

hyväksymispäivä arvosana

arvostelija

Kellojen synkronointi itsestabiloituvasti

Mikko Pervilä

Helsinki 14.11.2007
seminaarikirjoitelma
HELSINGIN YLIOPISTO
Tietojenkäsittelytieteen laitos

Sisältö

1 Johdanto	1
2 Ongelmakenttä ja aiempia ratkaisuja	2
2.1 Mahdottomuus.....	3
2.2 Helpotukset.....	4
2.3 Cristianin algoritmi.....	5
2.4 Berkeleyyn algoritmi.....	5
2.5 Lamportin aikaleimat.....	6
3 Dolev ja Welch	7
3.1 Oletukset ja merkinnät.....	7
3.2 Kellopulssiin perustuva algoritmi.....	9
3.3 Epäsynchroninen algoritmi.....	12
3.4 Todistukset.....	14
3.5 Ongelmat.....	14
4 Yhteenveto	15
Lähteet	16

1 Johdanto

Itsestabiloituvuus on verrattain vanha [Dij74] ratkaisumalli vielä vanhempaan ongelmaan, vikojen syntymiseen laskentaa suoritettaessa. Matemaattisesti abstraktit mallit ovat onnekkain immuuneja sille laitteistovikojen joukolle, joka voi syntyä fyysisiä tietokoneita käytettäessä. Toisaalta kummatkin ovat alttiita ohjelmointivirheille, joihin oletettavasti törmäämme niin kauan kuin inhimillinen tekijä on osa kehitysprosessia [BrP01].

Vikasietoisuus on käsitteenä verrattain yksinkertainen. Järjestelmän toiminnassa saa syntyä vikoja, mutta ne eivät saa aiheuttaa käyttäjän pyytämän palvelun estävää häiriötä [Avi04]. Itsestabiloituvan järjestelmän perusominaisuus on algoritmisesti koodattu palautumiskyky. Ajatellaan, että kunhan järjestelmä saa toimia moitteetta tarpeeksi kauan, se palautuu suorittamaan tarkoitettua laskentaa. Toisaalta virheitä saa syntyä lähes mielivaltaisella tavalla, jättäen järjestelmän satunnaiseen tilaan. Tärkein rajoitus on se, että virheet eivät saa muuttaa itse palautumiskoodia eivätkä estää sen suoritusta. Virheiden vaikutuskyky on olennaisesti suurempi hajautetuissa järjestelmissä, jossa tilojen määrä moninkertaistuu useampien rinnakkain toimivien järjestelmien myötä.

Itsestabiloitavuus on ratkaisuna olennaisesti monimutkaisempi kuin vikasietoisuus, koska sen käsitteeseen liittyy vikojen sijainnin kaltaisia lisäoletuksia. Oletukset esitetään *ehtoina*, jotka toteuttaessaan järjestelmä on itsestabiloituva. Itse järjestelmä voidaan mallintaa esimerkiksi tilakoneiden verkkona tai päättymättömiä silmukoita suorittavina ohjelmina [Suo07]. Järjestelmän suoritus kuvataan algoritmisesti siten, että jokaisessa suoritusvaiheessa esitetyt ehdot toteutuvat. Varsinainen vaikeus seuraa ehtojen toteutumisen todistamisesta, joka hankaloituu rinnakkaisuutta ja satunnaisuutta käsiteltäessä. Todistusta seurattaessa on helppo unohtaa algoritmin tarkoitus, mutta toisaalta ilman todistusta esitetty algoritmi on kyseenalainen. Itsestabiloituvien järjestelmien yleistymiseksi tarvitaan siis sekä algoritmista lähestymiskykyä, että kykyä hahmottaa kuvaillun algoritmin soveltuvuus osaksi olemassa olevia järjestelmiä. Soveltuvuus perustuu yleensä valitun algoritmin oletusten vertailuun ratkaistavan ongelman ympäristön kanssa. Kyseeseen tulevat esimerkiksi itsestabiloituvan algoritmin palautumiseen käytämä aika, tähän tarvittavien viestien määrä ja koko, sekä mahdolliset todistuksessa tehdyt lisäoletukset.

Kellojen synkronointi on mielenkiintoinen ja yleinen ongelma, johon kenen tahansa

käyttäjän on helppo törmätä: riittää seurata vaikkapa puhelimen ja tietokoneen kelloa minuutin ajan. Laskennan suorituksessa on kuitenkin toistuvia tilanteita, joissa olisi edullista suorittaa jokin vaihe minuuttia paremmalla tarkkuudella yhtä aikaa kaikkialla. Synkronointiin liittyviä ratkaisuja onkin tutkittu ahkerasti. Synkronointiongelma on palautettavissa *konsensusongelmaan* [Väl07], jossa hajautetun järjestelmän yksiköiden pitäisi yhdessä päästä yksimielisyyteen järjestelmän tilasta.

Tämän seminaarityön tarkoitus on esitellä kellojen synkronointiin liittyvä ongelmakenttä [Lam78, DHS86] ja perustella, miksi kellojen tarkka täsmäys on yleisessä (todellisessa) tilanteessa mahdotonta toteuttaa täsmällisesti. Mahdottomuuden välttämiseksi nykyiset tekniikat tekevät oletuksia, joiden avulla synkronointiongelma pystytään rajaamaan *kelvollisesti* ratkeavaksi. Epätarkkuuden hyväksymisen jälkeen käydään lyhyesti läpi joitakin olemassa olevia ratkaisuja, järjestettynä niiden ympäristöä koskevien oletusten vahvuuden mukaan. Ensimmäiset näistä ovat keskitettyä aikapalvelijaa käyttävät Cristianin ja Berkeleyyn algoritmit. Näitä seuraavat hajautetusti toimivat Lamportin aikaleimat. Olemassa olevien algoritmien lisäksi on syytä tutkia itsestabiloituviin algoritmeihin liittyviä oletuksia ja niiden soveltuvuutta todellisiin hajautettuihin järjestelmiin.

Aiempien ratkaisujen kertauksen jälkeen edetään niihin verrattain heikkoihin oletuksiin, joihin Dolevin ja Welchin [DW04] julkaisun esittelemät algoritmit nojaavat. Algoritmien toimintaa esitellään simuloimalla ja intuitiivisesti. Muodolliset oikeellisuustodistukset sivuutetaan tämän työn laajuudessa. Kiinnostuneelle lukijalle esitellään todistusten perusajatus karkeasti ja viitataan tarkemman läpikäynnin sisältäviin töihin. Lopuksi pyritään vielä esittelemään, minkälaisissa olosuhteissa kyseiset algoritmit toimivat huonosti tai eivät ollenkaan.

