

Jaetun muistin muuntaminen viestinvälitykseksi

Otto Räsänen

Helsinki 10.10.2007

Hajautetut algoritmit -seminaarin kirjallinen työ

HELSINGIN YLIOPISTO

Tietojenkäsittelytieteen laitos

Sisältö

1 Johdanto	1
2 Valtuuden välitys	2
2.1 Määritelmiä	3
2.2 Valtuuden välitys käyttäen laskuria ilman ylärajaa	4
2.3 Tilavaativuuden alaraja	6
2.4 Valtuuden välitys, kun laskurilla on yläaraja	7
2.5 Jaetun muistin simulointi käyttäen itsestabiloituvaa valtuuden välitystä	9
Lähteet	11

1 Johdanto

Kirjoitelman pääasiallinen lähde on Shlomi Dolevin kirja *Self-Stabilization* [Dol00], ja erityisesti sen luku 4.2. Tarkoituksena on esitellä, kuinka jaettua muistia käyttävään järjestelmään suunniteltu itsestabiloituva algoritmi voidaan muuntaa toimivaksi viestinvälitystä käyttävässä järjestelmässä.

Hajautetut algoritmit voivat toimia useissa erilaisissa järjestelmissä. Voidaan käyttää jaettua muistia tai viestinvälitystä, voidaan käyttää keskitettyä vuorottajaa, hajautettua vuorottajaa, tai lomitettuja luku- ja kirjoitusoperaatioita. Jotta hajautetun algoritmin laatiminen olisi helpompaa, muunnos mallista toiseen pitäisi tapahtua automaattisesti. Algoritmin suunnittelija voi näin valita ongelman kannalta luontevimman mallin ja muuntaa algoritmin toimimaan erilaisissa järjestelmissä.

Monet itsestabiloituvat algoritmit olettavat, että käytössä on hajautettu tai keskitetty demoni, joka jakaa suorittimille suoritusvuoroja. Hajautettu demoni valitsee toistuvasti joukon suorittimia suoritusvuoroon. Yhden vuoron aikana kukin valituista suorittimista lukee naapureidensa tilat ja kirjoittaa oman tilansa. Muuttuneen tilan kirjoittaminen tapahtuu vasta, kun kaikki suorittimet ovat saaneet lukuoperaationsa päätökseen. Tämän jälkeen demoni valitsee uuden joukon suorittimia suoritusvuoroon.

Keskitetty demoni on erikoistapaus hajautetusta demonista. Keskitetty demoni valitsee kerralla suoritusvuoroon vain yhden suorittimien kerrallaan. Vastaavasti synkroninen järjestelmä, jossa kaikki suorittimet aktivoidaan joka kerta, on myös erikoistapaus hajautetusta demonista.

Keskitetyn demonin käyttö on ilmeisesti perua Dijkstralta [Dij74], joka aloitti itsestabiloituvuuden tutkimuksen. Oletus keskitetyn demonin olemassaolosta myös helpottaa algoritmin suunnittelua, koska kommunikaatiorekistereiden arvoja ei tarvitse kopioida talteen paikallisiin muuttujiin.

Muunnos keskitetyn demonin käytöstä viestinvälitykseen voidaan tehdä käyttäen itsestabiloituvaa keskinäistä poissulkemista tai valtuuden välitystä (*token passing*) käyttäen. Vain yhdellä suorittimella kerrallaan on suoritusvuoro tai valtuus, jolloin se voi lukea naapureidensa tilat ja muuttaa omaa tilaansa niiden perusteella. Kun suoritin on muuttanut oman tilansa, antaa se suoritusvuoron tai valtuuden jollekin toiselle suorittimelle.

2 Valtuuden välitys

Viestinvälitystä suorittimien väliseen kommunikointiin käytävässä järjestelmässä suorittimet on yhdistetty toisiinsa käyttäen linkkejä. Viestit linkissä kahden suorittimen välillä voivat korruptoitua, joten linkkien sisältö ja suorittimien tilat saattavat olla täysin mielivaltaisia. Itsestabiloituvan algoritmin tehtävänä on päästä lailliseen tilaan mistä tahansa alkutilasta. Kun järjestelmän tilaksi ymmärretään suorittimien tilojen lisäksi myös linkkien sisältö, pystyy itsestabiloituva algoritmi pääsemään lailliseen myös viestinvälitystä käytävässä järjestelmässä.

