

Stabiloivat synkronoijat ja nimeäminen

Mikko Ajoiviita

Helsinki 28.10.2007

Hajautettujen algoritmien seminaari

HELSINGIN YLIOPISTO

Tietojenkäsittelytieteen laitos

Sisältö

1 Johdanto	1
2 Stabiloivat synkronoijat	2
2.1 Rajoittamaton α -synkronoija	2
2.2 Rajoitettu α -synkronoija	4
2.3 β -synkronoija	6
3 Nimeämistekniikka	9
Lähteet	11

1 Johdanto

Synkronoimattoman järjestelmän suunnittelemisen ja analysointi voi olla hyvinkin monimutkaista. Asynkronisessa järjestelmässä jokainen alkutila voidaan tulkita joukoksi suorituksia, jotka eroavat järjestykseltään toisistaan. Suorituksen etenemiselle voi olla monia erilaisia vaihtoehtoja, koska prosessorit suorittavat toimiaan eri tahdissa. Näin ollen kaikkien erilaisten alkutilojen analysointi voi olla työlästä.

Synkronisessa järjestelmässä sitä vastoin jokainen prosessi vaihtaa tilaa samanaikaisesti. Näin alkutila määrittelee yksiselitteisesti suoritukset, jotka ovat mahdollisia siitä eteenpäin. Siksi onkin järkevää lähestyä asynkronisia ongelmia käyttäen apuna synkronisia tuloksia. Tämä tapahtuu kehittämällä yleinen simulointitekniikka, jonka avulla synkroniset ongelmat voidaan laajentaa suoritettavaksi synkronoimattomissa järjestelmissä. Tästä simulointitekniikasta käytetään nimeä *synkronoija*.

Synkronoijan perusidea on, että asynkroniselle ongelmalle etsitään synkroninen ratkaisu. Synkronoijaa avuksi käyttäen synkroninen algoritmi voidaan suorittaa asynkronisessa järjestelmässä. Aina ei ole kuitenkaan järkevää käyttää synkronoijaa, sillä tehokkuus kärsii samalla. Vuorottelemalla tasaisesti suorituksia nopeimmat prosessit kärsivät ja niiden toiminta rajoittuu hitaampien prosessien tasolle. Vastaavasti nopeammat prosessit eivät kykene auttamaan hitaampia prosesseja suorituksessa.

Kirjoitelman lopussa tutustutaan pintapuolisesti *nimeämistekniikkaan* [Dol00]. Verkon solmut voidaan erottaa toisistaan yksilöivillä tunnuksilla, jotka kaikki eroavat toisistaan. Tietyt sovellukset vaativat nimenomaan, että kaikilla prosessoreilla verkossa on yksilöivät tunnisteet. Voidaanko tällaisia sovelluksia käyttää myös anonyymeissa verkoissa, joissa solmujen tunnisteita ei tunneta?

Nimeämistekniikka tarjoaa tähän ongelmaan erään ratkaisun. Tekniikka nimeää satunnaisesti anonyymin verkon solmut ja selvittää sen jälkeen, onko samoja tunnuksia olemassa. Tarkoituksena on lopulta päästä vastaavaan tilanteeseen kuin systeemissä, jolla on yksilöivät tunnisteet prosessoreillaan. Nimeämistekniikka pohjautuu pääosin synkronoijan käyttöön sekä päivitysalgoritmiin.

2 Stabiloivat synkronoijat

Alunperin synkronoija käsitteen esitteli Baruch Awerbuch vuonna 1985 [Awe85]. Hän kehitti kolme erilaista synkronoija-algoritmia, joita kutsutaan *alpha*-, *beta*- ja *gamma*-synkronoijiksi. Tässä yhteydessä tarkastelemme kahta ensin mainittua algoritmia. γ -synkronoija on niiden eräänlainen yhdistelmä, mutta se ei ole itsestään stabiloiva algoritmi.

2.1 Rajoittamaton α -synkronoija

Itsestabiloituva α -synkronoija soveltuu sanomanvälitys järjestelmään. Se käyttää apunaan itsestabiloivaa datayhteys -algoritmia (engl. self-stabilizing data-link algorithm). Tämä takaa sen, että päästessään stabiloituun tilaa, kaikki prosessorin lähettämät viestit ovat saapuneet perille.

α -synkronoija edellyttää jokaisen prosessorin ottavan huomioon kaikki naapurinsa kertaalleen yhden suorituskierroksen aikana. Prosessori odottaa niin kauan, kunnes se on saanut viestin kaikilta naapureiltaan koskien edeltävää kierrosta. Vasta tämän jälkeen se päivittää tilansa tarvittaessa ja lähettää uuden viestin naapureilleen.