2 Ongelmakenttä ja aiempia ratkaisuja

Kellojen synkronointi on yleinen ja maallikollekin helposti ymmärrettävissä oleva ongelma, jonka ratkaisut ovat kuitenkin enenevässä määrin monimutkaisia ja siten hankalasti todistettavia. Perusongelman ollessa ratkaisematon fysikaalisessa mielessä, laskennan viitekehyksessä pystytään usein tyytymään ratkaisuihin, jotka sisältävät yleistä tilannetta tiukemmin määriteltyjä oletuksia. Näiden oletusten nojalla eristetty ongelma on siis ratkaisun kannalta helpompi kuin kaikkein yleisin tilanne. Oletusten luonteeseen onkin syytä kiinnittää huomiota, sillä ne sanelevat ympäristön, jossa valit-

tua ratkaisua voidaan käyttää. Oletusten kumoutuessa ei ole syytä odottaa ratkaisunkaan toimivan, mikä on sekä teorian että käytännön kannalta merkittävä rajoitus.

Tässä luvussa esiteltävistä algoritmeista kiinnostuneille suositeltava lähde on Tanenbaumin ja van Maartenin oppikirja [TM02], jota on käytetty muun muassa Tietojenkäsittelytieteen laitoksen kurssilla 582417 Hajautetut järjestelmät.

2.1 Mahdottomuus

Mikäli sivuutetaan hetkeksi suhteellisuusteoria ja oletetaan kyky mitata aikaa täsmälleen oikein kaikkialla, voidaan tätä ”oikeaa aikaa” kutsua *ideaalikelloksi*. Kellojen epätasällisyys johtuu fyysisten kellojen epätäydellisyydestä aiheutuvista poikkeamista, jotka vaihtelevat eri kellojen välillä. Kellon sanotaan joko edistävän tai jättävän jos sen osoittama aika on ideaalikelloa edellä tai jäljessä. Vääristymä toistuu jokaisella kellon osoittamalla askeleella. Askeleen pituus riippuu kellon tarkkuudesta, eli sen kyvystä mitata sekunteja, millisekunteja tai vielä pienempiä yksiköitä. Fyysisen kellon virheeseen voivat vaikuttaa myös kellokohtaiset seikat, kuten ympäristön lämpötila, kellon virtalähteen varaustila ja muut kelloa ympäröivät voimat. Tätä virhettä kutsutaan kellon ajalehtimiseksi (eng. clock drift). Ajalehtimisen seurauksena hyvin pienetkin epätarkkuudet kerääntyvät pidemmällä aikavälillä merkittäväksi poikkeamiksi kahta kelloa keskenään verrattaessa. Korjauksena kellot *synkronoidaan* ajoittain, eli ne asetetaan osoittamaan samaa arvoa. Tällöin kellot näyttävät hetken samaa aikaa, mutta ajalehtiminen alkaa uudelleen heti synkronointihetkestä.

Mielenkiintoiseksi synkronointiongelman tekee sen levinneisyys. Käytännössä jokainen tietokone sisältää kellosirun, ja atomikellot mukaan lukien jokainen kello ajalehtii jonkin verran [TAI07]. Toisaalta poikkeamat ovat erisuuruisia eri kellojen välillä. Ongelmaksi muodostuu siis itse synkronointitapahtuma, tarkemmin menetelmä, jolla kellot saadaan muutettua samalla ideaalikellon hetkellä osoittamaan valittua ajanhetkeä. Hajautetuissa järjestelmissä tämän tiedon välitykseen vaikuttaa vahvasti käytetyn siirtotien viipeet. Tyypillisesti siirtotien viive ei ole ennustettavissa, ja toisaalta sen mittaamiseen tarvitaan kelloa.

Tilannetta pahentavat olennaisesti kelloja käyttävien tietokoneiden kohtaamat virheet, jotka voivat kohdistua myös synkronoinnin suoritukseen. Bysanttilaiset [LSP82] virheet aiheuttavat ilmiön, jossa virheen kohdannut yksilö näyttää pyrkivän tahallaan vääristämään hajautetun järjestelmän laskennan lopputulosta, eli tässä tapauksessa kellojen

synkronoitumista samaan arvoon. Ilmiö on hyvin tunnettu laskennan yhteydessä johtuen sen aiheuttamista kustannuksista. Yleisessä tilanteessa bysanttilaisista virheistä toipuminen edellyttää yli kolminkertaisen enemmistön oikein toimivia järjestelmiä vikaantuneisiin verrattuna. Vastaaviin synkronointiongelmiiin on esitetty vuosien varrella sekä useita eri tilanteisiin liittyviä ratkeamattomuustodistuksia että ratkaisuja helpompiin osaongelmiin [SWL90].

2.2 Helpotukset

Eräs perustavanlaatuinen helpotus on mittauksessa käytettävän nollakohdan siirtäminen koskemaan jotain sovittua, yhteistä ajanhetkeä¹. Tämän ratkaisu on tuttu myös yleisesti käytössä olevista kalentereista. Laskennassa nollahetken sopiminen voidaan toistaa ta-pauskohtaisesti, esimerkiksi jonkin tietyn laskennan vaiheen alkaessa tai järjestelmän käynnistyessä. Nollahetkestä eteenpäin yksiköt toimivat sovitun protokollan mukaisesti, lisäten kelloa mallintavaa laskurin arvoa eteenpäin aina ohjelmakoodissa valittujen ehto-
tojen täytyessä. Toinen mittausta helpottava esimerkki on siirtymä itse *ajan* mittaamisesta valittujen *tapahtumien* mittaamiseen. Tässä tapauksessa oletetaan sovi-
tuksi jokin suoritettavissa olevien toimintojen joukko. Joukkoon kuuluvan toiminnon suorittaminen kasvattaa yksikön laskuria yhdellä. Tyypillinen toiminto on viestin lähe-
tys toiselle yksikölle tai viestin vastaanotto toiselta yksiköltä.

Kolmas merkittävä liennytyys on siirtyminen malliin, jossa kaikkia tapahtumia ei edes mitata. Joissakin järjestelmissä riittää, että vain osasta tapahtumapareja toinen voidaan osoittaa syyksi ja toinen seuraukseksi. Tällöin puhutaan *kausalliteetista*, eli syy-seu-
raus-järjestyksestä kyseisessä mallissa. Itse asiassa tällaisia malleja on useita, ja niitä nimitetään valitun järjestyksen vahvuuden perusteella. Luvussa 2.5. esitellään Lampor-
tin aikaleimojen toteuttama osittaisjärjestys (eng. partial ordering), joka voidaan vahvistaa totaalijärjestykseksi (eng. total ordering). Joukko muita järjestyksuvauksia löytyy esimerkiksi Tanenbaumin ja van Maartenin oppikirjasta [TM02].