Linkissä voi olla useita viestejä kerralla matkalla suorittimelta toiselle. Matkalla olevien viestin lukumäärälle ei välttämättä voida asettaa mitään ylärajaa, kuten ei viestin perillemenon kestollekaan. Niinpä tilojen lukumäärä, joista järjestelmän pitää stabiloitua, voi olla äärettömän suuri. Algoritmin suunnittelu suoraan viestin välitystä varten on vaikeampaa kuin atomisille luku- ja kirjoitusoperaatioille, joten muunnos näiden kahden välillä on hyödyllinen algoritmeja suunniteltaessa.

Muunnosta varten tarvitaan linkkikerroksen algoritmi, joka myös on itsestabiloituva. Linkkikerroksen algoritmi siirtää tietoa kahden suorittimen välillä, joista toinen on *lähettäjä* ja toinen *vastaanottaja*. Viestejä ei saisi kadota, niiden sisältö ei saisi muuttua, eivätkä ne saisi monistua. Esimerkiksi *stop-and-wait* -algoritmi ratkaisee ongelman lähettämällä viestin ja odottamalla vastaanottajalta kuittausta. Jos kuittausta ei kuulu tietyn ajan sisällä, lähetetään viesti uudelleen. Vasta, kun kuittaus on saatu, lähetetään seuraava viesti.

Stop-and-wait -algoritmin kanssa saman ongelman ratkaisee valtuuden välitys (*token passing*) -algoritmi, jossa *valtuutta* siirretään kahden suorittimen välillä. Valtuuden välityksessä vain toisella suorittimista voi olla valtuus (*token*) yhdellä ajan hetkellä. *Stop-and-wait* -algoritmi saadaan suorittamaan valtuuden välitystä kun sovitaan, että lähettäjällä on valtuus silloin, kun se lähettää uuden viestin. (Uusintalähetyksessä lähettäjällä ei siis ole valtuutta). Vastaanottajalla on valtuus, kun se päättää välittää vastaanotetun viestin eteenpäin. Vastaavasti valtuuden välitys toimii kuin *stop-and-wait* viestinvälitysalgoritmi, kun valtuuden mukana lähettäjältä vastaanottajalle siirretään viestejä.

Koska valtuuden välitys kelpaa linkkikerroksen algoritmiksi, keskitytään seuraavaksi tarkastelemaan vain valtuuden välitystä. Valtuuden välityksessä valtuus saa olla yhdellä ajan hetkellä vain toisella suorittimista, mutta kummankin suorittimen täytyy saada äärettömän pituisessa suorituksessa valtuus äärettömän usein.

2.1 Määritelmiä

Valtuuden välitys -algoritmissa keskitytään kahden suorittimen väliseen kommunikointiin käyttäen viestin välitystä. Tällöin konfiguraatio $c = (s_s, s_r, q_{s,r}, q_{r,s})$, missä s_s on lähettäjän tila, s_r vastaanottajan tila, $q_{s,r}$ lähettäjältä vastaanottajalle matkalla olevien viestin jono, ja $q_{r,s}$ vastaanottajalta lähettäjällä matkalla olevien viestin jono.

Konfiguraatiosta toiseen siirrytään, kun jompi kumpi suorittimista suorittaa yhden laskenta-askeleen a . Laskenta-askeleen aluksi suoritin joko lähettää tai vastaanottaa viestin, jonka jälkeen se suorittaa sisäisiä laskentaoperaatioitansa. Askel päättyy juuri ennen seuraavan viestin lähetystä tai vastaanottoa. Askel voi alkaa myös seurauksena aikakatkaisu mekanismista, jonka tarkoituksena on algoritmin toipuminen tilanteesta, jossa jokin viesti on kadonnut matkalla. Ilman tällaista ajastinta voitaisiin joutua lukkiutuneeseen tilanteeseen, jossa kumpikin suoritin odottaa viestiä toisiltaan.

Suoritus $E = (c_1, a_1, \dots, a_k, c_{k+1})$ on konfiguraatioiden ja laskenta-askeleiden sarja, jossa siirtyminen konfiguraatiosta toiseen on aina tulosta laskenta-askeleesta: $c_{i-1} \xrightarrow{a_{i-1}} c_i$ ($i > 1$).

