculul benzii de trecere urmează în linii mari schema explicată în continuare.
O legătură este în starea $X = 1$ ($X = 0$) dacă are (nu are) o solicitare pentru memoria specificată; o legătură defectă este în starea $X = 0$.
Atribuirea de numere celor $k + 1$ straturi ale retelei ($k = \log_2 N$).

- Stratul 0 este ultimul strat; legăturile de iesire sunt conectate la memorii.
- Stratul k este primul si preia iesirile procesoarelor.

X^i, Y^i sunt iesirile unui comutator din stratul/treapta i.
X^{i+1}, Y^{i+1} sunt intrările unui comutator din stratul i, în acelasi timp iesiri a două dintre comutatoarele (diferite) din stratul $i + 1$.

Banda de trecere în cazul rețelelor fără redundanțe

Solicitările de memorie sunt distribuite uniform între memorii; o solicitare care soseste este rutată la oricare legătură de iesire din cele două ale unui comutator, cu probabilitate egală (0,5). În calculul probabilitătii $\Pr(X^i = 1)$, este suficient a lua în considerare numai una din legăturile de iesire.
O cerere către un modul de memorie poate atinge legătura de iesire a unui comutator pe oricare dintre cele două legături de intrare

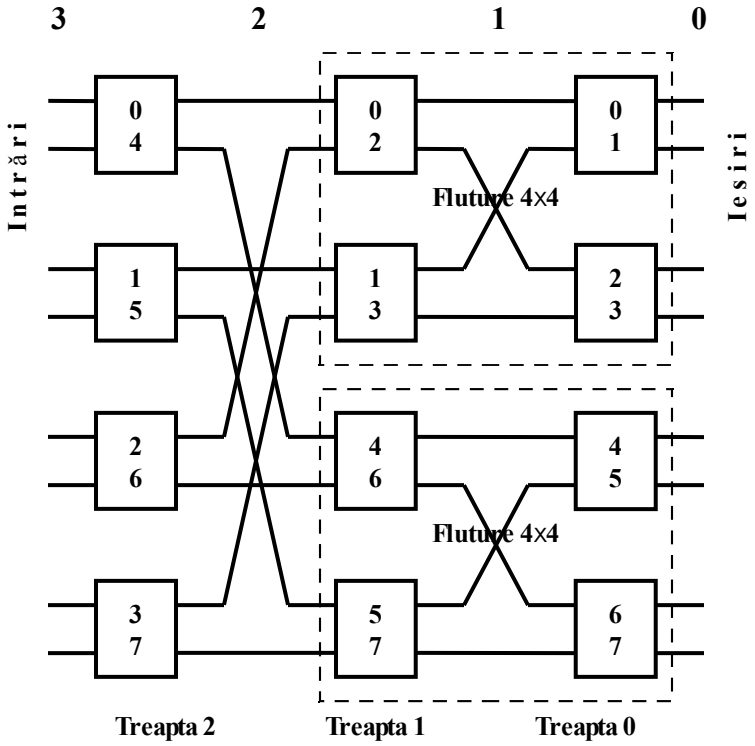
$$\Pr(X^i = 1) = \Sigma\Sigma_{u,v=0,1}\Pr(X^i = 1/X^{i+1} = u,\ Y^{i+1} = v)\Pr(X^{i+1} = u,\ Y^{i+1} = v) =$$
$$= 0.\Pr(X^{i+1} = 0,\ Y^{i+1} = 0)) + (1/2)p_l\Pr(X^{i+1} = 0,\ Y^{i+1} = 1)) +$$
$$+ (1/2)p_l\Pr(X^{i+1} = 1,\ Y^{i+1} = 0)) + (p_l + (1/4)p_l^2)\Pr(X^{i+1} = 1,\ Y^{i+1} = 1))$$

Sunt luate în considerare numai defectele legăturilor de intrare. Defectele legăturilor de iesire sunt considerate defecte ale legăturilor de intrare ale etapei următoare.
Stările intrărilor unui comutator sunt presupuse a fi independente statistic:

$$\Pr(X^i = u,\ Y^i = v) = \Pr(X^i = u)\ \Pr(Y^i = v)$$

$$\Pr(X^i = 0) = \Pr(Y^i = 0) = 1 - \Pr(X^i = 1)$$

După câteva prelucrări algebrice nu foarte complicate se poate arăta că $\Pr(X^k = 1) = p_q p_r$.



Probabilitatea $\Pr(X^0 = 1)$ se calculează recursiv.
Memoria si legătura ei de intrare pot fi fără defecte: probabilitatea unei astfel de stări este

$$\Psi_m = \Pr(X^0 = 1) p_i p_m$$

si în final

$$BW = N\,\Psi_m$$

Conectivitatea rețelelor de interconectare fără redundanțe

Cum s-a definit, Q este numărul mediu de căi operationale pentru perechi procesor-memorie conectate. În lipsa oricăror redundanțe există exact o cale între un procesor si o memorie.

În cuvinte, conectivitatea Q este produsul numărului de perechi procesor-memorie cu probabilitatea existentei unei căi fără defecte. Probabilitatea aceasta este $p_r p_l^{k+1} p_m$, cu $k + 1$ numărul de legături pe cale, adică numărul de trepte + 1 ($k = \log_2 N$).
Deoarece numărul de căi procesor-memorie este N^2, rezultă

$$Q = N^2 p_r p_l^{k+1} p_m.$$

Calculul măsurilor adiționale pentru rețelele de interconectare fără redundanțe

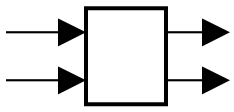
Am notat cu A_r numărul mediu de procesoare accesibile. A_r este produsul numărului de procesoare N cu probabilitatea ϕ_r ca un procesor (de pildă procesorul 0) să fie accesibil.
O legătură este în starea $X = 0$ ($X = 1$) dacă toate (nu toate) căile de la procesor la memorie sunt defecte.

- O cale defectă este o cale cu cel putin o legătură defectă.
- O legătură defectă este în starea $X = 0$.

Numerotarea treptelor se face si de această dată de la k la 0; X^i exprimă starea legăturii în treapta i. În particular, $\Pr(X^0 = 1) = p_m$; $\Pr(X^0 = 0) = 1 - p_m$.
O ecuatie recursivă

$$\Pr(X^{i+1} = 1) = p_l[1 - \Pr(X^i = 0)^2]$$

este suportul evaluărilor curente.



În cele din urmă

$$\phi_r = p_r p_l \Pr(X^k = 1)$$

si

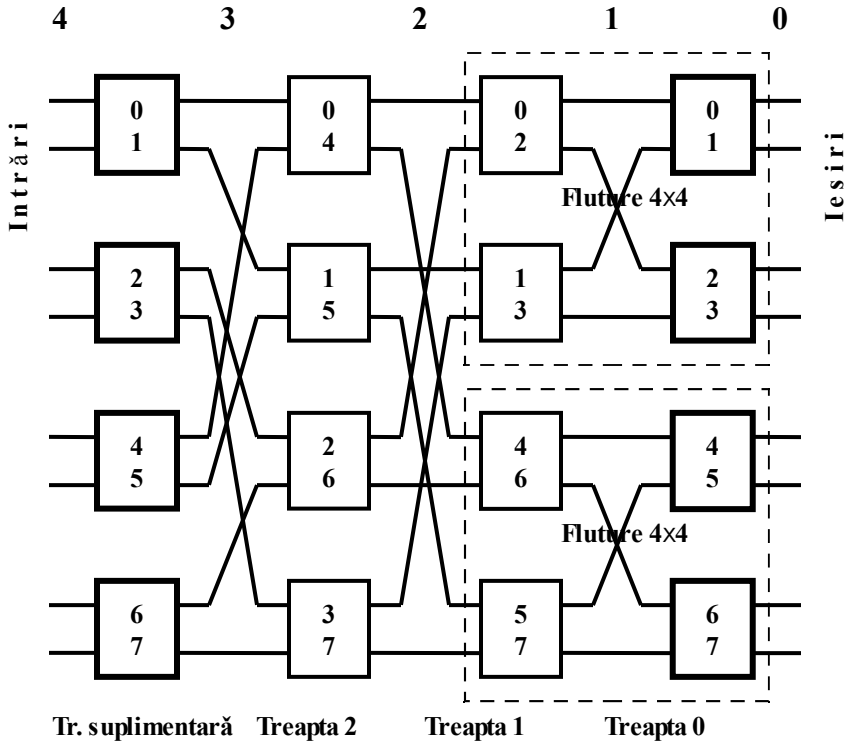
$$A_r = N \phi_r$$

Numărul mediu de memorii accesibile, A_m se obtine pe o cale similară, prin schimbarea probabilitătii p_r cu probabilitatea p_m.
Urmează un exemplu numeric.
Calculul benzii de trecere, în ipoteza unor legături fără defecte, în cazul $N = 8$, $k = 3$, pentru un moment t fixat.
$p_r = 0{,}8$; $p_m = 0{,}9$; $p_l = 1$ (legături fără defecte); $p_q = 0{,}7$.
Calculul benzii de trecere:
$\Pr(X^3 = 1) = p_q p_r = 0{,}56$
$\Pr(X^2 = 1) = 0{,}56 - 0{,}25 \times 0{,}562 = 0{,}536$
$\Pr(X^1 = 1) = 0{,}536 - 0{,}25 \times 0{,}5362 = 0{,}464$
$\Pr(X^0 = 1) = 0{,}464 - 0{,}25 \times 0{,}4642 = 0{,}41$
$BW = 0{,}41 N p_m = 0{,}41 \times 8 \times 0{,}9 = 2{,}95$

Calculul conectivității si al măsurilor suplimentare:
$Q = N^2 \times 0{,}8 \times 0{,}9 = 0{,}72N^2 = 46{,}08$
$A_r = Np_r[1 - (1 - p_m)^N] = 0{,}8N(1 - 0{,}1^N) \approx 0{,}64$
$A_m = Np_m[1 - (1 - p_r)^N] = 0{,}9N(1 - 0{,}2^N) \approx 0{,}72$
$N_r = Q/A_m = 64$
$N_m = Q/A_r = 72$
Asupra acestor ultime rezultate sugerăm cititorului o discuție.

Retele fluture si retele fluture cu trepte suplimentare

În reteaua fluture fără redundante, cele două intrări în orice comutator sunt considerate independente statistic. Într-o retea cu o treaptă în plus sunt câte două căi care conectează orice pereche procesor-memorie. Legăturile sunt de data aceasta dependente si ecuatiile date mai sus nu mai sunt valide.
Asa cum s-a mai spus, primul si ultimul strat au multiplexoare si demultiplexoare pentru care analiza este diferită de aceea a etajelor interioare.
Sunt patru legături care duc la două comutatoare, două perechi sunt independente, desi legăturile din aceeasi pereche sunt dependente.