2.3 Cristianin algoritmi

Cristianin algoritmin tarkoitus on ottaa huomioon hajautetun järjestelmän siirtotien vii-
ve kelloja synkronoitaessa. Algoritmin toiminnassa erotellaan asiakas, joka tekee synkronointipyynnön, ja palvelija, joka valitsee uuden ajankohdan. Ajatellaan, että

¹ UNIX-yhteensopivien järjestelmien nollahetkeksi lasketaan keskiyö 1.1.1970.

asiakkaan kellonaika on vähemmän luotettava kuin palvelijan, joka voi hakea oman aikansa jostakin luotettavasta lähteestä. Asiakas aikaleimaa pyyntönsä viimeisimmällä mahdollisella hetkellä ennen lähetystä. Palvelija kopioi aikaleiman vastaukseensa. Asiakas lisää uuden aikaleiman vastaukseen heti tämän saapuessa, käyttäen yhä vanhaa kellonaikaansa. Nyt asiakas voi laskea näiden kahden aikaleiman erotuksesta siirtotien viiveen, joka täytyy huomioida uutta kellonaikaa asettaessa.

Algoritmi on siis suunniteltu sietämään hyvin siirtotien epädeterministisiä viipeitä. Samaten algoritmi saadaan helposti sietämään viestien toistumista ja vääristymistä käyttämällä luotettavaa siirtotietä. Tarvittaessa myös palvelijan autentikointi voi tulla kysymykseen. Algoritmi ei kuitenkaan ota kantaa palvelija vikaantumiseen. Asiakkaalle voidaan määritellä useita palvelijoita etukäteen häiriöiden varalle, mutta nyt ongelmaksi muodostuu ylläpidettävien palvelijoiden keskinäinen synkronointi.

2.4 Berkeleyyn algoritmi

Berkeleyyn algoritmi on nimetty BSD (eng. lyh. Berkeley Software Distribution) UNIXin käyttämän kellon synkronointiprotokollan [GZ85a, GZ85b] mukaan. Protokollan toteutus on viralliselta nimeltään TEMPO, mutta se tunnetaan paremmin tiedostonimeltään *timed* (eng. lyh. time daemon). Yksi synkronointiprotokollaa noudattavista järjestelmistä valitaan johtajaksi, jonka jälkeen johtaja pyytää määritellyin aikavälein² muilta järjestelmiltä näiden kellonajat. Mikäli jokin järjestelmä ilmoittaa kellonajan, joka on liian kaukana muista, tätä aikaa ei oteta huomioon. Keräysvaiheen jälkeen johtaja laskee hyväksymistään kellonajoista keskiarvon ja ilmoittaa jokaiselle järjestelmälle kuinka paljon tämän tulee muuttaa kelloaan asettuakseen keskiarvon mukaiseksi. Ilmoittamalla kellonaika muutoksena entiseen kellonaikaan järjestelmä on sietokykyisempi vaihtelevia tiedonsiirtoviipeitä vastaan. On syytä huomata, että keskiarvon käyttö ei takaa minkään järjestelmän olevan algoritmin suorituksen jälkeen ideaalikelloon verrattuna oikeassa ajassa. Itse asiassa on todennäköisempää, että mikään kelloista ei ole oikeassa ajassa.

TEMPO on suunniteltu vikasietoiseksi siten, että se kykenee valitsemaan uuden johtajan entisen kadotessa verkosta. Näin TEMPO sietää myös verkon osittumisen toisistaan erillisiin osiin, joiden sisällä viestiliikenne jatkuu. Mikäli verkot aikanaan yhdistyvät entiselleen, TEMPO osaa poistaa toisen johtajan käytöstä. Nämä tilanteet on ratkaistu

² Oletusarvona neljä minuuttia.

järjestelyllä, jossa johtajaksi pyrkivät kandidaatit lähettävät toisilleen kieltävän vastauksen. Kieltävän vastauksen saapuessa molemmat kandidaatit palaavat odottamaan satunnaisesti valitun ajan ennen uutta yritystä. Todennäköisyys, jolla kaksi järjestelmää yrittää uudelleen samalla ajanhetkellä, on laskettu Gusellin ja Zattin julkaisussa [GZ85b].

Berkeleyn algoritmin oletuksiin on jo alkuperäisessä julkaisussa kirjattu oletus siitä, ettei verkossa esiinny vihamielisesti toimivia järjestelmiä. Algoritmi ei siten kestä bysanttilaisia virheitä. Tämän lisäksi osittuneen verkon palautumisesta aiheuttava kahden johtajan tilanne on havaittavissa vasta, kun jokin uusi järjestelmä liittyy verkkoon ja lähettää pyynnön rekisteröityä johtajalle. Kunnes näin käy, tilanne pysyy vakaana: verkon aikaisempiin osioihin kuuluneet järjestelmät kommunikoivat kukin oman johtajansa kanssa.

2.5 Lamportin aikaleimat

Lamportin vuonna 1978 julkaisemassa työssä [Lam78] lähestytään oikean ajan mittauksen sijaan riittävää matemaattista *kuvausta*, joka riittää mallintamaan tapahtumien välisiä riippuvuuksia. Nimitys tälle on ”tapahtui-ennen-kuvaus” (eng. happened before relation [sic]), ja sitä merkitään nuolella \rightarrow . Kun kahden tapahtuman välillä on mainittu kuvaus, merkitään esimerkiksi **lähetys** \rightarrow **vastaanotto** ja luetaan ”lähetys tapahtui ennen vastaanottoa”. Merkittävin seikka tämän kuvauksen käytössä on se, että kaikkia hajautetun järjestelmän tapahtumia ei aseteta kuvauksen avulla järjestykseen. Jos kahden prosessin tapahtumat eivät liity toisiinsa, niiden välillä ei ole kuvausta ja tapahtumia pidetään rinnakkaisina.

Kuvauksen lisäksi Lamportin aikaleimat toteuttava järjestelmä käyttää *loogista* kelloa, joka ei ole suoraan sidoksissa ”todelliseen” eli *fysiseen* kelloon. Loogisen kellon tarkoitus on palauttaa monotonisesti kasvavia järjestysnumeroita kullekin tapahtumalle, jonka järjestys halutaan pystyä osoittamaan kaikkien järjestelmien laajuudessa. Olennaisesti jokainen järjestelmän sisältää oman loogisen kellonsa. Lisäksi kellon laskuri kasvaa vain tapahtumien *aikaleimauksen* yhteydessä, ei siis itsekseen.

