Poissulkemisongelman ratkaisun kehitys

Esitelmä tietojenkäsittelytieteen historia -seminaarissa

Marko Ullgrén

Tietojenkäsittelytieteen laitos

Helsingin yliopisto

16.3.2000

1 Johdanto

Tietojenkäsittelytieteen historiaa voidaan tarkastella myös yhden ongelman ja sen ratkaisuun kehitettyjen algoritmien näkökulmasta. Päähuomiota ei tällöin tule kiinnittää algoritmien teknisiin yksityiskohtiin, vaan siihen, miten ratkaisut ovat kehittyneet ja miten tietojenkäsittelytieteen ja tietotekniikan kehityksen suuret linjat näkyvät tämän yksittäisen ongelman ratkaisun kehityksessä. Tärkeää on huomata, että kerran ratkaistu ongelma voidaan joutua ratkaisemaan monia kertoja uudestaan ympäröivän maailman ja vaatimusten muuttuessa.

Poissulkemisongelma (mutual exclusion problem) on poikkeuksellisen pitkäikäinen tietojenkäsittelytieteen ongelma. Se ratkaistiin ensimmäisen kerran ohjelmallisesti vuonna 1965, ja uusia ratkaisuja on esitetään edelleen jatkuvasti. Tämä esitys rajoittuu tutkimaan nimenomaan ongelman ohjelmallisia ratkaisuja. Ohjelmallisten ratkaisujen kehityksessä on tapahtunut paljon mielenkiintoista viime vuosikymmenten aikana, kun taas laitetason ratkaisut ovat pysyneet miltei samoina lähes 40 vuotta. Rajaus ohjelmallisiin ratkaisuihin tehdään huolimatta siitä, että käytännön järjestelmissä laitetason ratkaisut ovat vielä laajassa käytössä.

Poissulkemisongelman ongelmanasettelu on yksinkertainen: järjestelmässä on kriittinen alue (critical section), jota saa käyttää vain yksi järjestelmän *n*:stä prosessista kerrallaan. Ongelmana on löytää prosesseille yhteinen käytäntö, jota noudattamalla voidaan taata, että

- korkeintaan yksi prosessi kerrallaan on suorittamassa kriittistä aluetta,
- mikä tahansa prosesseista voi olla suorittamassa kriittistä aluetta millä ajanhetkellä hyvänsä,

- prosessit ovat ratkaisussa tasa-arvoisia, ts. prosesseille ei ole määritelty minkäänlaisia pysyviä prioriteetteja,
- prosessien suoritusnopeuden vaihtelu ei saa vaikuttaa ratkaisun toimintaan, ja
- kriittiselle alueelle haluavan prosessin on myös lopulta päästävä sinne (freedom from starvation) [Dij65].

Käytännössä kriittinen alue voi olla esimerkiksi I/O-laitetta ohjaavan ajurin ohjelmakoodi.

2 Yhteistä muistia käyttävät ratkaisut

Poissulkemisongelman ratkaisu tuli ajankohtaiseksi 60-luvulla, kun ensimmäinen moniajokäyttöjärjestelmä, IBM System/360, tuli käyttöön [Tan92]. Moniajokäyttöjärjestelmässä prosessit eivät kuitenkaan voineet toimia täysin itsenäisesti, sillä osa laitteiston resursseista oli sellaisia, että ne kannatti esimerkiksi kustannussyistä toteuttaa yhteisinä kaikille prosesseille. Jotta prosessit voisivat käyttää näitä resursseja, täytyi niiden pystyä varamaan resurssi yksinoikeudella käyttöönsä. Näin päädyttiin poissulkemisongelmaan. Yksiprosessorikoneissa ongelma voitiin ratkaista laitetasolla esimerkiksi kieltämällä keskeytykset kriittisen alueen suorituksen ajaksi tai ottamalla käyttöön erityinen *test and set* -komento, joka tarkistaa ja muuttaa muistipaikan arvon yhden kellojakson aikana. Moniprosessorikoneissa nämä ratkaisut eivät kuitenkaan toimi, koska eri prosessien suoritus on aidosti samanaikaista [WWW1].