Exemplu: legăturile de iesire 0 si 1 din etajul 2 sunt dependente (procesoarele 0 si 1 trimit solicitări către memoria 0 prin ambele); legăturile 2 si 3 sunt si ele dependente; perechile 0, 1 si 2, 3 sunt însă independente.

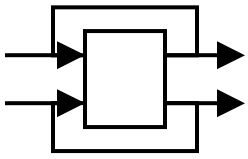
Banda de trecere pentru o rețea cu trepte suplimetare

Banda de trecere (*BW*), reamintim, este numărul mediu de procesoare care comunică activ cu (o parte din) memorie si este acelasi lucru cu numărul mediu de memorii care comunică activ cu vreun procesor. Banda de trecere se obtine ca produsul numărului de memorii N cu Ψ_m, probabilitatea ca o memorie dată (de pildă memoria 0) să fie fără defect si să aibă o solicitare la intrarea ei.
Ca si altădată, Ψ_m se calculează iterativ, urmând o cale de la procesor care duce la o anumită meorie.
Legătura este în starea 1 (0) dacă ea are (nu are) o solicitare de memorie; o legătură defectă este în starea 0.
Calculul benzii de trecere pentru reteaua din figură urmează pasii prezentati imediat.
Sunt $k + 2$ trepte numerotate $k + 1$, k, …, 0 (cu $k = \log_2 N$). Se notează

- X^i, Y^i – starea celor două legături de iesire din etajul i
- X^{i+1}, Y^{i+1}, Z^{i+1}, W^{i+1} – starea intrărilor pe legăturile din etajul i, totuna cu legăturile de iesire pentru etajul $i + 1$.

Probabilitatea ca o intrare din etajul i să aibă solicitare este calculată pe baza probabilității ca o solicitare să fie acceptată la legăturile de intrare.
Pentru primul etaj ($k + 1$ – etajul procesorului) se poate scrie

$$\Pr[X^{k+1} = 1] = p_q p_r;\ \Pr[X^{k+1} = 0] = 1 - \Pr[X^{k+1} = 1]$$



Pentru etajul k (procesoarele sunt independente statistic):

$$\Pr[(X^k, Y^k) = (0, 0)] =$$
$$= (\Pr[X^{k+1} = 0])^2 + q_l^4(\Pr[X^{k+1} = 1])^2 + q_l^3 2\Pr[X^{k+1} = 0]\Pr[X^{k+1} = 1]$$
$$\Pr[(X^k, Y^k) = (0, 1)] =$$
$$= (1 - q_l^3)\Pr[X^{k+1} = 0]\Pr[X^{k+1} = 1] + (1 - q_l^2)q_l^2(\Pr[X^{k+1} = 1])^2$$
$$\Pr[(X^k, Y^k) = (1, 0)] = \Pr[(X^k, Y^k) = (0, 1)]$$
$$\Pr[(X^k, Y^k) = (1, 1)] = (1 - q_l^2)^2(\Pr[X^{k+1} = 1])^2$$

Treptele interne în retelele cu trepte suplimetare

Expresiile de mai devreme au presupus că o solicitare este trimisă mai întâi prin conexiunea directă si se foloseste conexiunea încrucisată numai dacă cea directă este disfunctă.

- Protocol diferit: conexiunile directă si încrucisată cu probabilităti egale, expresiile pentru probabilităti vor fi diferite.

Pentru etajele interioare ($i = k - 1, \ldots, 1$):

$$\Pr[(X^i, Y^i) = (u, v)] =$$

$$= \sum_{(s_0,s_1,s_2,s_3)=(0,0,0,0)}^{(s_0,s_1,s_2,s_3)=(1,1,1,1)} \Pr[(X^{i+1}, Y^{i+1}, Z^{i+1}, W^{i+1}) = (s_0, s_1, s_2, s_3)] \cdot$$

$$\cdot \Pr[(X^i, Y^i) = (u,v) \mid (X^{i+1}, Y^{i+1}, Z^{i+1}, W^{i+1}) = (s_0, s_1, s_2, s_3)] =$$

$$= \sum_{(s_0,s_1,s_2,s_3)=(0,0,0,0)}^{(s_0,s_1,s_2,s_3)=(1,1,1,1)} \Pr[(X^{i+1}, Y^{i+1}) = (s_0, s_1)] \Pr[(Z^{i+1}, W^{i+1}) = (s_2, s_3)] \cdot$$

$$\cdot \Pr[X^i = u \mid (X^{i+1}, Z^{i+1}) = (s_0, s_2)] \Pr[Y^i = v \mid (Y^{i+1}, W^{i+1}) = (s_1, s_3)]$$

pentru $u, v = 0, 1$.

Numai probabilitătile combinate (*joint*) ale celor două legături sunt necesare. Acestea pot fi calculate recursiv de la etajul $k + 1$ (etajul procesor) la etajul 0 (etajul de memorare).

Etajul 0 include demultiplexoare

$$\Pr[X^0 = 1 | (X^1, Y^1) = (0, 0)] = 0$$

$$\Pr[X^0 = 1 | (X^1, Y^1) = (0, 1)] = (1/2)p_l$$

$$\Pr[X^0 = 1 | (X^1, Y^1) = (1, 0)] = (1/2)(1 - q_l^2)$$

$$\Pr[X^0 = 1 | (X^1, Y^1) = (1, 1)] = (1/2)(3p_l - p_l^2) - (1/4)p_l(1 - q_l^2)$$

$$\Psi_m = \Pr(X^0 = 1)p_l p_m$$

În final:

$$BW = N\Psi_m$$

Conectivitatea pentru o rețea cu trepte suplimetare

Q este produsul numărului de perechi procesor-memorie N^2 cu probabilitatea ca între componentele perechii să existe cel putin o cale fără defecte.

Fiecare pereche procesor-memorie este conectată prin două căi disjuncte (se-ntelege, cu exceptia celor două capete).

Probabilitatea ca cel putin o cale să fie fără defecte este egală cu probabilitatea ca prima cale să fie fără defecte adunată cu probabilitatea ca cealaltă cale să fie fără defecte din care trebuie scăzută probabilitatea ca ambele căi să fie fără defecte.

Probabilitatea poate asuma una din cele două expresii (a se compara calea între procesorul 0 si memoria 0 cu calea între procesorul 0 si memoria 1).

Calculul conectivitătii urmează pasii de mai jos.

Pentru căile dintre procesorul 0 si memoria 0:

$$\Pr(\text{cel putin o cale este fără defecte}) = \Pr(0, 0) =$$
$$= 2p_r(1 - q_l^2)p_l^{k+1}p_m - p_r p_l^{2k+2}(1 - q_l^2)^2 p_m$$

Pentru căile dintre procesorul 0 si memoria 1:

$$\Pr(\text{cel putin o cale este fără defecte}) = \Pr(0, 1) =$$
$$= p_r(1 - q_l^2)p_l^k(1 - q_l^2)p_m + p_r p_l^{k+2}p_m - p_r p_l^{2k+2}(1 - q_l^2)^2 p_m$$

Jumătate din perechile procesor-memorie urmează valoarea Pr(0, 0) si cealaltă jumătate urmează valoarea Pr(0, 1).

$$Q = [\Pr(0, 0) + \Pr(0, 1)]N^2/2$$

Măsuri adiționale pentru o rețea cu trepte suplimetare

A_r si A_m sunt numărul mediu de procesoare accesibile, respectiv numărul mediu de memorii accesibile.

ϕ_r (ϕ_m) sunt probabilitătile ca un procesor (o memorie) dat(ă) să fie conectat(ă) la cel putin o memorie (un procesor).

Pentru calcularea lui A_r se face aceeasi descriere de stări: legătura este în starea $X = 0$ ($X = 1$) dacă toate (nu toate) căile de la procesor la memorii sunt defecte. O cale defectă este o cale care contine cel putin o legătură defectă.

O legătură defectă este în starea $X = 0$.

Numerotarea etajelor se mentine, $k + 1$ (procesoarele) la 0 (memoriile).

Dacă X^i descrie starea legăturii din etajul i

$$\phi_r = p_r p_l \Pr(X^{k+1} = 1)$$

si

$$A_r = N\,\phi_r$$

Urmează calculul măsurii A_r.

$\Pr(X^i = 1)$ se calcuează recursiv de la treapta 0 la treapta $k + 1$.

X^i, Y^i notează si acum starea celor două legături din etajul i.

Pentru etajul 0:

$$\Pr(X^0 = 1) = p_m \text{ si } \Pr(X^0 = 0) = 1 - p_m.$$

Pentru etajul 1:

$$\Pr[(X^1, Y^1) = (0, 0)] = \{\Pr[X^0 = 0]\}^2 + 2\Pr[X^0 = 0]\Pr[X^0 = 1]q_l^3 +$$
$$+ \{\Pr[X^0 = 1]\}^2 q_l^6$$
$$\Pr[(X^1, Y^1) = (0, 1)] = \Pr[X^0 = 0]\Pr[X^0 = 1][q_l(1 - q_l^2) + q_l^2 p_l] +$$
$$+ \{\Pr[X^0 = 1]\}^2 q_l^3(1 - q_l^3)$$
$$\Pr[(X^1, Y^1) = (1, 0)] = \Pr[(X^1, Y^1) = (0, 1)]$$
$$\Pr[(X^1, Y^1) = (1, 1)] = 2\Pr[X^0 = 0]\Pr[X^0 = 1]p_l(1 - q_l^2) +$$
$$+ \{\Pr[X^0 = 1]\}^2(1 - q_l^3)^2$$

Pentru stările 2, …, k, variabilele X^{i-1}, Y^{i-1}, Z^{i-1}, W^{i-1} exprimă starea celor patru legături din etajul $i - 1$.

$$\Pr[(X^i, Y^i) = (u, v)] = \sum_{s_0 \ldots s_3 = 0,0,0,0}^{1,1,1,1} \Pr[(X^{i-1}, Y^{i-1}) = (s_0, s_1)] \cdot$$
$$\cdot \Pr[(Z^{i-1}, W^{i-1}) = (s_2, s_3)] \cdot \Pr[X^i = u \mid (X^{i-1}, Z^{i-1}) = (s_0, s_2)] \cdot$$
$$\cdot \Pr[Y^i = v \mid (Y^{i-1}, W^{i-1}) = (s_1, s_3)]$$