Kuvauksen ja loogisen kellon yhdistämällä voidaan toteuttaa protokolla, joka riittää keskinäiseen poissulkemiseen ja osajärjestelmien synkronointiin. Protokollaa noudattavat järjestelmät asettavat aina viestin saapuessa oman loogisen kellonsa arvon vähintään yhtä suureksi kuin saapuneen viestin aikaleiman. Näin taataan, että viestin vastaanotto

tapahtuu kaikissa järjestelmissä viestin lähetyksen jälkeen. Poissulkeminen onnistuu lisäämällä saapuva viesti aina käsittelyjonoon ja kuittaamalla viesti kaikille muille järjestelmille. Käsittelyyn hyväksytään vain sellaiset viestit, jotka ovat aikaleimansa perusteella jonon kärjessä ja joihin on saatu kuittaus kaikilta muilta järjestelmiltä.

Lampordin aikaleimoja käyttävä järjestelmä takaa viestien järjestyksen niin kauan kuin jokainen hajautetun järjestelmän osa toteuttaa protokollan omalta osaltaan. Kääntäen, mikäli yksikin järjestelmän kelloista alkaa edistää holtittomasti, johtaa tämä järjestelmän kaikkien kellojen edistämiseen [SWL99] aina viestien vaihdon yhteydessä. Lisäksi protokolla ei ota kantaa siihen, miten hajautettu järjestelmä koostetaan toimintaan osallistuvista yksiköistä. Toiminta kuitenkin edellyttää jokaiselta osallistujalta tietoa kaikista naapureistaan, ja etenkin näiden katoamisesta. Muuten kadonnut solmu voi puuttuvilla kuittauksillaan estää viestejä etenemästä pois käsittelyjonosta.

3 Dolev ja Welch

Dolevin ja Welchin esityksessä hajautettu järjestelmä koostuu prosesseista P_i , joiden lukumäärää kuvaa muuttuja n . Esiteltyt kaksi itsestabiloituvaa algoritmia eroavat olennaisimmalta osaltaan siinä, että ensimmäinen niistä (A1) olettaa hajautetun järjestelmän välittävän kaikille prosesseilleen synkronoidun kellopulssin tasaisin aikavälein. Toinen algoritmeista (A2) ei tee tätä oletusta. Algoritmien käsittelyyn tämän työn laajuudessa tarvittavat muut oletukset ja merkinnät ovat seuraavat.

3.1 Oletukset ja merkinnät

Järjestelmän prosesseista osa voi vikaantua mielivaltaisella tavalla. Vikaantuneiden lukumäärää kuvataan muuttujalla f . Dolevin, Halpernin ja Strongin tunnettu todistus [DHS86] osoittaa, että hajautettu järjestelmä voi toipua bysanttilaisesta virheestä jos prosessien lukumäärän suhteen epäyhtälö $n > 3f$ pätee. Vikaantuneiden prosessien lukumäärää käytetään algoritmin toiminnassa epäsuorasti siten, että arvoa f ei suoraan tunneta. Toisaalta prosessien lukumäärä voidaan olettaa tunnetuksi. Mikäli vikaantuneiden lukumäärä kasvaa lukuun $n/3$ (tai sen yli), algoritmi ei voi enää toimia oikein em. todistuksen nojalla. Siis muuttujan f tarkkan arvon sijaan voidaan käyttää ylärajaa $n/3$.

Jokaisella prosessilla on käytettävissään fyysinen kello, jonka tarkkuus on M_{pc}^3 bittiä.

3 Kello pystyy myös osoittamaan tyhjää arvoa *nil*.

Fyysisen kellon oletetaan ajalehtivan korkeintaan muuttujan ρ (rho) verran, eli jokainen askel ideaalikellolla mitattuna suoritetaan vähintään ajassa $1-\rho$, ja enintään ajassa $1+\rho$. Oletusta tarvitaan arvioitaessa viestien perillemenoon kuluva maksimiaikaa. Rhon arvo on kaikkien prosessien tiedossa.

Viestien välitykseen kuluvat ajat kiinnitetään kahden muuttujan avulla. Nämä ovat arvioitu tiedonsiirron kokonaisviive \mathbf{d} (eng. lyh. delay) ja arviointivirhe ϵ . Yhden viestin lähetykseen tai vastaanotto voi siis tapahtua vähintään ajassa $d-\epsilon$ ja enintään ajassa $d+\epsilon$. Kokonaisviive sekä maksimivirhe ovat jälleen kaikkien prosessien tiedossa.

Jokaisella prosessilla on lisäksi looginen kello, joka ottaa syötteenään prosessin fyysisen kellon ja palauttaa arvon suljetulta väliltä $[0, M_{ic}-1]$. Loogisen kellon tarkkuus ei välttämättä ole sama kuin fyysisen kellon, ja looginen kello voidaan toteuttaa pelkillä kokonaisluvulla. Lisäksi loogisen kellon osoittamat arvot lasketaan jakojäännösten avulla siten, että maksimiarvoa $M_{ic}-1$ seuraava arvo on 0. Jakojäännösten käyttö on perusteltua, sillä mielivaltaisesta tilasta toimintansa aloittava järjestelmä voisi muuten päätyä pian suurimpaan mahdolliseen kokonaislukuun. Tällöin kellon arvot loppuisivat heti kesken.

Algoritmien toiminnan sisäistämistä voi vielä helpottaa oivallus, joka koskee prosessien kellonaikojen säilytystä. prosessit eivät nimittäin tallenna tietoa muiden prosessien kellonajoista edelliseltä kierrokselta. Tähän palataan algoritmien satunnaisuutta käsiteltäessä.