Ensimmäiset monen prosessin järjestelmät toteutettiin yhteistä muistia käyttävinä. Yhteisen muistin järjestelmissä prosesseilla on niiden oman muistin lisäksi yhteistä muistia, johon kirjoittaminen ja josta lukeminen ovat atomaarisia (jakamattomia) operaatioita. Jos prosessit yrittävät käyttää muistia samaan aikaan, voivat operaatiot tapahtua missä järjestyksessä tahansa [Dij65], mutta eivät päällekkäin.

T. Dekker ratkaisi kahden prosessin poissulkemisongelman ohjelmallisesti vuonna 1965 [Ray86]. Edsger Wybe Dijkstra yleisti ratkaisun mielivaltaisen monelle prosessille myöhemmin samana vuonna julkaisemassaan sittemmin kuuluisaksi tulleessa yhden sivun mittaisessa artikkelissaan [Dij65]. Dijkstra tunnetaan omaperäisenä tietojenkäsittelytieteen tutkijana, hän kuuleman mukaan välttää tietokoneen käyttöä ja ratkaisee ongelmat mieluiten kynällä ja paperilla. Hänen luentonsa ovat hyvin pidettyjä, käsin kirjoitettujen kalvojen pohjalta on jopa tehty fontti. Dijkstran melko monimutkaisessa ratkaisussa prosessien käytössä on kaksi totuusarvoista taulukkoa, joiden molempien koko on prosessien lukumäärä *n*. Kullakin prosessilla on oma indeksinumeronsa väliltä *1...n*. Prosessi kirjoittaa totuusarvoisiin taulukkoihin ainoastaan oman indeksinsä kohdalle, muita taulukon paikkoja se ainoastaan lukee. Lisäksi käytössä on yksi yhteinen numeerinen muuttuja, jonka arvoa mikä tahansa prosesseista voi muuttaa. Ratkaisun algoritmi *i.* prosessille on esitetty kuvassa 1.

```
integer j;
Li0:  b[i] = false;
Li1:  if k ≠ i then
Li2:      begin
            c[i] := true;
Li3:        if b[k] then k := i;
            goto Li1;
          end
      else
Li4:  begin
          c[i] := false;
          for j := 1 step 1 until N do
            if j ≠ i and not c[j] then goto Li1
      end;
      kriittinen alue;
      c[i] := true;
      b[i] := true;
      goto Li0;
```

Kuva 1: Dijksran ohjelmallinen ratkaisu [Dij65] poissulkemisongelmaan hänen oman käsialansa pohjalta tehdyllä fontilla kirjoitettuna.

Prosessi voi päästä kriittiselle alueelle ainoastaan tilanteessa, jossa se on suoriutunut kohdasta *Lil4* alkavasta kootusta lauseesta ilman että se on hypännyt *goto*-käskyllä kohtaan *Lil1*. Tämä on mahdollista vain silloin, kun prosessi on asettanut oman paikkansa taulukosta *c* arvoon *epätosi*, ja havainnut kaikkien muiden taulukon paikkojen olevan asetettuna arvoon *tosi*. Näin ollen ainoastaan yksi prosessi kerrallaan voi olla kriittisellä alueella [Dij65].

Dijkstran ratkaisu ei takaa, että kriittiselle alueelle haluava prosessi lopulta pääsee sinne, ts. yksi prosessi voi joutua odottamaan ikuisesti samalla, kun toiset prosessit pääsevät kriittiselle alueelle. Dijkstran algoritmista esitettiinkin pian paranneltuja versioita, jotka takaavat, että jokainen pääsyä odottava prosessi pääsee kriittiselle alueelle polynomiaalisessa [Knu66] tai lineaarisessa [EiM72] odotusajassa.