Probabilitățile conditionate sunt:

$$\Pr[X^i = 0 \mid (X^{i-1}, Z^{i-1}) = (0,0)] = 1$$
$$\Pr[X^i = 0 \mid (X^{i-1}, Z^{i-1}) = (0,1)] = q_l$$
$$\Pr[X^i = 0 \mid (X^{i-1}, Z^{i-1}) = (1,0)] = q_l$$
$$\Pr[X^i = 0 \mid (X^{i-1}, Z^{i-1}) = (1,1)] = q_l^2$$

Pentru etajul suplimentar $k + 1$:

$$\Pr[X^{k+1} = 0] = \Pr[(X^k, Y^k) = (0,0)] + \Pr[(X^k, Y^k) = (0,1)](q_l + q_l^2) +$$
$$+ \Pr[(X^k, Y^k) = (1,1)]q_l^3$$
$$\Pr[X^{k+1} = 1] = 1 - \Pr[X^{k+1} = 0]$$
$$\phi_r = p_r p_l \Pr[X^{k+1} = 1]$$

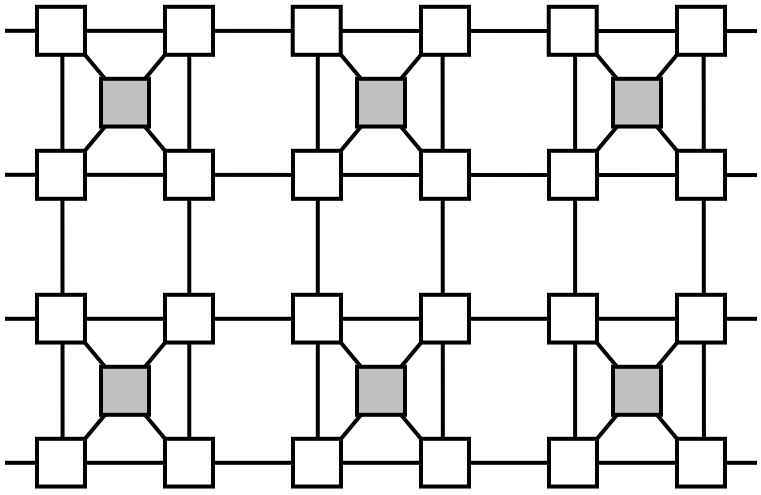
În final

$$A_r = N \phi_r$$

Măsura A_m se calculează similar înlocuind p_r cu p_m.

Plasa (*mesh*) interstițială

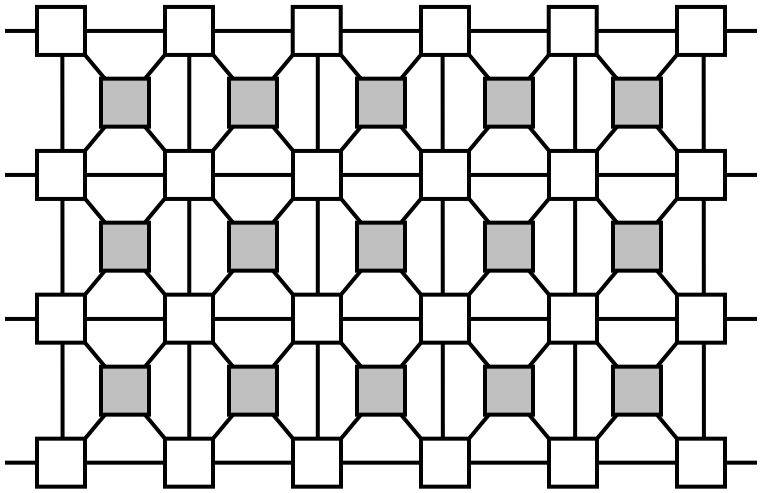
O retea conventională de genul plasă rectangulară bidimensională este incapabilă să tolereze vreun nod defect.
Redundanta interstitială (1, 4) este ilustrată de figura alăturată.



Nodurile umbrite sunt noduri de rezervă

Se observă câte un nod de rezervă adăugat pentru a înlocui oricare din cele patru noduri vecine, care a clacat. Asadar, fiecare nod primar are un singur nod de rezervă, fiecare nod suplimentar este rezervă pentru patru noduri primare. Overheadul de redundantă este 25%.

Avantajul principal rezidă în proximitatea fizică a nodului de rezervă fată de nodurile primare pe care le poate înlocui. Aceasta poate reduce penalitatea de întârziere datorată utilizării unei rezerve.
Redundanta interstitială se practică uneori într-o formă diferită. Iată imediat, în figura alăturată, redundanta interstitială (4, 4).
Un nod primar are în această schemă patru noduri de rezervă si fiecare nod suplimentar este rezervă pentru patru noduri primare. O astfel de structură are un nivel de tolerantă mai ridicat dar si overheadul de redundantă este mai ridicat, este de cca. 100%.



Nodurile umbrite sunt noduri de rezervă

Fiabilitatea plasei (*mesh*) cu redundanță interstițială (1, 4)

Plasa este de dimensiunile $m \times n$, cu m si n numere pare. Reteaua este alcătuită din clustere de patru noduri primare cu un nod de rezervă. Reteaua (*mesh*) are în total $mn/4$ astfel de clustere.
Fie $R(t)$ fiabilitatea unui nod primar sau de rezervă.
Fiabilitatea unui cluster este

$$R_{cluster}(t) = R^5(t) + 5R^4(t)[1 - R(t)]$$

iar fiabilitatea unei plase (mesh) de $m \times n$ este

$$R_{plasa}(t) = [R_{cluster}(t)]^{mn/4}$$

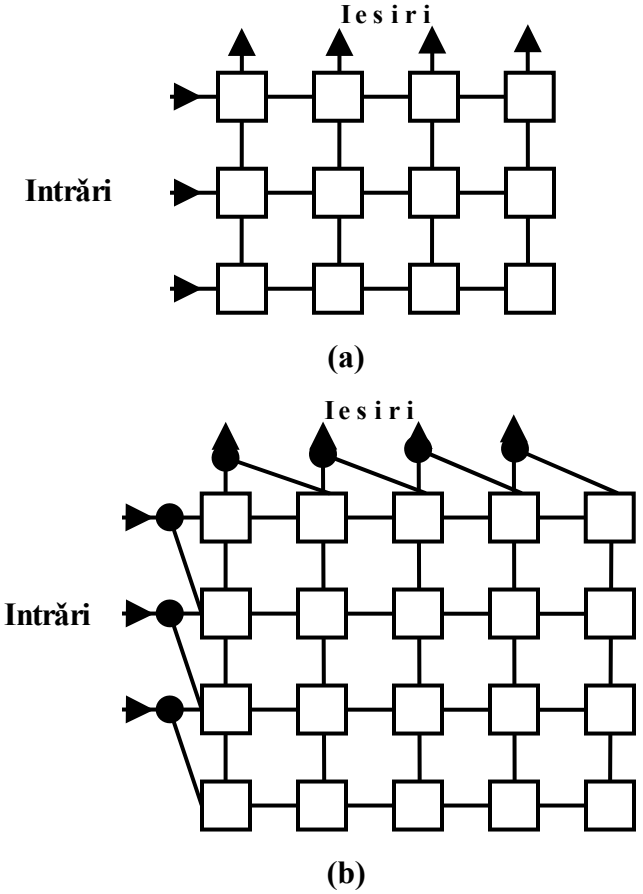
Dacă în cazul redundantei interstitiale (1, 4), pentru functia de fiabilitate există această expresie, pentru schemele cu redundantă interstitială (4, 4) nu există un algoritm simplu pentru calculul fiabilitătii.

Retele *crossbar* fără redundante

Figura alăurată arată o retea *crossbar* 3×4 (trei intrări si patru iesiri). În general, o retea crossbar $m \times n$ are n intrări, m iesiri si mn comutatoare. Comutatoarele leagă toate perechile alcătuite dintr-o intrare si o iesire. Reteaua crossbar nu este tolerantă la defecte. Disfunctia oricărui comutator deconectează anumite perechi de noduri.

Retele *crossbar* cu redundante

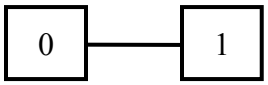
Se adaugă redundante pentru a face reteaua tolerantă la defecte. Pentru aceasta se adaugă, de pildă, o linie si o coloană de comutatoare. Conexiunile de intrare si de iesire sunt multiplicate prin faptul că fiecare intrare poate fi trimisă pe două linii si fiecare iesire poate fi obtinută de la două coloane.
Dacă un comutator se defectează, atunci linia si coloana de care acesta apartine sunt înlocuite de linia si coloana de rezervă (v.figura).



Retele *crossbar* fără redundante (a) si cu redundante (b)

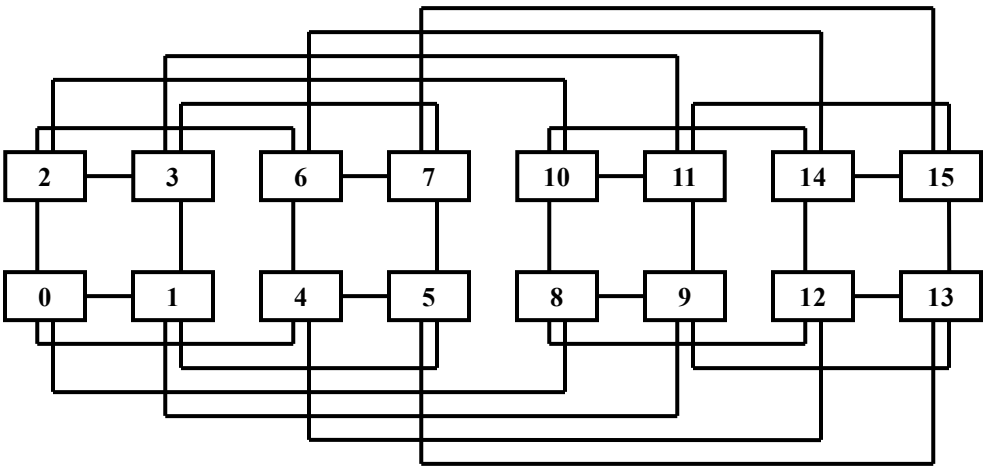
Rețele de tip hipercub

Cu H_n se notează o retea de tip hipercub n-dimensională care are 2^n noduri. Un hipercub 0-dimensional are un singur nod. Un hipercub H_n se construieste prin conectarea nodurilor corespondente din două retele H_{n-1}, două hipercuburi cu o dimensiune mai putin. Muchiile adăugate pentru a conecta noduri corespondente sunt numite muchii de dimensiune $(n - 1)$.