3.2 Kellopulssiin perustuva algoritmi

```

01 when pulse occurs:
02   broadcast clocki
03   collect clock values until  $(1 + \rho)(d + \varepsilon)$  time has elapsed on the physical clock
04   if  $|\{j \mid \text{clock}_j = \text{clock}_i\}| < n - f$  then (case 1)
05     {clocki ← 0; last_incrementi ← false}
06   else (case 2)
07     if clocki ≠ 0 then (case 2.1)
08       { clocki ← ( clocki + 1) mod Mlc; last_incrementi ← true}
09     else (case 2.2)
10       if last_incrementi = true then (case 2.2.1) clocki ← 1
11       else (case 2.2.2) clocki ← toss(0, 1)
12       if clocki = 1 then last_incrementi ← true
13       else last_incrementi ← false

```

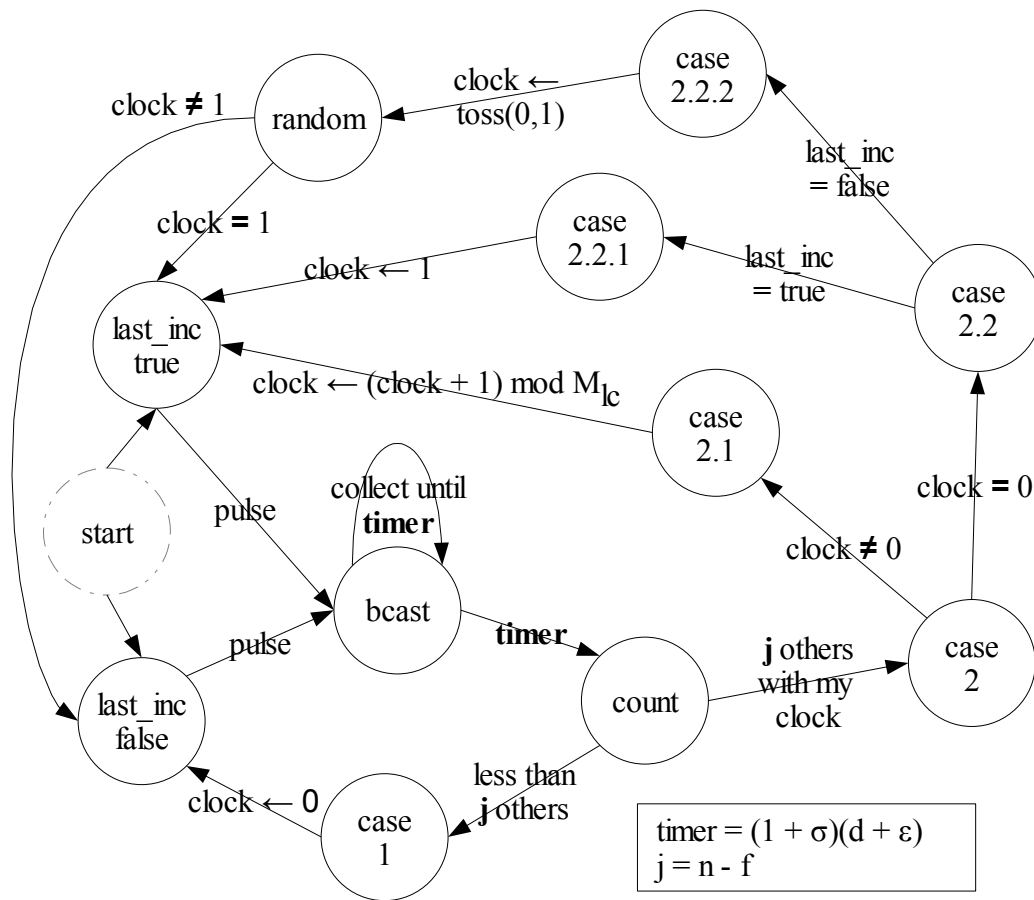
Kuva 1. Synkronoituun pulssiin perustuvan algoritmin (A1) toiminta ohjelmakoodina.

Synkronoitua pulssia käyttävän algoritmin (A1) toiminta on kuvattu ohjelmakoodina kuvassa 1. Aina pulssin saapuessa oikein toimivat prosessit monilähettävät oman kellonsa arvon kanavaan ja vastaanottavat muilta prosesseilta näiden kellonaikoja ajastimen laukeamiseen asti. Ajastimen arvoa $(1 + \rho)(d + \varepsilon)$ laskettaessa käytetään ajalehtimisen kattoa ρ , tiedonsiirron kokonaisviivettä d ja arviointivirhettä ε . Oletusten nojalla tämän jälkeen viestiliikenteessä ei voi enää olla saapumattomia viestejä. Mikäli viesti kuitenkin myöhästyy enemmän kuin odotettua, se voidaan yksinkertaisesti jättää käsittelemättä.

Toiminnan kannalta tärkein päätös on nyt vastaanotettujen kellonaikojen vertailu omaan kellonaikaan. Mikäli oma arvo löytyy vastaanotetuista viesteistä alle $n/3$ kertaa, tulkitaan oma kellonaika virheelliseksi (case 1). Kannattaa huomata, että ohjelmakoodissa on käytetty tarkempaa merkintää $n-f$. Mikäli vikaantuneiden prosessien lukumäärä on tunnettu, sitä voidaan myös käyttää.

Muuttuja last_increment_i kuvaa edellisen kierroksen tilannetta prosessissa i . Mikäli prosessi on edellisellä kierroksella kasvattanut kellon arvoa, on tilanne ollut prosessin tulkinnan mukaan ”kunnossa” (case 2.1). Algoritmin toimintaa havainnollistaa myös kuva 2, jossa toiminta on esitetty tilakoneena.

A1 pyrkii toiminnassaan nollaamaan loogisen kellon aina, kun prosessi pitää omaa kel-



Kuva 2: Synkronoituun pulssiin perustuvan algoritmin (A1) toiminta tilakoneena. Muuttujan last_increment (tässä lyhennetty last_inc) tila muistetaan jokaisella pulssin käynnistämällä kierroksella.

loaan viallisena. Kellon nollassa täytyy kuitenkin erottaa myös tilanne (case 2.2.1), jossa edellisellä kierroksella oltiin saavutettu maksimiarvo. Muuttujaa last_increment käytetään merkinä tästä.

Mihin tarvitaan askeleessa 11 (case 2.2.2) tapahtuvaa kolikonheittoa? Alustavasti vaikuttaa siltä, että A1 toimii ilman satunnaisuuttakin. Toiminnassa voi kuitenkin syntyä sykli, joka johtuu nolla-arvojen tulkinnasta. Mikäli satunnaisuus jätettäisiin pois, olisi kuvassa 3 esitetty suoritus mahdollinen. Bysanttilaisesti vikaantuneeksi oletetaan neljäs prosessi, jota merkitään P_4 . Koska prosessit eivät talleta muiden prosessien kellonaikoja koskevaa tietoa, huomataan että lopputilanne on sama kuin alkutilanne. Neljäs prosessi voisi nyt kahta eri viestiä vuorottelemalla vangita järjestelmän kuvattuun sykliin.