2 Tilamuuttujiin perustuvat ratkaisut

Yhteisen muistin käyttö osoittautui pian kohtuuttomaksi vaatimukseksi. Prosessien yhteinen muisti oli hankala toteuttaa, varsinkin muistiin kirjoittamisen atomaarisuus tuotti ongelmia: tilanne, jossa kaksi prosessia yritti samaan aikaan kirjoittaa samaa muuttujaa vaati itse asiassa laitetason poissulkemista. Avuksi otettiin tilamuuttujat. Tilamuuttujalla tarkoitetaan muuttujaa, jonka arvon kaikki prosessit voivat lukea, mutta vain yksi prosessi muuttaa. Lamportin vuonna 1974 esittämä ratkaisu [Lam74] perustuu tilamuuttujien käyttöön, eikä se tarvitse yhtään jaettua muuttujaa Dijkstran ratkaisun tapaan.

Ratkaisu tunnetaan nimellä Leipurin algoritmi (Bakery Algorithm), oikeampi ja kuvaavampi suomennos olisi ollut kääntää nimi suoraan Leipomoalgoritmiksi. Se perustuu kaupoista - tai leipomosta - tuttuun jonotusnumerokäytäntöön, jossa asiakkaita palvellaan heidän saamiensa jonotusnumeroiden mukaan. Käytössä on kaksi taulukkoa, joista toinen pitää sisällään numeerisia arvoja ja toinen totuusarvoja. Molemmat ovat tilamuuttujia: kukin prosessi voi kirjoittaa ainoastaan yhtä taulukon paikkaa. Taulukot on alustettu arvoihin *0* ja *epätosi*. Lisäksi kullekin prosessille on määritelty oma tunnistenumero eli indeksi. Algoritmi *i*. prosessille on kuvassa 2.

```
choice[i] := true;
number[i] := max(number[0]+...+ number[n]);
choice[i] := false;
for j = 0 to n-1,  i  ≠ j
begin
      wait ¬choice[j];
      wait (number[j] ≠ 0) and ( (number[i] < (number[j]) or
             ( number[i] = number[j] and i < j ) )
end;
kriittinen alue;
number[i] := 0;
```

Kuva 2: Leipurin algoritmi [Lam74].

Algoritmi toimii seuraavasti: prosessi asettaa totuusarvoiseen taulukkoon oman indeksinsä osoittamaan paikkaan arvon *tosi*. Tämän jälkeen se lukee kaikki numeerisen taulukon arvot ja asettaa oman indeksinsä kohdalle taulukkoon arvon, joka on yhtä suurempi kuin suurin taulukossa ollut arvo. Lopuksi totuusarvoisen taulukon muutettu arvo asetetaan takaisin arvoon *epätosi*. Tämä vaihe vastaa numeron ottamista jonotusnumerolaitteesta. Totuusarvoisen taulukon tarkoitus on ilmaista muille prosesseille, että prosessi on parhaillaan valitsemassa itselleen numeroa.

Seuraavaksi prosessi siirtyy odottamaan vuoroaan. Totuusarvoista taulukkoa tutkimalla varmistutaan, ettei minkään prosessin numeronvalinta ole kesken. Prosessi odottaa, kunnes kaikki numeerisen taulukon nollasta poikkeavat jonotusnumerot ovat suurempia kuin sen oma numero. Kun prosessin oma numero on taulukon pienin, saa prosessi siirtyä kriittiselle alueelle. Kriittisen alueen suorituksen jälkeen prosessi asettaa taulukossa olevan jonotusnumeronsa nollaksi.

Leipurin algoritmi on selvästi vikasietoisempi kuin aikaisemmat ratkaisut, joissa yhteiskäyttöistä muuttujaa käyttävän prosessin vikaantuminen saattoi pahimmassa tapauksessa lamauttaa koko järjestelmän. Yhteisen muuttujan kirjoittamisesta aiheutuvaa ongelmaa ei myöskään tässä ratkaisussa ole.