Muchie de dimensiune 0

Exemple de hipercuburi sunt date în figura alăturată. În hipercubul H_4 din figură se disting cu usurintă hipercuburi de dimensiuni inferioare si muchii de diverse dimensiuni.



Exemple diverse rezultate din lectura figurii:
Muchie de dimensiune 0: 8-9 (diferenta între numerele purtate de noduri: $1 = 2^0$).
Muchie de dimensiune 1: 4-6 (diferenta între numerele purtate de noduri: $2 = 2^1$).
Muchie de dimensiune 2: 10-14 (diferenta între numerele purtate de noduri: $4 = 2^2$).
Muchie de dimensiune 3: 3-11 (diferenta între numerele purtate de noduri: $8 = 2^3$).
Hipercuburi H_0: oricare nod.
Hipercuburi H_1: oricare pereche de noduri dintre următoarele: (0, 1), (2, 3), (4, 5), (6, 7), (8, 9), (10, 11), (12, 13), (14, 15).
Hipercuburi H_2: (0, 1, 2, 3), (4, 5, 6, 7), (8, 9, 10, 11) si (12, 13, 14, 15).

Hipercuburi H_3: (0, 1, 2, 3, 4, 5, 6, 7) si (8, 9, 10, 11, 12, 13, 14, 15).

Rutarea în hipercuburi

Pentru a simplifica rutarea se foloseste o numerotare specială. Numerele sunt exprimate în binar si dacă nodurile *i* si *j* sunt conectate de o muchie de dimensiune *k*, numerele pentru *i* si *j* diferă prin bitii de pe pozitia *k*.
Exemplu: nodurile 0000 si 0010 diferă numai prin bitul de pe pozitia 2^1; ele sunt conectate printr-o muchie de dimensiune 1.
Alt exemplu: un pachet trebuie să se deplaseze de la nodul $14 = 1110_2$ la nodul $2 = 0010_2$ într-o retea H_4. Rutările posibile sunt:

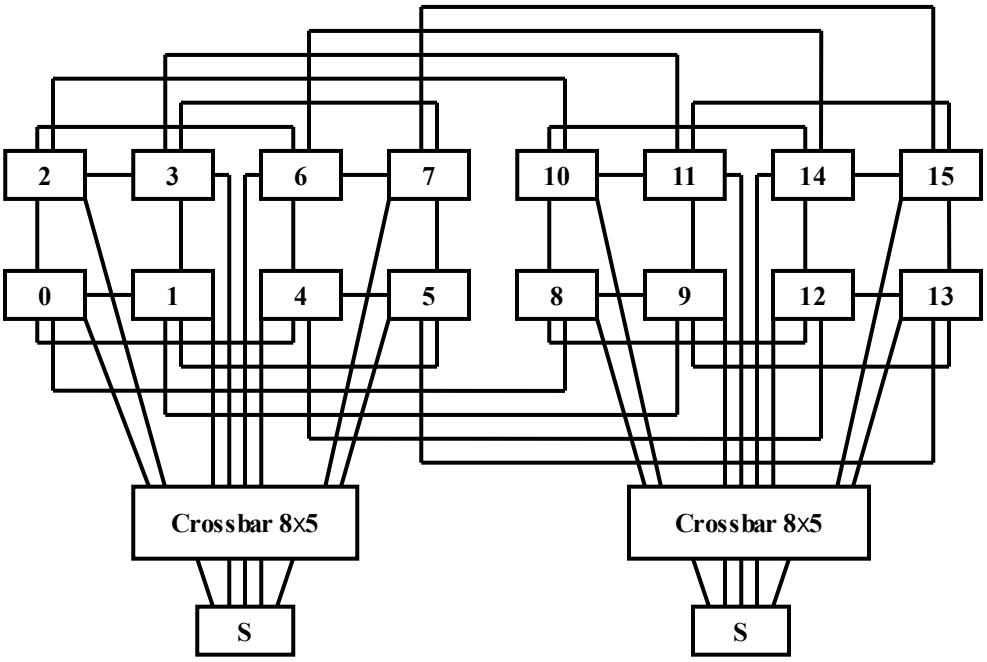
- $1110 \rightarrow 0110$ (dimensiune 3) $\rightarrow 0010$ (dimensiune 2)
- $1110 \rightarrow 1010$ (dimensiune 2) $\rightarrow 0010$ (dimensiune 3)

Rutarea în cazul general

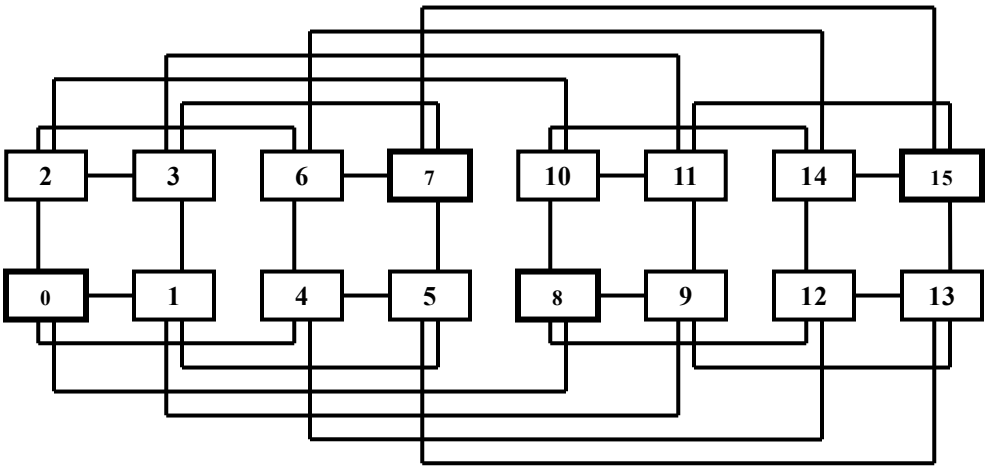
Distanta între sursă si destinatie este în general numărul de biti diferiti în cele două adrese (distanta Hamming). Transferul de la nodul *X* la nodul *Y* poate fi făcut prin trecerea câte o dată pe fiecare din dimensiunile prin care sursa si destinatia diferă.
Dacă adresele sunt $X = x_{n-1} \ldots x_0$ si $Y = y_{n-1} \ldots y_0$, se definesc bitii $z_i = x_i \oplus y_i$ ($i = 0, \ldots, n-1$) cu $\oplus$ operatorul "sau-exclusiv".
Pachetul trebuie să traverseze o muchie în fiecare dimensiune pentru care $z_i = 1$.

Toleranța la defecte în rețelele hipercub

Pentru $n \geq 2$, un hipercub H_n poate tolera disfunctii ale legăturilor deoarece există căi multiple de la orice sursă la orice destinatie.
Disfunctiile nodurilor pot însă compromite operatia. O modalitate de ameliorare a situatiei constă în cresterea numărului de porturi de comunicare ale fiecărui nod de la *n* la *n* + 1 si conectarea acestor porturi suplimentare prin legături aditionale la unul sau mai multe noduri de rezervă.
Exemplu: Se pot adăuga două noduri de rezervă, fiecare din acestea fiind o rezervă pentru 2^{n-1} noduri ale unui subcub H_{n-1}.
Nodurile de rezervă ar putea necesita 2^{n-1} porturi. Numărul de porturi poate fi redus prin utilizarea unor comutatoare *crossbar* ale căror iesiri sunt conectate la nodul de rezervă corespunzător. Numărul de porturi ale nodului de rezervă este redus la *n* + 1, acelasi ca pentru toate celelalte noduri.
Figura alăturată arată un hipercub H_4 cu două noduri de rezervă.



O metodă diferită de tolerare a defectelor constă în duplicarea procesoarelor din câteva (putine) noduri selectate. Fiecare procesor aditional este rezervă pentru oricare dintre procesoarele din nodurile vecine. În exemplul din figura următoare, nodurile 0, 7, 8, 15 ale unui hipercub H_4 sunt modificate prin duplicare (reprezentate îngrosat).



Fiecare nod are acum o rezervă la distantă nu mai mare de 1. Înlocuirea unui procesor defect cu unul din rezervă produce, desigur, o întârziere suplimentară în comunicare.

Rutarea în hipecuburi cu defecte

Algoritmul de rutare trebuie modificat pentru a ocoli nodurile sau legăturile defecte. Ideea de bază se poate formula astfel: se listează dimensiunile pe care un pachet trebuie să meargă si se parcurg acestea una câte una. Pe măsură ce muchiile sunt parcurse si marcate/eliminate (*crossed off*) din listă, dacă din cauza unui nod sau din cauza unei legături disfuncte legătura dorită nu este disponibilă se alege o altă muchie din listă (dacă există una) pentru continuarea parcursului; dacă pachetul atinge un anumit nod în care găseste toate dimensiunile din lista sa căzute, el revine (*backtracks*) la nodul anterior si încercarea continuă.
Algoritmul formal de rutare utilizează următoarele notatii:
TD – lista dimensiunilor pe care circulă mesajul, în ordinea parcurgerii.
TD^R – acelasi lucru în ordine inversă (*reversed*).
$\oplus_{i=1}^{k}$ – operatia sau-exclusiv executată de *k* ori, secvential.
Exemplu: $\oplus_{i=1}^{3} a_i$ înseamnă $(a_1 \oplus a_2) \oplus a_3$.
D – nod destinatie, *S* – nod sursă, $d = D \oplus S$ ($\oplus$ – operatia sau-exclusiv se execută bit-cu-bit pe bitii corespondenti din adresele binare *D* si *S*).
SC(*A*) – multimea de noduri vizitată pe un parcurs, pe fiecare din dimesiunile listate în multimea *A*.
Exemplu: la nodul 0010 – *SC*(1, 3) = {0000, 1000}.
e_n^i – un vector de *n* biti care are 1 în pozitia bitului *i* si 0 în rest.
Exemplu: $e_3^2 = 100$.
Pachetele sunt presupuse a consta în:

(i) *d*; $d = D \oplus S$
(ii) mesajul transmis ("încărcătura")
(iii) lista de dimensiuni vizitate deocamdată – *TD*

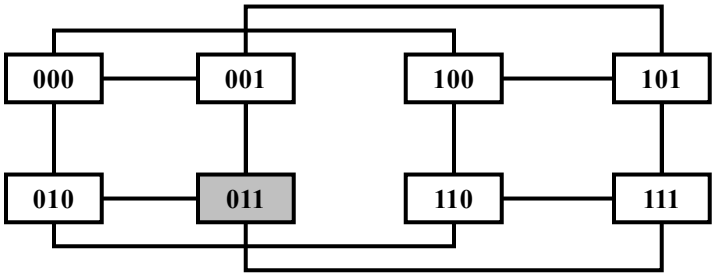
θ – operatia de adăgare (*append*). Scrierea $TD \; \theta \; x$ înseamnă adăugarea la finalul listei *TD* a lui *x*.
transmit(*j*) – rutina de trimitere a pachetului ($d \oplus e^j$, mesaj, $TD \; \theta \; x$) pe legătura *j*-dimensională de la nodul curent.