prosessit	P_1	P_2	P_3	P_4
alkutilanne	<u>0</u>	<u>0</u>	<u>1</u>	<u>2</u>
<hr/>				
1. pulssi				
lähetys		$P_1 \leftarrow 0$	$P_2 \leftarrow 1$	$P_1 \leftarrow 1$
	$P_2 \leftarrow 0$		$P_2 \leftarrow 1$	$P_2 \leftarrow 0$
	$P_3 \leftarrow 0$	$P_3 \leftarrow 0$		$P_3 \leftarrow 1$
	$P_4 \leftarrow 0$	$P_4 \leftarrow 0$	$P_4 \leftarrow 1$	
<hr/>				
vastaanotto		$tila_1 \leftarrow 0$	$tila_1 \leftarrow 0$	
	$tila_2 \leftarrow 0$		$tila_2 \leftarrow 0$	
	$tila_3 \leftarrow 1$	$tila_3 \leftarrow 1$		
	$tila_4 \leftarrow 1$	$tila_4 \leftarrow 0$	$tila_4 \leftarrow 1$	
<hr/>				
ajastin laukeaa				
tieto muista	<u>0011</u>	<u>0010</u>	<u>0011</u>	
n-f kpl oma aika?	ei	kyllä	ei	
vertailun tulos	$clock_1 \leftarrow 0$	$clock_2 \leftarrow 1$	$clock_3 \leftarrow 0$	
välitilanne	<u>0</u>	<u>1</u>	<u>0</u>	
<hr/>				
2. pulssi				
lähetys		$P_1 \leftarrow 1$	$P_2 \leftarrow 0$	$P_1 \leftarrow 1$
	$P_2 \leftarrow 0$		$P_2 \leftarrow 0$	$P_2 \leftarrow 1$
	$P_3 \leftarrow 0$	$P_3 \leftarrow 1$		$P_3 \leftarrow 0$
	$P_4 \leftarrow 0$	$P_4 \leftarrow 1$	$P_4 \leftarrow 0$	
<hr/>				
vastaanotto		$tila_1 \leftarrow 0$	$tila_1 \leftarrow 0$	
	$tila_2 \leftarrow 1$		$tila_2 \leftarrow 1$	
	$tila_3 \leftarrow 0$	$tila_3 \leftarrow 0$		
	$tila_4 \leftarrow 1$	$tila_4 \leftarrow 1$	$tila_4 \leftarrow 0$	
<hr/>				
ajastin laukeaa				
tieto muista	<u>0101</u>	<u>0101</u>	<u>0100</u>	
n-f kpl oma aika?	ei	ei	kyllä	
vertailun tulos	$clock_1 \leftarrow 0$	$clock_2 \leftarrow 0$	$clock_3 \leftarrow 1$	
lopputilanne	<u>0</u>	<u>0</u>	<u>1</u>	

Kuva 3: Algoritmin A1 toiminta ilman satunnaisuutta. Virheellisesti toimiva prosessi 4 aiheuttaa syklin hajautetun järjestelmän toiminnassa: alku- ja lopputilanne ovat samat.

Dolev ja Welch todistavat, että odotusarvoisesti Algoritmi A1 päättyy $(M_{lc} + 2) \cdot 2^{2(n-f)}$ synkronointipulssin välein tilaan, jossa kaikkien (oikein toimivien) prosessien loogisen kellon arvo on 1. Eksponentiaalisesta laskukaavasta voidaan huomata, että vaadittu pulssien määrä kasvaa nopeasti prosessien määrän kasvaessa. Algoritmi soveltuu siis etenkin ympäristöihin, joissa yksikin prosessi riittää suorittamaan halutun laskennan. Muiden prosessien tarkoitus on tällöin parantaa vikasietoisuutta ylimäärän [Per06] avulla. Käyttämällä Knuthin esittelemää *kiinalaisen jakojäännöksen laskuria* [Knu81] synkronoitumiseen vaadittavien pulssien lukumäärä voidaan osoittaa pienemmäksi. Täl-

lä menetelmällä 64 bittiä käyttävän ($M_{ic} = 2^{64}$) loogisen kellon stabiloitumisen ylärajaksi saadaan $381 \cdot 2^{2(n-f)}$ pulssia.

3.3 Epäsynkroninen algoritmi

```

01 if last_hop_timeri = 0 then
02   if last_avg_timeri = 0 and clocki ≤ δ then
03     choicei ← “average”
04   else choicei ← “hop”
05   broadcast “clock-request” message
06   let Si be multiset of clock values collected until 2(1 + ρ)(d + ε) time has elapsed
       on the physical clock
07   if n - f elements of Si are within δ of clocki then
08     Si ← reduce( Si , clocki , f )
09     if choicei = “average” then (*do the averaging procedure*)
10       clocki ← midpoint( Si )
11       last_avg_timeri ← Ta (*set physical clock timer - counts down to 0*)
12     else (*do the hopping procedure*)
13       clocki ← random( Si )
14       last_hop_timeri ← Ts (*set physical clock timer - counts down to 0*)
15   else (*less than n - f clock values are within clocki*)
16     clocki ← random(Si )
17   last_hop_timeri ← Ts