Itsestabiloituvan algoritmin tavoitteena on päästä mistä tahansa alkutilasta sellaiseen tilaan, eli konfiguraatioon, josta eteenpäin algoritmi suorittaa vain ”laillisia” toimintoja. Tällaista konfiguraatiota kutsutaan turvalliseksi konfiguraatioksi (*safe configuration*).

Vuorottaja–onni -peli (*scheduler–luck game*). Itsestabiloituvan algoritmin tulee päästä turvalliseen konfiguraatioon mistä tahansa aloituskonfiguraatiosta, kunhan vuorottaja antaa kullekin suorittimelle äärettömän pitkän suorituksen aikana vuoron äärettömän monta kertaa. Kun algoritmi on satunnainen, voidaan sen itsestabiloituvuuden todistamisessa käyttää vuorottaja–onni -peliä. Pelissä *vuorottaja* valitsee sellaisen strategian, että turvalliseen konfiguraatioon pääseminen olisi mahdollisimman hankalaa. Toisaalta *onni* voi puuttua peliin aina, kun suoritetaan satunnaisfunktio. Onni voi valita funktion arvon haluamallaan tavalla ja näin pakottaa algoritmin päättymään turvalliseen konfiguraatioon. Jos onni kiinnittää satunnaisfunktion arvot mahdollisten arvojen h osajoukkoon g , niin tällaisen valinnan todennäköisyys $p = g/h$. Jos onni puuttuu peliin useita kertoja, on valintojen yhdistetty todennäköisyys $cp = \prod p_i$. Olkoon odotusarvo askelten määrälle, joiden kuluessa onnen valitsema strategia johtaa stabiloitumiseen, r . Tällöin odotusarvo askelten määrälle,

```

Sender:
01  upon timeout
02      send(counter)
03  upon message arrival
04  begin
05  receive(MsgCounter)
06      if MsgCounter  $\geq$  counter then
07          (* token arrives *)
08              begin (* send new token *)
09                  counter := MsgCounter + 1
10                  send(counter)
11              end
12      else send(counter)
13  end
Receiver:
14  upon message arrival
15  begin
16      receive(MsgCounter)
17      if MsgCounter  $\neq$  counter then
18          (* token arrives *)
19          counter := MsgCounter
20      send(counter)
21  end

```

Kuva 1: Valtuuden välitys käyttäen laskuria ilman ylärajaa. Lähde: [Dol00]

joiden kuluessa satunnaisalgoritmi stabiloituu ilman onnen puuttumista peliin, on r/cp .

2.2 Valtuuden välitys käyttäen laskuria ilman ylärajaa

Kuvassa 1 on algoritmi, joka ratkaisee valtuuden välitys -ongelman käyttäen laskuria. Sekä lähettäjällä että vastaanottajalla on muuttuja *counter*, jonka perusteella ne pystyvät päättämään viestin vastaanoton jälkeen, kummalla valtuus on. Kuhunkin viestiin liittyy valtuusnumero, jota merkitään *MsgCounter*:lla.

Kun lähettäjä saa vastaanottajalta viestin (eli kuittauksen lähettämäänsä viestiin), se vertaan viestin numeroa laskurinsa arvoon. Niin kauan kuin numero ei täsmää, lähettäjä lähettää viimeisimmän viestin uudelleen. Kun numero täsmää, lähettäjä kasvattaa laskurin arvoa yhdellä ja lähettää tällä uudella valtuusnumerolla varustetun viestin vastaanottajalle. Lähettäjällä on valtuus sillä hetkellä, kun se kasvattaa laskurin arvoa. Välittömästi, kun se lähettää tällä uudella numerolla varustettu viestin vastaanottajalle, luopuu se valtuudesta.

Vastaanottaja kuittaa jokaisen saamansa viestin lähettämällä laskurinsa numeron lähettäjälle. Vastaanottaja saa valtuuden joka kerta, kun se vastaanottaa omasta laskuristaan eroavan valtuusnumeron. Tällöin se päivittää laskurinsa arvon vastaanottamaan uutta valtuusnumeroa ja luopuu valtuudesta lähettämällä kuittauksen.