Algoritm de rutare pentru hipercuburi cu defecte

```
if (d == 0…0)
        destinatia a fost atinsă; exit
else
        for j = 0 to (n – 1) step 1 do {
                if (d_j == 1) && (legătura în dimensiunea j din acest nod este fără
                defect) && (e_v^j ∉ SC|TD^R) {
                transmit(j)
                exit
                }
        }
```

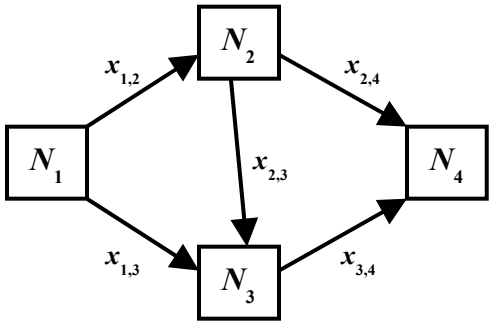
endif
if (există o legătură fără defect în $SC|TD^R$)
fie h o astfel de legătură
else {
$g = \max(m: \oplus_{i=1}^{m} e^{TD^R}(i) = =0...0)$
if (g == numărul de elemente din $SC(TD)$) {
nu există o cale
exit
else
h = elementul al (g + 1)-lea din TD^R
endif
transmit(h)
end

Exemplu pe hipercubul H_3:
H_3 cu nodul defect 011.
Nodul 000 trebuie să transmită un pachet la 111.
La 000, d = 111, trimite mesajul pe dimensiunea 0, la nodul 001.
La 001, d = 110 si TD = (0), tentative la muchiile de dimensiune 1: imposibil.
Bitul 2 din d este tot 1. Se verifică si se stabileste că muchia de dimensiune 2 la 101 este disponibilă, mesajul este trimis la 101 si apoi la 111.
Exercitiu: Ce se întâmplă dacă sunt căzute nodurile 001 si 101?



Fiabilitatea rețelelor punct-la-punct

Retelele nu sunt în mod necesar structuri regulate si de cele mai multe ori există mai multe căi între două noduri.
Se defineste *fiabilitatea terminală*: probabilitatea ca să existe o cale operațională între două noduri anumite, fiind date probabilitătile disfunctiilor pe fiecare legătură.
Exemplu: să se calculeze fiabilitatea terminală pentru perechea sursă-destinatie $N_1 - N_4$ (v.figura).



Sunt trei căi de la N_1 la N_4

- $P_1 = (x_{1,2}, x_{2,4})$
- $P_2 = (x_{1,3}, x_{3,4})$
- $P_3 = (x_{1,2}, x_{2,3}, x_{3,4})$

$p_{i,j}$ ($q_{i,j}$) – probabilitatea ca legătura $x_{i,j}$ să fie bună (respectiv defectă).

Nodurile sunt presupuse a fi fără defecte. Dacă nu aceasta este situatia, probabilitatea disfunctiei lor este încorporată în aceea a legăturilor care pleacă din noduri.

Multimea de căi trebue prelucrată pentru a obtine o multime echivalentă alcătuită din evenimente mutual exclusive, altminteri unele evenimente ar putea fi luate în calcul de mai multe ori.

Evenimente mutual exclusive în cazul în discutie:

(I) calea P_1 functională;

(II) calea P_2 functională si calea P_1 disfunctă;

(III) calea P_3 functională, căile P_1 si P_2 disfuncte.

Dacă numim reteaua din figură "punte" (denumire legată de forma si topologia ei), atunci fiabilitatea legăturii $N_1 - N_4$ este:

$$R_{\text{punte}} = p_{1,2}p_{2,4} + p_{1,3}p_{1,4}(1 - p_{1,2}p_{2,4}) + p_{1,2}p_{2,3}p_{3,4}(q_{1,3}q_{2,4})$$

Calculul fiabilității terminale

Pentru a calcula fiabilitatea terminală a unei retele cu m căi P_1, …, P_m de la sursă la destinatie se folosesc notatiile care urmează.

E_i ($\overline{E}_i$) – eveniment constând în operationalitatea (disfunctia) căii P_i.

R = Pr(existenta unei căi operationale) = $\Pr\left(\bigcup_{i=1}^{m} E_i\right)$.

Multimea de evenimente poate fi descompusă în evenimente mutual exclusive. După descompunere, expresia evenimentului "există o cale operațională" este

$$E_1 \cup (E_2 \cap \overline{E}_1) \cup (E_3 \cap \overline{E}_1 \cap \overline{E}_2) \cup \ldots \cup (E_m \cap \overline{E}_1 \cap \ldots \overline{E}_{m-1})$$

si

$$R = \Pr(E_1) + \Pr(E_2 \cap \overline{E}_1) + \Pr(E_3 \cap \overline{E}_1 \cap \overline{E}_2) + \ldots$$
$$\ldots + \Pr(E_m \cap \overline{E}_1 \cap \ldots \cap \overline{E}_{m-1})$$

Expresia din urmă poate fi rescrisă uzând de probabilităti conditionate

$$R = \Pr(E_1) + \Pr(E_2)\Pr(\overline{E}_1 | E_2) + \Pr(E_3)\Pr(\overline{E}_1 \cap \overline{E}_2 | E_3) + \ldots$$
$$\ldots + \Pr(E_m)\Pr(\overline{E}_1 \cap \ldots \cap \overline{E}_{m-1} | E_m)$$

Problema centrală este calcularea probabilităților conditionate de forma generală $\Pr(\overline{E}_1 \cap \ldots \cap \overline{E}_{i-1} | E_i)$.

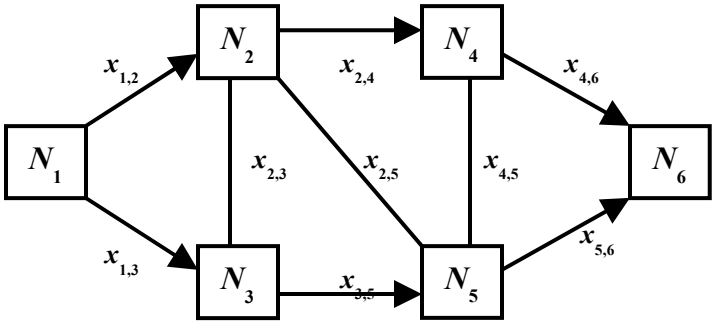
Pentru a identifica legăturile care trebuie să cadă pentru ca E_i să aibă loc dar nu $E_1, \ldots, E_{i-1}$, se folosesc asa-numitele multimi conditionate

$$S_{j/i} = P_j - P_i = \{x | x \in P_j \text{ si } x \notin P_i\}$$

Identificarea evenimentelor disjuncte în cazul general nu este totdeauna o operatie facilă.

Exemplu suplimentar pentru fiabilitatea terminală

O retea cu sase noduri care are 9 legături unidirectionale si 3 bidirectionale.



O listă cu toate căile de la N_1 la N_6:

$P_1 = \{x_{1,3}, x_{3,5}, x_{5,6}\}$
$P_2 = \{x_{1,2}, x_{2,5}, x_{5,6}\}$
$P_3 = \{x_{1,2}, x_{2,4}, x_{4,6}\}$
$P_4 = \{x_{1,3}, x_{3,5}, x_{4,5}, x_{4,6}\}$
$P_5 = \{x_{1,3}, x_{2,3}, x_{2,4}, x_{4,6}\}$
$P_6 = \{x_{1,3}, x_{2,3}, x_{2,5}, x_{5,6}\}$
$P_7 = \{x_{1,2}, x_{2,5}, x_{4,5}, x_{4,6}\}$
$P_8 = \{x_{1,2}, x_{2,3}, x_{3,5}, x_{5,6}\}$
$P_9 = \{x_{1,2}, x_{2,4}, x_{4,5}, x_{5,6}\}$
$P_{10} = \{x_{1,3}, x_{2,3}, x_{2,4}, x_{4,5}, x_{5,6}\}$
$P_{11} = \{x_{1,3}, x_{2,3}, x_{2,5}, x_{4,5}, x_{4,6}\}$
$P_{12} = \{x_{1,3}, x_{3,5}, x_{2,5}, x_{2,4}, x_{4,6}\}$
$P_{13} = \{x_{1,3}, x_{2,3}, x_{3,5}, x_{4,5}, x_{4,6}\}$

Căile, se observă, sunt ordonate de la cea mai scurtă la cea mai lungă.

Pentru a calcula alti termeni din sumă, trebuie avută în vedere intersectia mai multor multimi conditionate.

$P_1 = \{x_{1,3}, x_{3,5}, x_{5,6}\}$
$P_2 = \{x_{1,2}, x_{2,5}, x_{5,6}\}$
$P_3 = \{x_{1,2}, x_{2,4}, x_{4,6}\}$
$P_4 = \{x_{1,3}, x_{3,5}, x_{4,5}, x_{4,6}\}$

Pentru a calcula termenul al patrulea – expresia lui P_4 – multimile conditionate sunt:

$$S_{1/4} = \{x_{5,6}\};\ S_{2/4} = \{x_{1,2}, x_{2,5}, x_{5,6}\};\ S_{3/4} = \{x_{1,2}, x_{2,4}\}$$

$S_{1/4}$ este inclus în $S_{2/3}$; dacă $S_{1/4}$ este cu defecte, atunci si $S_{2/4}$ este cu defecte. $S_{2/4}$ poate fi ignorat în acest caz.
Al patrulea termen din ecuatia fiabilitătii este

$$p_{1,3}p_{3,5}p_{4,5}p_{4,6}(1-p_{5,6})(1-p_{1,2}p_{2,4})$$

Calculul termenului al treilea conduce la

$$S_{1/3} = \{x_{1,3}, x_{3,5}, x_{5,6}\}$$
$$S_{2/3} = \{x_{2,5}, x_{5,6}\}$$

Cele două multimi conditionate nu sunt disjuncte.
Evenimentul care constă în defectarea simultană a multimilor de arce $S_{1/3}$ si $S_{2/3}$ trebuie să fie împărtit în evenimente disjuncte:
(I) $x_{5,6}$ cu defecte
(II) $x_{5,6}$ este operational si atât $x_{1,3}$ cât si $x_{2,5}$ sunt defecte
(III) atât $x_{1,3}$ cât si $x_{5,6}$ sunt în functiune si atât $x_{2,5}$ cât si $x_{3,5}$ sunt defecte.
Pentru termenul al treilea rezultă expresia

$$p_{1,2}p_{2,4}p_{4,6}(q_{5,6} + p_{5,6}q_{1,3}q_{2,5} + p_{5,6}p_{1,3}q_{2,5}q_{3,5})$$

Termenii rămasi se calculează similar.
Fiabilitatea terminală este suma tuturor celor 13 termeni definiti mai devreme.