```

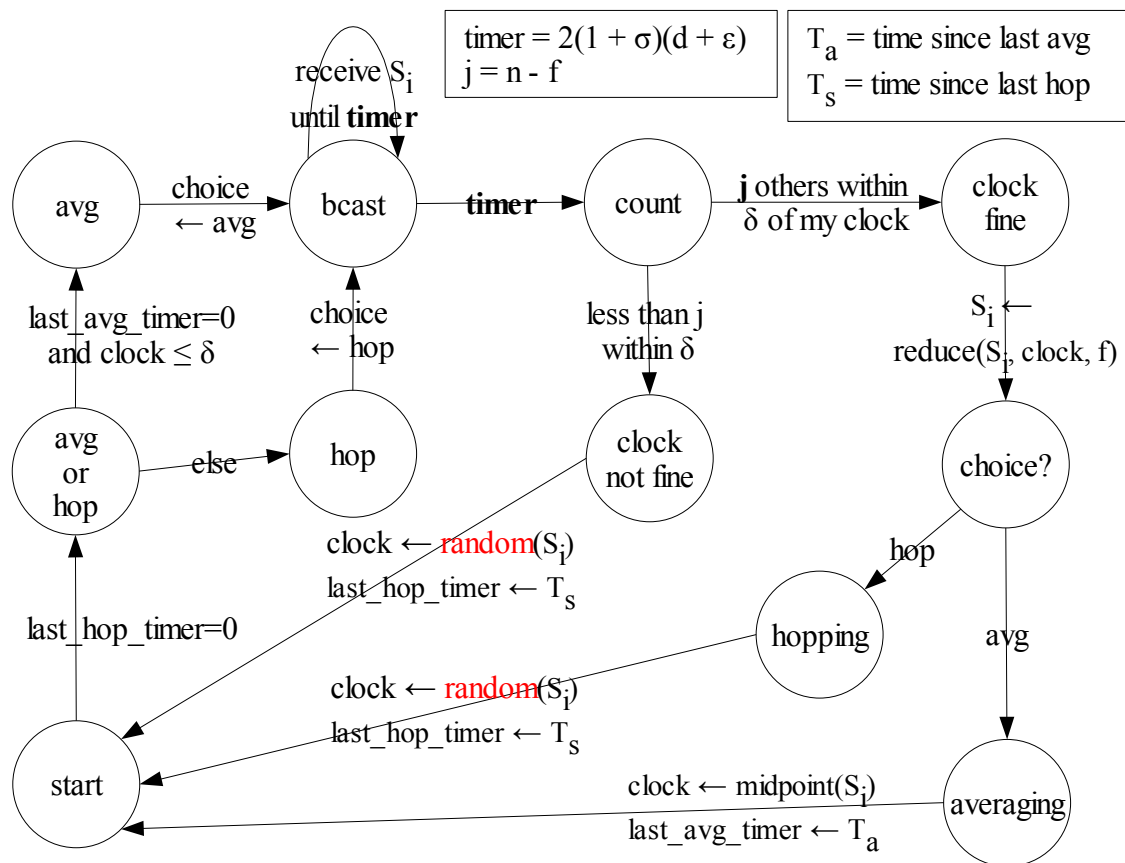
Kuva 4: Kellojen synkronointi ilman keskitettyä kellopulsia.

Kellopulsista toimivan algoritmin A2 toiminta perustuu Dolevin ja Welchin aiemmassa työssä [DHS86] julkaistuun vikasietoiseen keskiarvofunktioon. Funktio poistaa kelloaikojen joukosta f suurinta ja f pienintä tulosta, sekä tämän jälkeen laskee keskiarvon jäljelle jääneistä arvoista. Tarkoituksena on saada aluksi kellonajat *osittain* synkronoitua, eli riittävän lähelle toisiaan varsinaisen synkronoinnin onnistumiseksi. Lopputuloksena funktio puolittaa arvoalueen jolle kellonajat sijoittuvat.

Kuvassa 4 esitettyä algoritmia noudattavat prosessit käyttävät fyysistä kelloaan mitataksseen aikavälejä, joiden sisällä prosessi voi joko käyttää em. keskiarvofunktiota (averaging) tai sitten valita satunnaisesti uuden kellonajan vastaanottamiensa arvojen joukosta (hopping). Päätös näiden kahden toiminnon välillä tehdään vastaanotettujen kellonaikojen perusteella seitsemännessä askeleessa. Apuna käytetään valittua parametria δ (delta), joka määrittelee suurimman mahdollisen kellonaikojen välisen eron.

Lisäksi muuttujien T_a ja T_s valinta vaikuttaa kahden eri synkronointitoiminnon kestoihin. Liian pitkät arvot aiheuttavat kellojen ajelehtimista erilleen, kun taas liian pienet lisäävät tarpeetonta viestiliikennettä.

Valintaehtoien havainnollistamiseksi algoritmin toiminta esitetään myös tilakoneena kuvassa 5. Jokaisella kierroksella algoritmia suorittava prosessi monilähettää oman kellonaikansa ja vastaanottaa kaikkien muiden prosessien kellonajat. Mikäli kellonaikojen joukosta ei löydy $n-f$ kappaletta arvoa korkeintaan erotuksella δ , prosessi tulkitsee oman kellonaikansa ajelehtineen liian kauaksi. Tällöin kellonajaksi asetetaan satunnaisesti jokin vastaanotetuista arvoista.



Kuva 5: Algoritmin A2 toiminta tilakoneena. Muuttujan choice arvo muistetaan jokaisella kierroksella.

On epätriviaalia ymmärtää, miksi satunnaisesti valittu kellonaika johtaa prosessien osittaiseen synkronoitumiseen. Oikeille jäljille voi johdattaa havainto siitä, että prosessi valitsee kellonaikansa uuden arvon satunnaisesti kunnes uuden arvon ”vahvistaa” seuraavalla kierroksella $n-f$ kappaletta lähelle sijoittuvaa kellonarvoa.

Synkronoimattoman algoritmin tapauksessa stabiloitumiseen kuluvan ajan odotettu ylä-

raja on $T_a n^{6(n-1)}$ mittayksikköä. Tässäkin tapauksessa merkittävin tekijä on järjestelmän prosessien lukumäärä.

3.4 Todistukset

Dolevin ja Welchin algoritmien oikeellisuustodistukset nojaavat vahvasti heidän aikaisempiin julkaisuihinsa [DIM91, DIM95]. Valitun artikkelin pituuden ollessa jo itsessään 20 sivua, ei todistusten yksityiskohtainen läpikäynti ole mahdollista.

Oleennaista molempien algoritmien oikeellisuustodistuksissa on toiminnan jakaminen *peliksi*, jossa pelaajina ovat algoritmin oikeaan toimintaan pyrkivä *onni* ja mahdollisimman huonoon lopputulokseen pyrkivä *vastustaja*. Vastustaja valitsee prosessien suoritusvuorot, eli järjestyksen jossa ne toteuttavat algoritmia. Todistuksessa pyritään näyttämään, että jokaisella vuorolla onni pystyy vaikuttamaan algoritmissa käytettyyn satunnaisuuteen siten, että äärellisessä ajassa suoritus saadaan järjestelmän toiminnan kannalta hyväksyttävään tilaan.

3.5 Ongelmat

Todistusten osoittamat pahimmat tapaukset algoritmien toipumiselle rajoittavat tuntuvasti niiden käytettävyyttä väljästi (eng. *sparsely coupled*) hajautetuissa järjestelmissä. Ensimmäinen, keskitettyyn pulssiin perustuva algoritmi, näyttäisi alustavasti olevan toteutettavissa vain hyvin tiiviisti hajautetuissa ympäristöissä [DW93], tarkoittaen lähinnä symmetristä multiprosessointia. Toteutettavuus riippuu olennaisesti hajautettujen prosessien lukumäärästä, jonka seurauksena stabiloitumiseen vaadittu viestien lukumäärä kasvaa nopeasti. Tämän lisäksi keskitetyn kellopulssin olemassaolo tarkoittaa käytännössä prosessien sitomista lähes piirilevytasolle. On mahdollista, että toinen algoritmi, joka ei nojaa keskitettyyn kellopulssiin, voisi olla laajemmin käytettävissä.