Valtuuden välitys -algoritmin kannalta turvallinen konfiguraatio on sellainen, jossa matkalla jonoissa $q_{s,r}$ ja $q_{r,s}$ olevien viestien numerot ovat yhtä suuria sekä keskenään että lähettäjän ja vastaanottajan laskurimuuttujien arvojen kanssa. Algoritmin toiminnan oikeaksi todistus muistuttaa paljon Dijkstran keskinäisen poissulkemisen algoritmin [Dij74] todistusta. Itse asiassa kummankin algoritmin tehtävä on sama: keskinäisessä poissulkemisessä vain yksi suoritin voi olla kerralla suoritusvuorossa, ja valtuuden välityksessä vain yhdellä suorittimella voi olla kerralla valtuus. Erona on, että valtuuden välityksessä valtuuden mukana voidaan siirtää muuta tietoa. Seuraava lemma osoittaa, että edellä kuvatusta turvallisesta tilasta alkava suoritus on laillinen suoritus valtuuden välitys -algoritmille. Sen jälkeen voidaan todistaa, että algoritmi on itsestabiloituva.

Lemma. *Konfiguraatio c , jossa viestien ja suorittimien laskurien arvot ovat yhtä suuria, on turvallinen konfiguraatio kuvan 1 algoritmille.*

Todistus. Kun lähettäjä vastaanottaa seuraavan viestin konfiguraation c jälkeen, on viestin valtuusnumero välttämättä sama lähettäjän laskurin *counter* kanssa, koska joko tällainen viestin on jonossa odottamassa vastaanottoa, tai sellainen tullaan lähettämään ajastimen lauettua. Nyt lähettäjällä on valtuus. Vastaanotettuaan viestin, lähettäjä luo uuden valtuusnumeron kasvattamalla laskurinsa arvoa, ja lähettää uuden viestin tällä valtuusnumerolla, samalla luopuen valtuudesta. Ennen kuin lähettäjä saa takaisin viestin, jossa on äsken valittu valtuusnumero, täytyy viestin välttämättä tavoittaa vastaanottaja. Siten vastaanottajalla täytyy olla valtuus ennen kuin lähettäjä saa valtuuden uudelleen. Kun lähettäjä vastaanottaa viestin, jossa on lähettäjän laskuria vastaava valtuusnumero, on kaikissa laskureissa sekä matkalla olevissa viesteissä sama valtuusnumero. Siten on saavutettu uusi turvallinen konfiguraatio, johon voidaan soveltaa samaa päättelyä. \square

Lause. *Jokaiselle konfiguraatiolle c pätee, että kaikki reilut suoritukset jotka alkavat c :stä, johtavat turvalliseen konfiguraatioon.*

Todistus. Aluksi osoitetaan, että jokaisessa reilussa suorituksessa lähettäjän laskurin *counter* arvoa kasvatetaan äärettömän usein. Tehdään vastaoletus, että on olemassa konfiguraatio, jonka jälkeen lähettäjän laskuria ei kasvateta. Koska suoritus

on reilu, niin jokainen kulloinkin suoritettavissa oleva laskenta-askel suoritetaan äärettömän usein. Erityisesti, jokainen viesti, joka lähetetään äärettömän usein, myös tulee vastaanotetuksi. Ajastin takaa, että lähettäjä lähettää ainakin yhtä viestiä toistuvasti. Vastaanottaja ennen pitkää vastaanottaa tämän viestin ja lähettää kiittauksen takaisin lähettäjälle. Ennen pitkää lähettäjä saa kiittauksen, jossa on sama valtuusnumero lähettämänsä viestin kanssa, joten se kasvattaa laskurinsa arvoa. Siten vastaoletus on väärä, eli ei ole olemassa konfiguraatiota, jonka jälkeen lähettäjä ei ennen pitkää kasvattaisi laskuriaan.