ANEXA 1

ELEMENTE DE TEORIA PROBABILITĂȚILOR SI DE STATISTICĂ MATEMATICĂ

Spatiul evenimentelor

Un experiment oarecare, provocat sau spontan poate avea rezultate diverse. Aceste rezultate sunt denumite *evenimente*. Astfel, rostogolirea unui zar pe o suprafată plană orizontală (exemplu aparent banal dar des utilizat de matematicieni) poate avea ca rezultat aparitia pe fata sa de deasupra a, să spunem, cinci puncte. S-a produs asadar evenimentul aparitiei pe fata de deasupra a cinci puncte. Tot asa, conform definitiei de mai sus, extragerea valetului de cupă dintr-un pachet de cărti de joc bine amestecat este un eveniment.

Dacă E este multimea tuturor evenimentelor posibile relativ la un experiment, această multime poate fi numită, cum adesea se întâmplă**,** *spatiul evenimentelor*. Evenimentele unui astfel de spatiu se pot găsi în anumite relatii unul cu altul si cu evenimentele acelui spatiu se pot face unele operatii.

O relatie importantă între evenimente este *implicatia*. Implicatia se notează $A \subset B$ si se citeste *evenimentul A implică evenimentul B*, ceea ce înseamnă că producerea evenimentului A conduce automat, implicit la producerea evenimentului B; implicatia mutuală, $A \subset B$ si $B \subset A$ este un mod de a exprima egalitatea sau echivalenta a două evenimente.

Operatiile cu evenimente sunt *unare*, cu un singur eveniment ca operand, sau *binare*, cu două evenimente ca operanzi.

Operatia de luare a *complementarului* sau, ceea ce este totuna, a *contrarului* unui eveniment este unară, operează cu un singur eveniment. *Reuniunea* si *intersectia* de evenimente sunt operatii binare, operează pe două evenimente.

Complementarul sau contrarul unui eveniment este acel eveniment care se produce atunci când nu se produce evenimentul al cărui contrar/complementar este. Într-un exemplu foarte simplu, aruncarea unei monede cu cădere pe o suprafată plană orizontală poate avea ca rezultat afisarea deasupra fie a unei fete, fie a celeilalte. Fiecare din cele două evenimente generate de acest experiment simplu este contrarul celuilalt. Dacă evenimentul asupra căruia se operează este A atunci evenimentul contrar lui se notează cu $\overline{A}$. De ce *contrar* si/sau *complementar* se va explica mai în detaliu după definirea celor două operatii binare anuntate.

Reuniunea a două evenimente se notează $A \cup B$ si este evenimentul care constă în producerea a cel putin unuia din cele două evenimente, adică sau a unui sau a celuilalt sau a ambelor evenimente.
Intersectia a două evenimente se notează $A \cap B$ si este evenimentul constând în producerea ambelor evenimente, adică atât a unui eveniment cât si a celuilalt.
Există două evenimente speciale care se includ în multimea *E*. Unul este evenimentul imposibil, notat cu $\emptyset$, evenimentul care nu se produce niciodată. Celălalt este evenimentul sigur, notat cu *E*, evenimentul care se produce de fiecare dată.
O relatie de forma $A \cap B = \emptyset$ exprimă incompatibilitatea mutuală a celor două evenimente *A* si *B*, în alte cuvinte producerea unuia exclude producerea celuilalt.
Acum se poate formula mai precis raportul între un eveniment si contrarul lui: $A \cap \overline{A} = \emptyset$, $A \cup \overline{A} = E$. Într-o lectură în cuvinte a acestor relatii, un eveniment este incompatibil cu contrarul său, producerea unui eveniment *sau* a contrarului său este sigură. Este acum momentul să se aducă precizarea că contrarul contrarului unui eveniment este exact acel eveniment. Simbolic, aceasta se scrie $\overline{\overline{A}} = A$.
Multimea *E* este partial ordonată, relatia de ordine este implicatia.
Multimea *E* împreună cu operatiile de luare a contrarului unui eveniment, de reuniune si de intersectie a evenimentelor se organizează ca o algebră booleană.
Între evenimentele dintr-o multime *E* se disting atomi (sau evenimente elementare) si evenimente compuse. De pildă, prin aruncarea zarului se pot produce între altele evenimentele A_2 si A_5 care constau în aparitia pe fata de deasupra a numărului de puncte trecut ca indice. Ambele sunt atomi sau evenimente elementare în sensul că nu sunt alte evenimente încă mai simple decât ele. Reuniunea $A_2 \cup A_5$ este însă un eveniment compus.
Fie acum Ω multimea tuturor evenimentelor elementare dintr-o multime finită *E* de evenimente. Evident $\Omega \neq \emptyset$. O submultime de părti ale lui Ω, $K \subset P(\Omega)$ se organizează ca un corp dacă

$$A \in K \Rightarrow \overline{A} \in K$$
$$A, B \in K \Rightarrow A \cup B \in K$$
$$A, B \in K \Rightarrow A \cap B \in K$$

În aceste coditii perechea (Ω, *K*) este un corp de evenimente si este un σ-corp sau corp borelian de evenimente dacă orice reuniune sau intersectie finită sau infinită de evenimente din *K* apartine multimii *K*.
Într-un spatiu *E* complet si atomic, orice eveniment $A \neq \emptyset$ se poate scrie ca o reuninune de elemente din Ω

$$A = \bigcup_{\omega_i \in \Omega} \omega_i$$

Se numeste *partitie* a unui eveniment $A \in K$ o multime de evenimente $A_i \in K$, ($i = 1, 2, \ldots, n$), care sunt mutual incompatibile, adică $A_i \cap A_j = \emptyset$ pentru orice pereche $A_i, A_j \in K$ cu $i \neq j$, astfel încât

$$\bigcup_{i=1}^{n} A_i = A$$

Dacă $A = \Omega$ atunci evenimentele $A_i \in K$, ($i = 1, 2, \ldots, n$) alcătuiesc un *sistem complet de evenimente* sau o *familie exhaustivă de evenimente*.

Probabilităti, probabilităti conditionate

Pe multimea evenimentelor din K se defineste o functie reală P, numită *probabilitate*, care are proprietătile:

1. $P(A) \geq 0, \forall A \in K$
2. $P(\Omega) = 1$
3. $P\left(\bigcup A_i\right) = \sum P(A_i), A_i \in K, A_i \cap A_j = \emptyset, i \neq j$

Dacă ultima proprietate are loc si pentru reuniuni numerabile atunci probabilitatea P se numeste complet aditivă (sau σ-aditivă) pe corpul (borelian) de evenimente (Ω, K).

Tripletul (Ω, K, P) se numeste câmp (borelian) de probabilitate. Dacă Ω este o multime finită atunci (Ω, K, P) este un câmp de probabilitate discret.

Din proprietătile de mai sus derivă alte câteva proprietăti importante ale probabilitătii P. Astfel

4. $P(\emptyset) = 0$
5. $P(A) = 1 - P(\overline{A})$
6. $P(A - B) = P(A) - P(A \cap B)$
7. $0 \leq P(A) \leq 1$
8. $P(A \Delta B) = P(A) + P(B) - 2P(A \cap B)$
9. $P(A \cup B) = P(A) + P(B) - P(A \cap B)$

unde $A - B = A \cap \overline{B}$ si $A \Delta B = (A - B) \cup (B - A)$ sunt diferenta, respectiv diferenta simetrică a două evenimente. O extindere a relatiei ultime la reuniunea a n evenimente este

$$P\left(\bigcup_{i=1}^{n} A_i\right) = \sum_{j=1}^{n} (-1)^{j+1} S_j \quad \text{cu} \quad S_j = \sum_{i_1, i_2, \ldots, i_j \leq n} P(A_{i_1} \cap \ldots \cap A_{i_j}) \quad j \leq n.$$

Dacă $F = \{A_i\}_{i \in I}$ este o familie numerabilă de evenimente două câte două mutual incompatibile, atunci $P\left(\bigcap_{i \in I} A_i\right) = 0$.

Dacă familia $F = \{A_i\}_{i \in I}$ este si exhaustivă, adică este un sistem complet de evenimente, atunci $P\left(\bigcup_{i \in I} A_i\right) = 1$.

Evenimentele se pot afla în relatie de conditionare reciprocă în sensul că un eveniment odată produs poate modifica probabilitatea de producere a altui eveniment. Probabilitatea devine astfel *conditionată*.

Relatia de bază pentru calculul probabilităților conditionate este

$$P_B(A) = P(A / B) = P(A \cap B) / P(B)$$

cu evenimentul B, cel care conditionează producerea evenimentului A, trecut ca indice sau pe pozitia a doua în notatia $P(A/B)$, deci după caracterul "/" (sau "|") în argumentul functiei probabilitate.

În general,

$$p(A / B) \neq P(A) \text{ si } P(B / A) \neq P(B)$$

ceea ce indică o dependentă, o conditionare reală între cele două evenimente. Dacă are loc egalitatea în ambele relatii atunci evenimentele nu se conditionează în nici un fel, sunt independente.

Dacă probabilitatea unei intersectii finite de evenimente este nenulă

$$P\left(\bigcap_{i=1}^{n} A_i\right) \neq 0$$

atunci probabilitatea respectivă se poate calcula cu formula

$$P\left(\bigcap_{i=1}^{n} A_i\right) = P\left(A_n / \bigcap_{i=1}^{n-1} A_i\right) \ldots P(A_2 / A_1) P(A_1)$$

care se demonstrează inductiv pornind de la relatia pentru două evenimente care se conditionează unul pe altul

$$P(A \cap B) = P(A / B) P(B) = P(B / A) P(A)$$

derivată simplu din formula probabilitătii conditionate.

Dacă $F = \{A_i\}_{i=\overline{1,n}}$ este o partitie a câmpului Ω atunci probabilitatea unui eveniment oarecare se poate calcula cu relatia

$$P(A) = \sum_{i=1}^{n} P(A_i) P(A / A_i)$$

cunoscută sub numele de *formula probabilitătii totale*.