Molemmissa algoritmeissa prosessien täytyy kyetä lähettämään ja vastaanottamaan viestejä kaikille naapureilleen. Viestinvälitykseen vaadittujen kanavien lukumäärä kasvaa siis myös prosessien lukumäärän myötä. Vaikka kanavien lukumäärää voitaisiin pienentää esimerkiksi viestejä reitittämällä naapurilta toiselle, täytyy tämäkin rajoite huomioida toteutusvaiheessa viestiliikenteen viipeitä arvioidessa.

4 Yhteenveto

Olemassa olevien, kellojen synkronointiin liittyvien algoritmien tutkinta paljastaa samankaltaisuuksia sekä vanhojen että aivan uusienkin ratkaisujen välillä. Perusongelman ollessa ratkeamaton hyvinkin pienten yksityiskohtien, eli valittuun ongelmaan liittyvien oletusten, muutokset aiheuttavat merkittäviä eroja algoritmien toiminnassa käytännössä. Esitellyt algoritmit pyrkivät ratkaisemaan joitakin synkronointiin liittyvistä osaongelmista. Toisaalta yhden algoritmin ei olekaan tarve ratkaista koko ongelmaa, sillä Dolevin ja Welchin tapaan oikeaksi todistettujen algoritmien ominaisuuksia voidaan yhdistellä uusien ratkaisujen luomiseksi. Tällä tavoin laajempi ongelma puretaan paloiksi ja pyritään ratkaisemaan vaiheittain.

Näin koostetun järjestelmän toiminnassa on kiinnitettävä erityisesti huomiota vaiheittain tehtyjen oletusten toteutettavuuteen. Etenkin on pidettävä huoli siitä, että tehdyt oletukset eivät ole keskenään ristiriitaisia. Mutta myös lievemmin on pyrittävä pitämään silmällä algoritmin yleistä toteutettavuutta, sillä ratkeavaksikin osoitetut välivaiheet voivat osoittautua hankalaksi ohjelmoida. Täten uusissakin töissä voidaan palata jo ratkaistuihin ongelmiin, jos on oletettavissa, että uusi ratkaisu on olennaisesti joltakin osaltaan parempi kuin vanha. Tavoiteltavia ominaisuuksia voivat olla toteutuksen helpouden lisäksi ratkaisun yleinen ymmärrettävyys tai jotkin ulkoiset tekijät, jotka vaikuttavat ohjelman suoritukseen todellisissa järjestelmissä.

Lähteet

- Avi04 Avižienis, A., et al. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1,1(2004), sivut 11-33.
- BrP01 Brown, A., Patterson, DA., To Err is Human. *Proc. of the First Workshop on Evaluating and Architecting System dependability*, Göteborg, Ruotsi, heinäkuu 2001. [Myös <http://roc.cs.berkeley.edu/papers/easy01.pdf>, 11.10.2007].
- DHS86 Dolev, S., Halpern, J.Y., Strong, H.R., On the possibility and impossibility of achieving clock synchronization. *Journal of Computer and System Sciences*, 32,2(huhtikuu 1986), sivut 230-250.
- Dij74 Dijkstra, E. W., Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 17,11(1974), sivut 643–644.
- DIM91 Dolev, S., Israeli, A., Moran S., Uniform Dynamic Self-Stabilizing Leader Election. *Proc. of the 5th International Workshop on Distributed Algorithms*, Springer-Verlag, London, UK, 1991, sivut 167-180.
- DIM95 Dolev, S., Israeli, A., Moran, S. Analyzing expected time by scheduler-luck games. *IEEE Transactions on Software Engineering*, 21,5(toukokuu 1995), sivut 429-439.
- DW93 Dolev, S., Welch, J.L., Wait-free clock synchronization. *Proc. of the 12th ACM Symposium on Principles of Distributed Computing*. ACM, New York, USA, 1993, sivut 97-108. Myös *Algorithmica* 18(1997), sivut 486-511.
- DW04 Dolev, S., Welch, J.L., Self-stabilizing clock synchronization in the presence of Byzantine faults. *Journal of the ACM*, 51,5(syyskuu 2004), sivut 780-799.
- GZ85a Gusella, R., Zatti, S., The Berkeley 4.3BSD time synchronization protocol: protocol specification. *Report No. UCB/CSD 85/250*, University of California, Berkeley, kesäkuu 1985.
- GZ85b Gusella, R., Zatti, S., An election algorithm for a distributed clock syn-

- chronization program. *Report No. CSD-86-275*, University of California, Berkeley, joulukuu 1985. [Myös <http://digitalassets.lib.berkeley.edu/techreports/ucb/text/CSD-86-275.pdf>, 25.10.2007].
- Knu81 Knuth, D. E., *The art of computer programming*, Vol. 2, toinen painos. Addison-Wesley, Reading, Massachusetts, USA, 1981.
- Lam78 Lamport, L., Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21,7(heinäkuu 1978), sivut 558-565.
- LSP82 Lamport, L., Shostak, R., Pease, M., The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4,3(heinäkuu 1982), sivut 382-401.
- Per06 Pervilä, M., Luotettavuus vikasietoisin toisinnoin. Kandidaatintutkielma, Tietojenkäsittelytieteen laitos, Helsingin Yliopisto, joulukuu 2006.
- Suo07 Suomela, J., Itsestabilointi: perusmääritelmiä ja klassisia tuloksia. Seminaarikirjoitelma, Hajautetut algoritmit, syksy 2007, <http://www.cs.helsinki.fi/u/jkivinen/opetus/seminaarit/hajalg/suomela.pdf>. [25.10.2007]
- SWL90 Simons, B., Welch, J. L., Lynch, N. A., An overview of clock synchronization. *Proc. of the Asilomar Workshop on Fault-Tolerant Distributed Computing*, Springer-Verlag, Lontoo, UK, 1990, sivut 84-96. [Myös <http://faculty.cs.tamu.edu/welch/papers/lncs90.pdf>, 25.10.2007].
- TAI07 International Atomic Time (TAI), Bureau International des Poids et Mesures, <http://www.bipm.org/en/scientific/tai/tai.html>. [29.10.2007]
- TM02 Tanenbaum, A.S., Maarten van, S. *Distributed Systems: Principles and Paradigms*. Prentice Hall, 2002.
- Väl07 Välimäki, N., Konsensusongelma hajautetuissa järjestelmissä. Seminaarikirjoitelma, Hajautetut algoritmit, *toistaiseksi julkaisematon*.