Olkoon max suurin luku, joka on ensimmäisessä konfiguraatiossa c lähettäjän tai vastaanottajan laskureissa, tai matkalla olevissa viesteissä valtuusnumerona. Ennen pitkää lähettäjä tulee kasvattamaan laskurinsa arvon lukuun $max + 1$, jolloin se tuo järjestelmään uuden valtuusnumeron, jollaista ei löydy yhdestäkään aiemmasta konfiguraatiosta. Kun lähettäjä vastaanottaa kiittauksen, jossa on valtuusnumero $max + 1$, on järjestelmä saavuttanut turvallisen konfiguraation. \square

2.3 Tilavaativuuden alaraja

Edellisessä luvussa esitetty algoritmi ei aseta laskurille mitään ylärajaa. Siten järjestelmän tarvitseman muistin määrä kasvaa suorituksen edetessä. Käytetyn muistin määrä lasketaan katsomalla, kuinka monta bittiä tarvitaan sen hetkisen konfiguraation tallettamiseen. Mukana on siis sekä jonojen sisältämien viestien vaatima tila, että lähettäjän ja vastaanottajan muuttujien varaama tila.

Syynä kasvavaan tilan tarpeeseen on, että lukkiutumisen välttämiseksi tarvitaan ajastin, joka lähettää uusia viestejä, jos vastausta ei kuulu riittävän nopeasti. Algoritmissa oletetaan, että viive ajastimen laukeamiseen on asetettu ainakin niin pitkäksi, että matkalla jonoissa ei pitäisi olla yhtään viestiä. Tällainen oletus ei kuitenkaan ole välttämättä realistinen, joten ajastimella joudutaan lähettämään uusia viestejä jonon jatkoksi, jotta algoritmi stabiloituisi mistä tahansa konfiguraatiosta lähtien.

Algoritmin tilavaativuus kasvaa logaritmisesti suorituksen edetessä. Tämän todistus menee pääpiirteissään siten, että ensin määritellään heikko poissulkemisalgoritmi. Valtuuden välitys on erikoistapaus heikosta poissulkemisesta, joten mikä pätee heikolle poissulkemiselle, pätee myös valtuuden välitykselle. Heikolle poissulkemisalgoritmille voidaan todistaa, että jokainen järjestelmän konfiguraatio on yksilöllinen, tai muuten algoritmi ei ole itsestabiloituva. Koska jokainen konfiguraatio on yksi-

```

Sender:
01  upon timeout
02      send(label)
03  upon message arrival
04  begin
05  receive(MsgLabel)
06      if MsgLabel = label then
07          (* token arrives *)
08              begin (* send new token *)
09                  label := ChooseLabel(MsgLabel)
10                  send(label)
11              end
12      else send(label)
13  end
Receiver:
14  upon message arrival
15  begin
16      receive(MsgLabel)
17      if MsgLabel ≠ label then
18          (* token arrives *)
19              label := MsgLabel
20      send(label)
21  end

```

Kuva 2: Valtuuden välitys käyttäen satunnaista valtuuksien numerointia. Lähde: [Dol00]

öllinen, niin jokaiselle $t > 0$ pätee, että ainakin yksi ensimmäisestä t :stä konfiguraatiosta vie tilaa vähintään $\lceil \log_2(t) \rceil$ bittiä. Tarkempi todistus on Dolevin kirjassa [Dol00].

2.4 Valtuuden välitys, kun laskurilla on yläraja

Tavallisissa sovelluksissa 64-bittinen laskuri riittää lähes mihin tahansa käyttötarkoitukseen. Itsestabiloivissa algoritmeissa pitää kuitenkin ottaa huomioon, että virhetilanteesta johtuen laskuri voi hetkessä hypätä maksimiarvoonsa. Ongelman voi ratkaista, jos viestien määrällä kommunikaatiolinkeissä on olemassa jokin yläraja. Merkitään ylärajaa vakiolla cap . Tällöin kuvan 1 algoritmia voidaan muokata siten, että rivillä 8 laskuria kasvatetaan modulo $cap + 1$. Algoritmista tulee näin itsestabiloituva, koska (yksinkertaistettuna) lähettäjä ennen pitkää valitsee laskurin arvoksi sellaisen luvun, jolla varustettua viestiä ei ole matkalla lähettäjältä vastaanottajalle eikä takaisin.

Jos ylärajaa ei pystytä määrittämään, voidaan käyttää satunnaisalgoritmia. Kuvan 2 algoritmissa laskurin kasvattamisen sijaan uusi valtuuden numero (*label*) määrätään satunnaisesti. Uuden numeron pitää kuitenkin poiketa edellisestä, minkä takia edellinen numero välitetään *ChooseLabel*-funktiolle parametrina.