3 Viestinvälitykseen perustuvat ratkaisut

Tilamuuttujan käytöstä tuli pian rasite prosessien itsenäisyydelle. Tilamuuttuja ei ollut ainoastaan vaikeasti toteutettavissa, siihen perustuvat ratkaisut olivat myöskin väistämättä alttiita vioille. Lähiverkon (local area network, LAN) tultua käyttöön 70-luvulla alettiin myös tutkia hajautettuja järjestelmiä, joissa tilamuuttujan toteuttaminen hajautetusti on hankalaa. Monella taholla ryhdyttiinkin tutkimaan mahdollisuutta toteuttaa keskinäinen poissulkeminen ilman tilamuuttujien käyttöä.

Prosessien välinen viestinvälitys tarjosi uuden välineen ongelman ratkaisuun. Lamportin esittämästä leipurin algoritmista voidaankin helposti muokata viestinvälitykseen perustuva versio, jossa tilamuuttujan lukeminen korvataan muuttujan arvoa kysyvällä viestillä ja tämän vastauksella.

Miltei kaikki tämän jälkeen esitetyt viestinvälitykseen perustuvat ratkaisut voidaan luokitella toimintaperiaatteensa perusteella kahteen ryhmään: vuoromerkkiä käyttäviin ja luvan kysymiseen perustuviin algoritmeihin [Ray91].

3.1 Vuoromerkkiin perustuvat ratkaisut

Vuoromerkkiin (token) perustuva keskinäisen poissulkemisen toteutus on perusperiaatteeltaan hyvin yksinkertainen: järjestelmässä on yksi vuoromerkki, joka kiertää prosessilta toiselle. Prosessit ajatellaan järjestetyksi renkaan muotoon. Vuoromerkki kulkee renkaassa aina samaan suuntaan prosessilta toiselle. Vuoromerkkiä voi kerrallaan pitää hallussaan vain yksi järjestelmän prosesseista ja ainoastaan vuoromerkkiä hallussaan pitävä prosessi voi käyttää kriittistä aluetta. Keskinäisen poissulkemisen ehto on siis triviaalisti voimassa. Jotta prosessin äärellinen odotusaika ennen kriittiselle alueelle pääsemistä voidaan taata, on pidettävä huoli vuoromerkin jatkuvasta siirtymisestä prosessilta toiselle [Ray91]. Samanlainen malli on käytössä IBM:n kehittämässä *Token Ring* - lähiverkossa, jossa vuoromerkkiä hallussaan pitävä tietokone saa lähettää verkkoon.

Vian mahdollisuus on vuoromerkkiin perustuvassa järjestelmässä ilmeinen: vuoromerkkiä hallussa pitävän prosessi voi vikaantua ja vuoromerkki kadota. Tällöin on generoitava uusi vuoromerkki. Vuoromerkin katoamisen havaitseminen edellyttää jonkinlaista aikaviiveluopumiskäytäntöä (timeout). Vuoromerkin uudelleen generoiminen on selvästikin järjestelmää huomattavasti hidastava toimenpide aikaviipeineen ja tilamuuttujien arvojen tutkiminiseen. Vuoromerkkiin perustuva ratkaisu on siis täysin hajautettu, mutta vikasietoisuuden kannalta se ei ole kovin houkutteleva.

3.2 Luvan kysymiseen perustuvat ratkaisut

Luvan kysymiseen perustuvien algoritmien perusperiaate on yksinkertainen: kriittiselle alueelle haluava prosessi kysyy muilta prosesseilta lupaa mennä kriittiselle alueelle. Poissulkemista haluava prosessi lähettää pyynnön (request) muille prosesseille. Pyynnön vastaanottava prosessi lähettää vastauksen (reply) välittömästi, jos se ei itse ole menossa kriittiselle alueelle sillä hetkellä. Jos pyynnön vastaanottaja on sekin kiinnostunut kriittisestä alueesta, syntyy usean samanaikaisen pyynnön välinen kilpailutilanne. Tällöin verrataan saapuneen pyynnön ja prosessin itsensä lähettämän pyynnön aikaleimoja. Jos saapunut pyyntö on lähetetty aikaisemmin, lähetetään sille vastaus. Lopuksi, kun poissulkemista haluava prosessi on saanut vastauksen kaikilta prosesseilta, voi se siirtyä suorittamaan kriittistä aluetta [Ray91].