Synkronisessa järjestelmässä prosessorin tila ajan hetkellä $t + 1$ riippuu sen naapureiden tilasta ajan hetkellä t . Kun käytetään α -synkronoijaa, jokainen prosessori laskee tilansa vastaavista naapuriprosessoreiden tiloista. Tällöin käytetään eräänlaista jäljiteltyä systeemiä, jolla simuloidaan synkronista toimintaa asynkronisessa järjestelmässä.

Yksinkertaisuuden vuoksi voidaan aluksi määritellä *rajoittamaton* tilamuuttuja. Tällöin jokaisella prosessorilla on mielivaltainen arvo *phase*-muuttujassaan. Jokaisella suorituskierroksella prosessori lähettää tiedon omasta tilastaan muille (kuva 1 rivi 4) ja saa tiedot naapureiden tilamuuttujista (kuva 1 rivit 5 - 10). Tämän jälkeen prosessori vertaa niitä omaan tilaansa (kuva 1 rivi 11). Jos kaikkien naapureiden tilat ovat suurempia tai yhtä suuria kuin prosessorin itsensä tila, se kasvattaa tilamuuttujansa arvoa yhdellä (kuva 1 rivi 12). Tilamuuttujan arvo ei voida koskaan pienentyä.

Kun näin edetään ja annetaan suoritusaikaa kaikille prosessoreille rajattomasti, päädytään lopulta *turvalliseen* tilaan koko asynkronisessa järjestelmässä.

```

1.  do forever
2.    forall  $P_j \in N(i)$  do receivedj := false
3.    do
4.      DLsend(phasei) (*start send to all neighbors*)
5.      upon DLreceive(ackj, phasej)
6.        receivedj := true
7.        phaseji := phasej
8.      upon DLreceive(phasej)
9.        phaseji := phasej
10.     until  $\forall P_j \in N(i)$  receivedj := true (*all send terminated*)
11.     if  $\forall P_j \in N(i)$  phasei ≤ phaseji then
12.       phasei := phasei + 1
13.   od

```

Kuva 1: α -synkronoija (rajoittamaton versio)

Lemma 1.1 *Jokaisen kelvollisen suorituksen jälkeen päädytään sellaiseen alkutilaan, jossa kaikkien naapuri prosessorien tilat eroavat toisistaan korkeintaan yhdellä.*

Oletetaan, että kaikissa alkutiloissa ensimmäisen prosessorin tila $phase_i$ on vähintään kahta suurempi kuin sen naapuriprosessorin tila $phase_j$. Koska $phase_j$ ei voi koskaan pienentyä, päädytään jompaan kumpaan seuraavista vaihtoehdoista. (1) joko $phase_i$ korotetaan äärettömän monta kertaa, ja se on kaikissa alkutiloissa edelleen vähintään kahta suurempi kuin $phase_j$ tai (2) kumpikaan prosessori ei kasvata tilamuuttujansa arvoa.

(1) vaihtoehto ei pidä paikkaansa. Tilamuuttujan arvo ei voi koskaan laskea. Tilamuuttujan $phase_i$ arvo ei voi myöskään kasvaa, koska $phase_i$ kasvaa vain, jos epäyhtälö $phase_i \leq phase_j$ (kuva 1 rivit 11-12) on totta. Kun prosessorille P_j annetaan aikanaan suoritusvuoro, se kasvattaa tilamuuttujansa arvoa saavuttaen prosessorin P_i tilamuuttujan arvon. Näin ensimmäinen kohta on mahdoton.

(2) tapauksessa $phase_i$ arvoa ei enää koroteta jostain alkutilanteesta eteenpäin. Tässä tapauksessa prosessorilla $phase_i$ täytyy olla jokin naapuri, jolla on pienempi arvo tilamuuttujassaan. Naapurikaan ei kasvata arvoaan, jos sen toisella naapurilla on pienempi arvo tilamuuttujassaan. Sama tilanne toistuu seuraavan prosessorin kohdalla, kunnes lopulta löytyy naapuriprosessori, jolla ei enää ole pienempää tilaa. Näin ollen $phase_j$ arvo kasvaa aikanaan eikä tilamuuttujan arvo voi pysyä muuttumattomana eli seuraa ristiriita väitteen kanssa. ■

Lemma 1.2 *Jokaisen kelvollisen suorituksen aikana tilamuuttujan arvoa kasvatetaan äärettömän monta kertaa.*