Satunnaisalgoritmi stabiloituu todennäköisyydellä 1, mistä johtuen on mahdollista olla äärellisen pitkä suoritus E , jossa turvallista konfiguraatiota ei saavuteta. Erilaisia valtuuksien numeroita, joiden välillä arvonta suoritetaan, pitää olla vähintään 3, jotta algoritmi olisi itsestabiloituva. Lähettäjä lähettää numerolla *label* varustettua viestiä niin kauan, kunnes se vastaanottaa samalla numerolla vastaanotetun viestin. Tämän jälkeen se arpoo uuden numeron, joka eroaa edellisestä arvotusta numerosta.

Algoritmin toiminnan oikeaksi osoittaminen tapahtuu pääpiirteittäin seuraavasti: Konfiguraatiossa c merkitään lähettäjältä vastaanottajalle matkalla olevien valtuusnumeroiden jonoa $L_{sr}(c) = l_1, l_2, l_3, \dots, l_k$. Jono $L_{rs}(c) = l_{k+1}, l_{k+2}, l_{k+3}, \dots, l_{k+q}$ on vastaava jono vastaanottajalta lähettäjälle. Edellisten yhdistelmää merkitään $L(c) = l_1, l_2, \dots, l_k, l_{k+1}, l_{k+2}, \dots, l_{k+q}$. Jonossa $L(c)$ on siis peräkkäin lähettäjältä vastaanottajalle menossa olevien ja takaisin päin tulossa olevien viestin valtuusnumerot.

Segmentti $S(c) = l_j, l_{j+1}, \dots, l_{j+n}$ on maksimaalinen peräkkäisten valtuuksien numeroiden jono $L(c)$:ssä siten, että numerot ovat yhtä suuria. Jonossa $S(c)$ voi olla useita segmenttejä, mutta turvallisessa konfiguraatiossa niitä on selvästi aina vain yksi. Merkinnällä *Segmenttejä*($L(c)$) tarkoitetaan segmenttien lukumäärä $L(c)$:ssä.

Pseudo-stabiloitunut konfiguraatio on sellainen konfiguraatio, jossa seuraavan lähettäjältä tulossa olevan viestin valtuusnumero on yhtä suuri kuin muuttujan *label* arvo. Matkalla lähettäjältä vastaanottajalle tai takaisin saattaa siis olla viestejä, joiden numerot eroavat lähettäjän muuttujan *label* arvosta. Niinpä *pseudo-stabiloitunut konfiguraatio* on turvallinen konfiguraatio, jos *Segmenttejä*($L(c)$) = 1, eli jos kaikkien matkalla olevien viestien numerot ovat yhtä suuria sekä keskenään että muuttujien *label* kanssa.

Olkoot c_1 ja c_2 kaksi peräkkäistä pseudo-stabiloitunutta konfiguraatiota. Osoitetaan, että *Segmenttejä*($L(c_1)$) \geq *Segmenttejä*($L(c_2)$). Lisäksi, jos segmenttien lukumäärä c_1 :ssä on suurempi kuin 1, niin todennäköisyydellä $1/2$ segmenttien lukumäärä c_2 :ssa on pienempi kuin c_1 :ssä.

Aloitetaan suorituksen tarkastelu konfiguraatiosta c_1 , jonka jälkeen lähettäjä vastaanottaa viestin m_{k+q} . Koska c_1 on pseudo-stabiloitunut, niin viestin valtuusnumero

on sama kuin lähettäjän muuttujassa *label* oleva numero. Siten viestin vastaanoton seurauksena lähettäjä valitsee uuden valtuusnumeron, joka eroaa viestin m_{k+q} numerosta. Käytetään kolmea eri valtuusnumeroa. Sovitaan, että uusi valtuusnumero on 2 ja edellinen numero oli 0.

Tämän jälkeen yksikään sen segmentin S viesteistä, joihin äsken vastaanotettu viesti kuului, ei aiheuta valtuuden vastaanottoa. Toisin sanoen segmentti S poistuu jonosta, ja jonon L alkuun luodaan uusi segmentti, jonka kaikilla viesteillä on valtuusnumero 2. Jos segmenttiä S seuraa toinen segmentti S' , jonka valtuusnumero eroaa S :n valtuusnumerosta (eli se on 1), niin myös tämä segmentti poistuu. Jälkimmäisessä tapauksessa segmenttien määrä siten vähenee yhdellä.