Ricartin ja Agrawalan vuonna 1981 esittämässä ratkaisussa [RiA81] poissulkemista haluava prosessi toimii edellä kuvatulla tavalla lähettäen pyynnön kaikille järjestelmän prosesseille. Näin ollen keskinäisen poissulkemisen toteuttaminen n-prosessisessa järjestelmässä vaatii $2*(n-1)$ viestiä: $n-1$ pyyntöä ja $n-1$ vastausta. Jokaiseen pyyntöön liitetään Lamportin säännön mukainen aikaleima, jonka perusteella voidaan ratkaista tilanne, jossa samaan aikaan saapuu monta pyyntöä.

Ratkaisu olettaa, että jokaisella prosessisolmulla on tieto kaikista muista solmuista. Uuden solmun liittyessä tai jonkin solmun poistuessa järjestelmästä on sen tiedotettava tapahtumasta kaikille järjestelmän muille solmuille. Prosessisolmun vikaantumista ei algoritmissa itsessään ole huomioitu. Vian varalle voidaan määritellä aikaviiveluopumiskäytäntö (timeout), jossa prosessi, joka ei vastaa pyyntöihin määritellyn ajan sisällä, poistetaan järjestelmästä. Tämän ajan täytyy olla pidempi kuin pisin mahdollinen kriittisen alueen suoritusaika, jotta kriittisellä alueella olevia prosesseja ei luultaisi vikaantuneiksi. Ratkaisu on siis suhteellisen tehokas täydellisesti toimivassa järjestelmässä, mutta viat hidastavat järjestelmän toimintaa dramaattisesti. Ongelmallinen on myös tilanne, jossa vikaantunut prosessisolmu alkaa taas toimia. Tämäkin ratkaisu on siis hajautettu, mutta se kärsii vikatilanteessa vastaavanlaisesta aikaviiveluopumiskäytännön tuomasta viiveestä kuin vuoromerkkiin perustuva ratkaisu. Aidosti vikasietoinen ratkaisu antoi vielä odottaa itseään.

Agrawalin ja el Abbadin vuonna 1991 esittämä ratkaisu [AgA91] on huomattavasti vikasietoisempi kuin edellä esitetty ratkaisu. Ratkaisu perustuu havaintoon, että lupaa ei tarvitse kysyä kaikilta muilta prosesseilta. Suppeampikin joukko riittää. Tämän vaikutus vikasietoisuuteen on ilmeinen: vikaantunut prosessi voidaan useimmiten jättää pois niiden prosessien joukosta, joille lupapyyntö lähetetään.

Määritellään käsite edustajisto (quorum) joukkona, jolle pätevät seuraavat ehdot:

- kahden edustajiston leikkaus on epätyhjä (1)
- edustajisto ei voi olla toisen edustajiston osajoukko, ts. edustajistolla on oltava vähintään yksi alkio, joka ei kuulu toiseen edustajistoon (2).

Hajautetun järjestelmän prosessien joukosta voidaan aina valita edustajisto. Ehdot (1) ja (2) takaavat keskinäisen poissulkemisen, jos resurssia tarvitseva prosessi kysyy lupaa kaikilta johonkin edustajistoon kuuluvilta prosesseilta. Triviaali tapa valita edustajisto on valita enemmistö kaikista prosesseista. Artikkelissa esitetyllä tavalla voidaan muodostaa edustajisto, vaikka $log\ n$ kappaletta prosesseista olisi toimimattomia. Joissain tapauksissa edustajisto pystytään muodostamaan, vaikka $n - log\ n$ solmua olisi pois toiminnasta. Edustajiston muodostaminen on siis parhaassa tapauksessa - kaikki prosessit toiminnassa - nopeaa ($log\ n$ viestiä). Myös vikaantuneessa tilassa, jossa suuri osa prosesseista on poissa toiminnasta, saadaan edustajisto muodostettua lähettämällä kohtuulliset $(n+1) / 2$ viestiä.