Mai este de retinut *formula lui Bayes*

$$P(A_i / A) = \frac{P(A_i) P(A / A_i)}{\sum_{i=1}^{n} P(A_i) P(A / A_i)}$$

care în aceleasi conditii, $F = \{A_i\}_{i=\overline{1,n}}$ o partitie a câmpului Ω, permite calculul probabilitătii fiecărui eveniment al partitiei conditionat de evenimentul $A \in K$, altfel oarecare.

Exemplu. În cazul zarului corect enuntat mai devreme, multimea Ω este alcătuită din evenimentele A_1, A_2, A_3, A_4, A_5, A_6. Multimea de părti ale lui Ω care

se constituie în corp de evenimente este multimea tuturor reuniunilor de 2, 3, 4, 5 sau 6 evenimente la care se adaugă evenimentele atomice, elementare deja enumerate si evenimentul imposibil $\emptyset$.

Prin perceptie imediată se poate afirma că sansele de producere a celor sase evenimente sunt egale (sansa aceasta de producere a unui eveniment sau a altuia este măsurată de probabilitate). Se poate scrie, asadar

$$P(A_1) = P(A_2) = P(A_3) = P(A_4) = P(A_5) = P(A_6) = p$$

Evenimentul sigur Ω se poate scrie ca o reuniune

$$\Omega = A_1 \cup A_2 \cup A_3 \cup A_4 \cup A_5 \cup A_6$$

si deoarece evenimentele din reuniune sunt două câte două mutual incompatibile (nu pot apărea deasupra două fete diferite deodată), conform proprietătii 3 se poate scrie

$$1 = P(\Omega) = P(A_1) + P(A_2) + P(A_3) + P(A_4) + P(A_5) + P(A_6) = 6p$$

adică $p = 1/6$. Acum se pot calcula probabilităti diverse.

a) Probabilitatea aparitiei unui număr par de puncte este

$$P(A_2 \cup A_4 \cup A_6) = P(A_2) + P(A_4) + P(A_6) = 3(1/6) = 1/2$$

ca probabilitate a unei reuniuni de evenimente două câte două reciproc incompatibile.

b) Probabilitatea evenimentului A_4 conditionată de evenimentul reuniune de la punctul precedent, $A = A_2 \cup A_4 \cup A_6$

$$P(A_4/A) = \frac{P(A_4 \cap A)}{P(A)} = \frac{P[A_4 \cap (A_2 \cup A_4 \cup A_6)]}{P(A)} = \frac{P(A_4)}{P(A)} = \frac{1/6}{1/2} = \frac{1}{3}$$

etc.

De retinut din acest exemplu o modalitate de a evalua probabilităti prin raportarea numărului de situatii favorabile unui eveniment la numărul total de situatii. De pildă, evenimentul A_3 din cele de mai sus se produce în proportia 1 caz favorabil din 6 posibile, adică 1/6.

La loteria "6 din 49", se pot juca C_{49}^6 (combinări de 49 de numere luate câte 6) variante distincte. Sansa (probabilitatea) unei variante particulare de a fi câstigătoare a premiului cel mare este de $1/C_{49}^6$, o probabilitate foarte, foarte mică desigur.

Sansa de a câtiga la categoria a II-a este întrucâtva mai mare. Un bilet câtigător poate contine una din cele $C_6^5 = 6$ combinatii de 5 numere din cele iesite din urnă la extragerea duminicală, la care se adaugă unul din celelalte 43 de numere din afara extragerii. Numărul de situatii favorabile câstigării unui premiu la categoria a II-a este, evident, 6x43 = 258 si probabilitatea este de $258/C_{49}^6$, încă destul de mică dar mai mare decât cea de la categoria I.

La jocul de table (foarte cunoscut în toată lumea – backgammon), evolutia disputei dintre jucători este hotărîtă pas cu pas prin aruncarea a două zaruri. Într-o anumită fază a jocului, unul dintre jucători are nevoie ca zarurile să producă o sumă a punctelor egală cu 5. Care este probabilitatea acestui eveniment? Numărul total de rezultate este 36: fiecare din cele sase fete ale

unui zar poate apărea combinată cu oricare din cele sase fete ale celuilalt zar. Suma punctelor este 5 în următoarele 4 cazuri: (1, 4), (2, 3), (3, 2) si (4, 1). Prin raportarea numărului de cazuri favorabile (4) la numărul total de cazuri (36) se obtine răspunsul la întrebare: 4/36 = 1/9.

Variabile aleatoare

Formal, o *variabilă aleatoare* este o functie definită pe o multime atomică, cu valori reale, $X:\Omega \rightarrow R$, care are proprietatea

$$\{X < x\} \Rightarrow \{\omega \in \Omega / X(\omega) < x\} \in K, \forall x \in R$$

În termeni mai putini rigurosi din punct de vedere matematic, o variabilă aleatoare este o variabilă care ia valori la întâmplare dar în nici un caz haotic. Explicit sau tacit, în spatele manifestării varibilei aleatoare prin valori diverse se află un câmp de probabilitate (Ω, K, P) definit de multimea atomică Ω, de corpul de părti ale acesteia K si de probabilitatea P. Probabilitatea face ca unele valori pe care variabila aleatoare le poate lua să fie (eventual) mai probabile decât altele. Numărul de pucte afisate de un zar comun este o variabilă aleatoare. Cu fetele zarului, care pot fi de pildă colorate, nu neapărat "punctate", se pot asocia si alte numere printr-o functie X particulară. Functia X este o altă variabilă aleatoare definită pe câmpul (Ω, K, P).

O *variabilă aleatoare simplă* ia numai un număr finit de valori. De exemplu functia indicator a unui eveniment $A \in K$, care se poate produce sau nu

$$\chi_A = \begin{cases} 0 & \omega \notin A \\ 1 & \omega \in A \end{cases}$$

este o variabilă aleatoare simplă care ia numai două valori, 0 si 1. Variabilele aleatoare definite în relatie cu zarul sunt variabile aleatoare simple.

Dacă X este o variabilă aleatoare definită pe câmpul (Ω, K, P) atunci pentru oricare două valori $x_1, x_2 \in R, x_1 \leq x_2$ toate intervalele finite sau infinite delimitate de cele două valori corespund unor evenimente din K si, prin generalizare, pentru orice multime I reuniune de intervale din R, se poate calcula $P_X(I) = P[X(\omega) \in I] = P[X^{-1}(I)]$.

$P_X(I)$ reprezintă *distributia probabilistică* a variabilei aleatoare X. Se poate vorbi de P_X ca de o probabilitate definită pe câmpul (R, K_X) în care $K_X = \{I \subset R / X^{-1}(I) \in K\}$ este o multime de intervale ale multimii numerelor reale R, intervale care sunt imagini prin functia X ale unor evenimente din K.

Dacă variabila aleatoare X ia valori într-o multime cel mult numerabilă

$$\{x_i / x_i \in R, i \in I, I \subset N^+\}$$

atunci ea se numeste discretă si

$$\sum_{i \in I} P_X(x_i) = 1$$

$$P_X(J) = \sum_{x_i \in J} P_X(x_i), \forall J \in K_X$$

Dacă X poate lua toate valorile unui interval $I \in K_X$ atunci probabilitatea asociată intervalului este

$$P_X(I) = \int_I f_X(x)dx$$

si este o functie absolut continuă. Functia $f_X(x)$ care apare în formulă se numeste *densitatea de probabilitate* sau *densitatea de repartitie* a variabilei aleatoare X, este nenegativă pentru orice x si are proprietatea

$$\int_{-\infty}^{\infty} f_X(x)dx = 1$$

Functia

$$F_X(x) = P[X(\omega) < x] = P_X[(-\infty, x)] = \int_{-\infty}^{x} f_X(x)dx$$

se numeste *functie de repartitie* a variabilei aleatoare X.
Functia de repartitie este nedescrescătoare pe întreaga axă reală

$$a < b \Rightarrow F_X(a) \le F_X(b) \quad \forall a,b \in R$$

si este continuă la stânga în fiecare punct

$$\lim_{x \to a, x < a} F_X(x) = F_X(a) \; \forall a \in R$$

Valorea minimă si valoarea maximă ale unei functii de repartitie sunt date de limitele

$$\lim_{x \to -\infty} F_X(x) = 0, \quad \lim_{x \to \infty} F_X(x) = 1$$

Eventualele discontinuităti sunt de speta primă si sunt cel mult numerabile. Reciproc, orice functie cu proprietătile de mai sus poate fi pusă în corespondentă cu un câmp de probabilitate.
Pentru o variabilă aleatoare discretă

$$F_X(x) = \sum_{x_i < x} P_X(x_i)$$

iar pentru una continuă, în afară de relatia scrisă deja mai sus

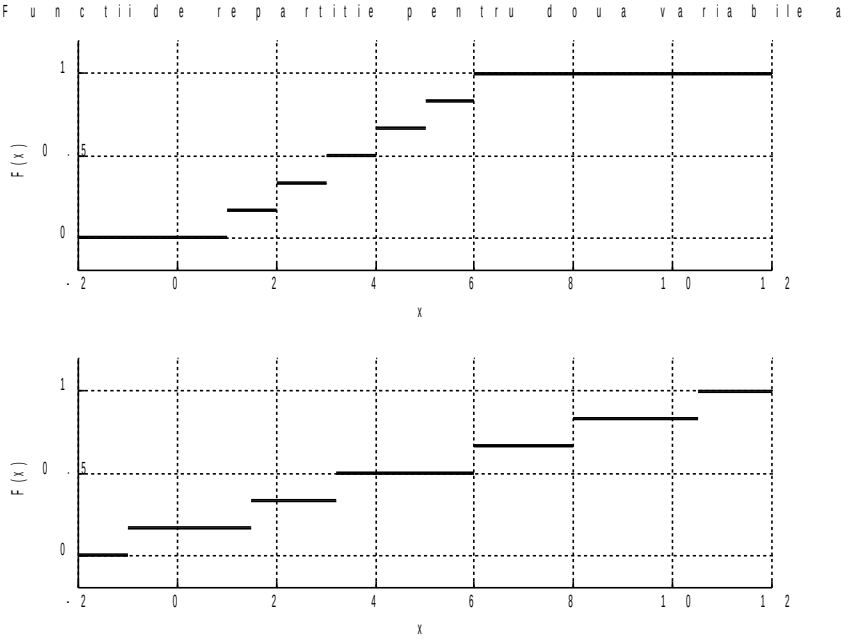
$$F_X(x) = \int_{-\infty}^{x} f_X(x)dx$$

are loc si relatia

$$f_X(x) = \frac{d}{dx} F_X(x)$$

Pentru orice interval $[a,b) \subset R \quad P_X\{[a,b)\} = F_X(b) - F_X(a)$ si pentru orice valoare a, $P_X(a) = 0$.
În referirea făcută putin mai devreme la cazul zarului, s-a semnalat posibilitatea ca pe acelasi câmp de probabilitate să se definească nu una ci mai multe variabile aleatoare. Se notează uzual cu $V(\Omega, K, P)$ multimea tuturor variabilelor aleatoare definite pe câmpul de probabilitate trecut între paranteze.