Osoitetaan vuorottaja-onni -pelin avulla, että algoritmi saavuttaa turvallisen konfiguraation. Onni valitsee aina sellaisen valtuusnumeron, että segmenttien lukumäärää saadaan pienennettyä yhdellä, jos mahdollista. Todennäköisyys tällaiselle valinnalle on $1/2$, joten jos jonossa on $sn = \text{Segmenttejä}(L(c))$ segmenttiä, niin yhdistetty todennäköisyys valinnoille on vähintään $2^{-(2 \cdot sn)}$. Onnen strategia takaa, että segmenttien lukumäärä puolittuu aina sellaisella suorituksella, joka alkaa konfiguraatiosta c ja päättyy sellaiseen konfiguraatioon c' , jota juuri ennen lähettäjä vastaanottaa kuittauksen konfiguraation c jälkeen lähettämäänsä viestiin. Niinpä valintojen lukumäärä, ennen kuin turvallinen konfiguraatio saavutetaan, on $sn + sn/2 + sn/4 + \dots = 2 \cdot sn$.

Koska viestien määrä jonossa ei välttämättä vähene, tarvitaan pelissä kierroksia $O((k+q) \cdot \log(k+q))$ kappaletta, missä $k+q$ on viestien määrä jonossa ensimmäisessä konfiguraatiossa. Koska onnen strategian todennäköisyys on $2^{-(2 \cdot sn)}$, niin turvallinen konfiguraatio saavutetaan enintään $O((k+q) \cdot \log(k+q) \cdot 2^{-(2 \cdot sn)})$ kierroksessa.

Tarvittavien kierrosten määrän odotusarvoa saadaan pienennettyä lisäämällä erilaisten valtuusnumeroiden lukumäärää.

2.5 Jaetun muistin simulointi käyttäen itsestabiloituvaa valtuuden välitystä

Oletetaan, että simuloitavassa algoritmissa käytetään suorittimien P_i ja P_j väliseen kommunikointiin kahta rekisteriä, jotka tukevat atomisia luku- ja kirjoitusoperaatioita. Suoritin P_i kirjoittaa rekisteriin r_{ij} ja lukee rekisteristä r_{ji} . Suoritin P_j käyttää rekistereitä toiseen suuntaan, eli se lukee rekisteristä r_{ij} ja kirjoittaa rekisteriin r_{ji} . Suorittimien välistä kommunikointi pystytään siten simuloimaan käyttäen kahta

yhdensuuntaista linkkiä: yksi P_i :stä P_j :hin ja toinen päinvastaiseen suuntaan.

Simulointi voidaan suorittaa käyttäen itsestabiloituvaa valtuuden välitys -algoritmia. Valtuuden välityksessä tietoa siirretään kaksisuuntaisesti, joten kumpaakin kahden suorittimen välisistä yhdensuuntaisista linkeistä pystytään simuloimaan samalla kertaa. Kutakin kahden suorittimen välistä linkkiä varten tarvitaan oma instanssinsa algoritmista, ja siten myös oma ajastimensa viestin uudelleenlähetystä varten.

Valtuuden välityksessä täytyy päättää, kumpi suorittimista on lähettäjä ja kumpi vastaanottaja. Oletetaan, että suorittimilla on yksilölliset numerot. Suoritin voi liittää jokaiseen lähettämäänsä viestiin oman numeronsa, jolloin naapurit oppivat toistensa numerot ennen pitkää. Voidaan sopia, että suuremmalla numerolla varustettu suoritin toimii valtuuden välityksessä lähettäjänä.

Jaetun muistin simulointia varten jokaisella suorittimella P_i on paikallinen muuttuja R_{ij} , joka sisältää simuloitavaan jaettuun rekisteriin r_{ij} viimeksi kirjoitetun arvon. Suorittimella P_j on siis vastaavasti paikallinen muuttuja R_{ji} .