Poissulkemista haluava prosessi lähettää pyynnön (request) muodostamalleen edustajistolle. Jokainen pyynnön vastaanottaja tallettaa pyynnön omaan sisäiseen pyyntöjonoonsa, joka on järjestyksessä pyynnön saapuessa kiinnitetyn aikaleiman mukaan. Jonon kärjessä olevan pyynnön lähettäjälle lähetetään vastaus (reply). Kun poissulkemista haluava prosessi on saanut vastauksen kaikilta edustajistoon kuuluvilta prosesseilta, voi se siirtyä suorittamaan kriittistä osiota. Prosessin poistuessa kriittisestä osiosta se lähettää poistumisviestin (relinquish) edustajistoon kuuluville prosesseille, jolloin nämä voivat poistaa pyynnön jonoistaan.

Jos prosessi saa pyynnön, jonka aikaleima on aikaisempi kuin pyyntöjonon ensimmäisenä olevan pyynnön aikaleima, talletetaan uusi pyyntö jonon kärkeen. Aikaisemmin jonon kärjessä olevalle prosessille lähetään kysely (inquire). Jos tämä prosessi ei ole vielä suorittamassa kriittistä osiota, ts. se ei ole vielä saanut pyynnölleen kaikkia vastauksia, lähettää se luopumisviestin (yield). Tällöin luopumisviestin lähettäneen prosessin alkuperäinen pyyntö laitetaan pyyntöjonoon omalle paikalleen, ja sille lähetään aikanaan uusi vastaus. Jos taas kyselyn vastaanottanut prosessi on jo suorittamassa kriittistä osiota, odottaa kyselyn lähettänyt viesti poistumisviestiä kuten tavallisesti. Lopuksi jonon kärkeen sijoitetun uuden pyynnön lähettäjälle lähetetään vastaus.

Juuri algoritmin vikasietoisuus tekee siitä hyvin käyttökelpoisen: järjestelmän vikaantuessa viestien määrä kasvaa hyvin hallitusti, eikä järjestelmä pääse ruuhkautumaan. Vikaantuneiden prosessien korjaantuminen ei myöskään tuota ongelmia, koska edustajisto muodostetaan jokaisessa poissulkemistilanteessa uudestaan.

Agravalin ja el Abbadin esittämä ratkaisu on kehittynyt aimo harppauksen Dijkstran 25 vuotta aiemmin esittämästä ratkaisusta. Ratkaisu on sekä hajautettu että vikasietoinen, mutta tämäkään ei ole viimeinen sana poissulkemisongelman tutkimuksessa.

3 Katsaus viimeisimpään tutkimukseen

Poissulkemisongelma askarruttaa tutkijoita vieläkin, 35 vuotta ensimmäisen ratkaisun julkaisemisen jälkeen. Vuonna 1999 pidetyssä 18. ACM:n *Symposium on Principles of Distributed Computing* -konferenssissa julkaistiin peräti neljä artikkelia tai tutkimustulosta ongelmasta. Eräs näistä tutkimustuloksista [CSR99] keskittyy viestien määrän vähentämiseen quorum-ratkaisussa. Tässä ratkaisussa prosessisolmu, joka lähetettyään *reply*-viestin saa uuden *request*-viestin, lähettää kriittistä aluetta suorittavalle prosessisolmulle erityisen *transfer*-viestin. Kriittistä aluetta suorittava prosessisolmu lähettää tämän *transfer*-viestin perusteella *reply*-viestin suoraan seuraavalle poissulkemista haluavalle prosessille heti, kun se lopettaa kriittisen alueen suorittamisen. Tämä käytäntö vähentää viestien määrää, joten tätä algoritmia käyttämällä on riski viestien katoamisesta johtuvista virheistä pienempi. Toinen samassa konferenssissa julkaistu artikkeli [BCK99] käsittelee vuoromerkkiin perustuvan ratkaisumenetelmän vikasietoisuuden parantamista.