Dacă $X, Y \in V(\Omega, K, P)$ atunci suma, diferenta, produsul celor două variabile aleatoare, modulul, puterea, în general o functie măsurabilă Borel de oricare dintre ele sunt toate variabile aleatoare din multimea $V(\Omega, K, P)$.
Ori de câte ori nu este pericol de confuzie, variabila aleatoare trecută până acum ca indice al functiei de repartitie sau al functiei densitate de probabilitate/de repartitie poate lipsi din acea pozitie.
Dacă se reia exemplul zarului, care la fiecare experientă este făcut să se rostogolească pe o suprafată plană, orizontală, atunci multimea evenimentelor elementare (atomi) Ω este alcătuită din aparitiile deasupra a celor sase fete, marcate uzual cu unu până la sase puncte. Multimea de părti ale lui Ω este alcătuită din evenimentele elementare si din toate reuniunile posibile de evenimente elementare la care se adaugă evenimentul imposibil. Multimea K organizată ca un corp de evenimente coincide chiar cu multimea de părti $P(\Omega)$, iar functia numită probabilitate ia valoarea 1/6 pentru fiecare din evenimentele elementare deoarece fețele zarului au sanse egale de a apărea deasupra.



Funcții de repartiție pentru cazul zarului corect

Cum s-a mai spus, numărul de puncte observat pe fața de deasupra a zarului poate fi considerat o variabilă aleatoare. În acest caz functia de repartitie se prezintă ca în graficul superior din desenul de mai sus si este, ca pentru orice variabilă aleatoare discretă, o functie în scară.
Dar pe acelasi câmp de probabilitate se pot defini si alte variabile aleatoare. Pe câmpul asociat zarului perfect se poate imagina, de pildă, functia $X:\Omega \to R$ definită astfel

ω_1	ω_2	ω_3	ω_4	ω_5	ω_6
- 1	1,5	3,2	6	8	10,5

($\omega_i \equiv A_i$, $i = 1, 2, 3, 4, 5, 6$) si atunci functia de repartitie se prezintă diferit, ca în graficul inferior din aceeasi figură. Asadar, multimea $V(\Omega, K, P)$ este extrem de bogată.

De variabilele aleatoare sunt legate câteva valori remarcabile. Una foarte importantă este *media* definită ca

$$M(x) = \int_{-\infty}^{\infty} xf(x)dx$$

care face parte din lista nesfârsită a momentelor de diferite ordine ale variabilei, acesta fiind momentul de ordinul 1.

Cu o relatie similară se poate calcula media unei functii $g(x)$ de variabila aleatoare x având în vedere caracterul aleator al valorilor functiei

$$M[g(x)] = \int_{-\infty}^{+\infty} g(x)f(x)dx$$

si dacă în particular $g(x) = x^r$, $r \in N$ avem tocmai *momentul de ordinul r* despre care s-a amintit.

În cazul particular $g(x) = [x - M(x)]^2$ se obtine o altă valoare importantă, caracteristică variabilei aleatoare descrise de functia de repartitie $F(x)$ sau de densitatea de repartitie $f(x)$, si anume *dispersia*. Rădăcina pătrată pozitivă a dispersiei se numeste *abaterea medie pătratică* a variabilei aleatoare respective. Dispersia este momentul centrat de ordinul doi al variabilei aleatoare, unul din multiplele momente ale variabilei, centrate pe medie, de ordine diferite.

Nu numai variabilele aleatoare continue au momente, medii, dispersii, ci si cele discrete. În cazul discret, formulele de calcul contin sume în locul integralelor si valorile variabilei parcurg întreaga listă de valori posibile, iar densitatea de repartitie este înlocuită de probabilitătile asociate valorilor pe care variabila le poate lua efectiv.

Câteva legi de repartitie teoretice foarte utilizate sunt prezentate pe scurt în continuare.

Legea binomială (Bernoulli) este exprimată de relatia

$$P(m) = C_n^m p^m (1 - p)^{n-m}$$

cu $0 \le m \le n$ si p un număr în intervalul [0, 1]. Legea binomială este de tip discret. Variabila aleatoare este m. Are media np si dispersia $np(1 - p)$. Există un model fizic legat de această lege de repartitie. Modelul îl constituie o urnă cu bile de două culori, iar evenimentele constau în rezultatele extragerii repetate a câte unei bile după care bila extrasă este reintrodusă în urnă. Variabila m reprezintă numărul bilelor de o anumită culoare din cele două, în n extrageri succesive, conform schemei cu bila returnată. Numărul p reprezintă proportia de bile de acea culoare în urnă, cu alte cuvinte probabilitatea de extragere a unei bile de culoarea respectivă.

Legea Poisson exprimată sub forma

$$P(m) = \frac{\mu^m}{m!}\exp(-\mu)$$

cu $\mu > 0$ si m natural ca variabilă aleatoare. Media variabilei m este μ, dispersia ei este de asemenea μ. Un modelul fizic îl reprezintă numărul dezintegrărilor radioactive, numărul de apeluri telefonice solicitate într-o centrală etc. într-un interval de timp precizat, (relativ) scurt.

Legea normală (gaussiană) care este dată de densitatea de probabilitate

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-m)^2}{2\sigma^2}}$$

în care m este media variabilei x si σ^2 este dispersia ei. Legea normală este considerată o lege limită pentru sumele de variabile aleatoare. Un fenomen afectat de foarte multi factori aleatori care actionează prin însumare (aditiv) se prezintă de cele mai multe ori ca un fenomen aleator descris de o lege normală.

Variabilele aleatoare din expunerea teoretică sau din exemplele prezentate mai sus au fost până acum simple, adică a fost vorba în toate cazurile de o singură aplicatie $X:\Omega \to R$ legată de un unic câmp de probabilitate (Ω, K, P). Se pot imagina variabile aleatoare cu mai multe componente, variabile aleatoare sub forma unor vectori cu componente aleatoare definite relativ la un acelasi câmp de probabilitate sau la câmpuri de probabilitate diferite. Astfel, legea următoare se referă la o variabilă aleatoare vectorială.

Legea normală multidimensională dată de densitatea de repartitie

$$f(x) = \frac{1}{(2\pi)^{\frac{n}{2}}\sqrt{\det W}} e^{-\frac{1}{2}(x-m)^T W^{-1}(x-m)}$$

cu media m un vector cu n componente si cu matricea de covariatie W o matrice $n \times n$ pozitiv definită. Pentru ca expresia dată să aibă consistența necesară trebuie definită mai exact matricea W.

Este de comentat mai întâi problema corelatiei a două variabile aleatoare. Două variabile aleatoare pot fi necorelate, caz în care valorile uneia nu influentează în nici un fel valorile pe care le poate lua cealaltă, dar, alternativ, pot fi mai mult sau mai putin dependente, ceea ce înseamnă că dacă una din variabile a luat o valoare atunci legea de repartitie a celeilalte se modifică în functie de acea valoare a primei variabile.

Fiind date două variabile aleatoare x si y de medii nule, media produsului lor $M(xy)$ se numeste covariatie. Dacă covariatia este nulă se poate spune în general că cele două variabile nu sunt corelate. Dimpotrivă, dacă $M(xy) \neq 0$ variabilele sunt corelate, există o corelatie între ele, există o dependență între valorile pe care ele le iau în sensul arătat putin mai devreme. Dacă mediile sunt diferite de zero, afirmatia si definitia se mentin pentru abaterile de la medie. Întrucât covariatia $M(xy)$ poate lua valori foarte diferite, pentru o apreciere cantitativă mai riguroasă a tăriei corelatiei se utilizează coeficientul de corelatie

$$\rho = \frac{M(xy)}{\sqrt{M(x^2)M(y^2)}}$$

care ia valori în intervalul [–1, 1] si în expresia căruia se disting dispersiile celor două variabile, $M(x^2)$ si $M(y^2)$. O valoare apropiată de extremele intervalului indică o corelatie strânsă, o valoare apropiată de zero exprimă o corelatie slabă.

Componentele unui vector aleator, privite ca variabile aleatoare simple sunt mutual mai mult sau mai putin corelate. Se defineste ca matrice a covariatiilor unui vector aleator x media produsului xx^T, adică media produsului acelui vector cu transpusul său. Se obtine o matrice pătrată simetrică care are pe diagonală dispersiile componentelor pure. Aceasta este matricea W utilizată în expresia densitătii de repartitie a variabilei aleatoare normale multidimensionale din discutia de mai sus. Dacă matricea covariatiilor este diagonală (are toate elementele nule cu exceptia celor de pe diagonala principală) atunci componentele vectorului aleator sunt mutual independente. Împărtirea fiecărui element al matricei covariatiilor cu abaterile medii pătratice ale componentelor corespunzătoare ale vectorului x produce o matrice a coeficientilor de corelatie, cu 1 pe diagonală, cu valori in intervalul [–1, 1] în rest.

B I B L I O G R A F I E

1. T.I.Băjenescu *Fiabilitatea, disponibilitatea si mentenabilitatea sistemelor electronice complexe*, Editura de Vest 1997

2. R.K.Iyer, Cursul ECE 542, Department of Electrical and Computer Engineering, University of Illinois at Urbana-Champaign, Ill., 2004

3. I.Karen, Cursul ECE 655, Department of Electrical and Computer Engineering, University of Massachusetts, Amherst Mass., 2006

4. A.Krings, Cursurile CS 449/549, Department of Computer Science, Unversity of Idaho, Moscow, Idaho, 2005

5. Gh.M.Panaitescu *Sisteme tolerante la defecte*, Note de curs (pe suport electronic), Universitatea "Petrol-Gaze" Ploiesti, 2006

6. K.K.Saluja, Cursul ECE 753, Department of Electrical and Computer Engineering, University of Wisconsin, Madison Wisc., 2005

7. C.Stefănescu *Sisteme tolerante la defecte*, Matrix Rom, Bucuresti 1999