Valtuuden välityksessä suorittimet P_i ja P_j siirtävät vuorotellen valtuuden toisilleen. Jaetun muistin simulointi onnistuu, kun valtuuden mukana siirretään myös simuloitavien rekistereiden viimeisimmät arvot. Suoritin P_i liittää siis valtuuden mukaan paikallisen muuttujansa R_{ij} arvon, ja suoritin P_j vastaavasti muuttujansa R_{ji} arvon. Rekisteriin r_{ij} kirjoittamisen simulointi tapahtuu yksinkertaisesti siten, että suoritin P_i kirjoittaa halutun arvon paikalliseen muuttujaansa R_{ij} . Kun suoritin P_j haluaa lukea rekisterin r_{ij} arvon, toimii se seuraavasti:

1. P_j vastaanottaa valtuuden P_i :ltä
2. P_j vastaanottaa valtuuden toiseen kertaan P_i :ltä. Rekisterin r_{ij} arvo on tämän jälkimmäisen valtuuden yhteydessä vastaanotettu R_{ij} :n arvo.

Suoritin P_j ei siis jatka simuloitavan ohjelmansa suorittamista valtuuksien vastaanottamisen välillä. Sen sijaan se jatkaa viestien vaihtamista naapurisuorittimiensa kanssa, jotta ne pystyvät lukemaan P_j :n rekistereihinsä kirjoittamia arvoja. Kaksisivaiheisen lukuoperaation tarkoituksena on taata, että luku- ja kirjoitusoperaatiot voidaan aina järjestää aikajärjestykseen.

Algoritmin oikeaksi todistamista varten osoitetaan, että jokaisella suorituksella E on loppuosa, jossa on mahdollista *linearisoida* kaikki simuloitavat luku- ja kirjoitusoperaatiot. Linearisointi on mahdollista, jos on olemassa sellainen täydellinen järjestyks suoritetuille luku- ja kirjoitusoperaatioille, että jokaisen suorittimen tekemien

luku- ja kirjoitusoperaatioiden järjestys säilyy, ja jokainen rekisteristä r luettu arvo on siihen (valitun täydellisen järjestyksen mukaan) viimeksi kirjoitettu arvo. Jos viimeksi kirjoitettua arvoa ei ole, eli lukeminen suoritetaan ennen ensimmäistä kirjoitusta, käytetään arvona vakiota x . Linearisoituvuudesta seuraa, että algoritmi simuloi jaettujen rekistereiden toimintaa oikein.

Valitaan jokaiselle luku- ja kirjoitusoperaatiolle omat ajanhetkensä. Suorittimen P_i suorittaman simuloitun lukuoperaation suoritus alkaa jossain tietyssä konfiguraatiossa c_r ja päättyy myöhemmin konfiguraatiossa c_{r+k} , jollain k . Suoritin ei jatka simuloitavan ohjelmansa suorittamista ennen kuin c_{r+k} on saavutettu. Erityisesti se ei muuta yhdenkään paikallisen muuttujansa R_{im} sisältöä c_r :n ja c_{r+k} :n välillä. Lisäksi, koska valtuus vastaanotetaan kahdesti lukuoperaation aikana, on luettu arvo varmasti jokin suorittimen P_j muuttujaansa R_{ji} kirjoittamista arvoista ajan hetkien r ja $r + k$ välillä. Siten lukuoperaation ajan hetken indeksiksi voidaan valita mikä tahansa luku väliltä r ja $r + k$.

Valitaan rekisterin r_{ij} simuloitun kirjoitusoperaation ajanhetken indeksiksi sen askeleen indeksi, jolloin suoritin P_i kirjoittaa muuttujaansa R_{ij} uuden arvon. Valitaan suorittimen P_j suorittaman simuloitun lukuoperaation ajanhetken indeksiksi sen askeleen indeksi, jolloin suoritin P_i lähettää toisen kerran (valtuuden mukana) muuttujan R_{ij} arvon P_j :lle.

Nyt jokaisella rekisteriin r_{ij} kohdistuvalla operaatiolla on sellainen ajanhetken indeksi, että rekisteristä luettu arvo on viimeisin siihen kirjoitetuista arvoista, kunhan alla olevan valtuuden välitys -algoritmin toiminta on stabiloitunut.

Lähteet

- Dij74 Dijkstra, E. W., Self-stabilizing systems in spite of distributed control. *Commun. ACM*, 17,11(1974), sivut 643–644.
- Dol00 Dolev, S., *Self-stabilization*. MIT Press, Cambridge, MA, USA, 2000.