3 Yhteenveto

Keskinäisen poissulkemisen ongelma on ollut mielenkiinnon kohteena jo yli kolme vuosikymmentä. Täysin uusia ratkaisuperiaatteita sekä vanhojen ideoiden optimoituja versioita on julkaistu tasaisesti eri aikoina. Poissulkemisongelma on ongelmana yksinkertainen, mutta jo ensimmäisen ohjelmallisen toteutuksen löytäminen vei aikansa.

Ensimmäisellä ratkaisulla ei ole nykyään käytännöllistä merkitystä. Historiallinen merkitys on kuitenkin suuri: Dijkstran artikkeli lienee yksi viitatuimpia artikkeleita tietojenkäsittelytieteen historiassa. Ratkaisun puutteet - monimutkaisuus ja yhteisen muistin käyttö - innoittivat heti uusien ratkaisujen kehittämistä. 70-luvulla alkanut hajautettujen järjestelmien tutkimus puolestaan edellytti taas täysin uusien ratkaisujen kehittämistä, ja myöhemmin tiukentuneet vaatimukset vikasietoisuudesta koskivat luonnollisesti myös poissulkemisongelman ratkaisua. Viimeisin tutkimus keskittyy tehokkuuteen ja vikasietoisuuden edelleen parantamiseen.

Tietojenkäsittelytieteen lyhyt mutta nopeasti kehittynyt historia on opettanut, miten vaikea tulevaa on ennustaa. Tulevia kehityssuuntia on mahdoton tietää ennalta, mutta poissulkemista tarvittaneen tulevaisuudenkin järjestelmäarkkitehtuureissa, mitä nämä lienevätkään. Lienee siis turvallista ennustaa, että tämä yksi tietojenkäsittelytieteen pitkäikäisimmistä ongelmista tulee tulevaisuudessa vaatimaan yhä uusia ratkaisuja uudenlaisten järjestelmien vaatimusten täyttämiseksi.

Lähteet

AgA91 Agrawal, D. ja el Abbadi, A., An Efficient and Fault-Tolerant Solution for Distributed Mutual Exclusion, ACM Tocs 9,1 (1991)

BCK99 Beaquier J., Ganolini C. ja Kutten S., Optimal Reactive k-stabilization: the Case of Mutual Exclusion. Proceedings of 18th Annual ACM Symposium on Principles of Distributed Computing, 1999. s. 209-218.

CSR99 Cao G., Singhal M. ja Rishe N., Delay-optimal Quorum-based Mutual Exclusion for Distributed Systems. Proceedings of 18th Annual ACM Symposium on Principles of Distributed Computing, 1999. s. 271

Dij65 Dijkstra E.W., Solution of a Problem in Concurrent Programming Control. Comm ACM 8,9 (1965), 569

EiM72 Eisenberg M.A. ja McGuire, M.R., Further Comments on Dijkstra's Concurrent Programming Control Problem. Comm ACM 15,11 (1972)

Knu66 Knuth D.E., Additional Comments on a Problem in Concurrent Programming Control. Comm ACM 9,5 (1966), 321-322

Lam74 Lamport L., A New Solution of Dijkstra's Concurrent Programming Problem. Comm ACM 17,8 (1974)

Pet81 Peterson G.L., Myths about the Mutual Exclusion Problem. Inf. Proc. Lett. 12 (3), 115-116

Ray86 Raynal M., Algorithms for Mutual Exclusion. North Oxford Academic Publishers Ltd, 1986

Ray91 Raynal M., A Simple Taxonomy for Distributed Mutual Exclusion Algorithms, Operating Systems Review 25,2, April 1991

RiA81 Ricart G. ja Agrawala, A.K., An Optimal Algorithm for Mutual Exclusion in Computer Networks, Comm ACM 24,1 (1981)

Tan92 Tanenbaum A. S., Modern Operating Systems. Prentice-Hall 1992.

WWW1 A Survey of Mutual-Exclusion Algorithms for Multiprocessor Operating Systems, Lawrence Kesteloot.
http://tofu.alt.net/~lk/242.paper/242.paper.html