Nyt voidaan olettaa, että päädytään johonkin alkutilaan, jolloin mikään prosessori ei enää kasvata tilamuuttujan arvoaan. Tällöin valitaan prosessori P_j , jolla on pienin tilamuuttujan arvo. Niin sanotun reiluuden nimissä suoritusvuoro tarjotaan myös prosessorille P_j ja se suorittaa kuvan 1 rivit 2-12 äärettömän monta kertaa. Tällöin P_j saa naapureidensa tilojen muuttumattomat vakioarvot. Koska epäyhtälö on totta, P_j kasvattaa omaa arvoaan yhdellä. ■

2.2 Rajoitettu α -synkronoija

Ongelmankuvaus on samanlainen kuin rajoittamattomassa α -synkronoijassa, mutta nyt käytössä on *rajoitettu* tilamuuttuja. Tällöin puhutaan α -synkronoijan rajoitetusta versiosta. Tilamuuttujassa arvona on modulo M , missä $M \geq N$. Muuttujalla N tarkoitetaan järjestelmässä olevien prosessorien lukumäärää. Lisäksi kaikilla prosessoreilla on ylimääräinen *reset*-muuttuja verrattuna rajoittamattomaan versioon.

Perusideana verrataan edelleen naapureiden tilamuuttujia toisiinsa, jotta systeemi saadaan synkronoitua. Kun naapurin tilamuuttuja eroaa yli yhdellä prosessorin omasta tilamuuttujasta, prosessori nolaa oman tilamuuttujansa. Näin saavutetaan tila, jossa kaikkien prosessorien tila lopulta eroaa toisistaan enintään yhdellä. Tämän tilan saavuttamisen jälkeen toiminta jatkuu vastaavasti kuin rajoittamattomassakin versiossa.

Käytännössä nollaamisella tarkoitetaan sitä, että prosessorin huomattessa naapurinsa tilamuuttujan arvon eroavan yli yhdellä, prosessori asettaa nollausmuuttujalle arvon 0 (kuva 2 rivit 26 -27). Samalla prosessorin tilamuuttuja nolataan (kuva 2 rivit 29-30). Resetoinnin johdosta prosessorin naapureilla tilamuuttujan arvo ei voi olla yli yhden tai muuten sekään nolataan. Naapurin seuraavalla naapurilla taas arvo ei voi olla yli kahden ja niin edelleen.

Ensimmäinen prosessori, joka asettaa nollausarvonsa nolaksi, levittää eräänlaisen nollausaalton läpi järjestelmän. Nollausaalto asettaa lopulta kaikkien prosessorien tilamuuttujat nolliksi. Laskentaa jatketaan vasta silloin, kun kaikki tilamuuttujat ovat nolattu. Kun nollausaalto etenee, mikään prosessori, joka on jo nolannut kertaalleen nollausmuuttujansa, ei voi kasvattaa nollausmuuttujaansa yli arvon $2N$. Tämän lisäksi tilamuuttujan arvo nolataan tarvittaessa, jos nollausmuuttujan arvo on alle $2N$.

```

1.  do forever
2.    forall  $P_j \in N(i)$  do receivedj := false
3.    PhaseUpdated := false
4.    ResetUpdated := false
5.    do
6.      DLsend(phasei, resetj)(*start send to all neighbors*)
7.      upon DLreceive(ackj, phasej, resetj )
8.        receivedj := true
9.        UpdatePhase()
10.     upon DLreceive(phasej, resetj)
11.       UpdatePhase()
12.     until  $\forall P_j \in N(i)$  receivedj := true (*all send terminated*)
13.     if ResetUpdate = false then
14.       if  $\forall P_j \in N(i)$  reseti ≤ resetji then
15.         reseti := min(2N, reseti+1)
16.       if PhaseUpdated = false and
17.          $\forall P_j \in N(i)$  phasei ∈ {phaseji, (phaseji-1) mod M} then
18.         phasei := (phasei + 1) mod M
19.     od

20. UpdatePhase()
21.   phaseji := phasej
22.   resetji := resetj
23.   if reseti > resetji then
24.     reseti := min(2N, resetji+1)
25.     ResetUpdated := true
26.   if phaseji ∉ {(phasei-1) mod M, phasei, (phasei+1) mod M} then
27.     reseti := 0
28.     ResetUpdated := true
29.   if reseti ≠ 2N then
30.     phasei := 0
31.     PhaseUpdated := true

```

Kuva 2: α -synkronoija (rajoitettu versio)

Lemma 2.1 *Jokainen kelvollinen suoritus päättyy turvalliseen tilaan, jos mikään prosessori ei aseta nollausmuuttujaansa nollassi.*

Tässä tapauksessa kaikkien naapurusten tilamuuttuja eroavat toisistaan korkeintaan yhdellä kaikissa suorituksen vaiheissa tai muussa tapauksessa jokin nollausmuuttuja nolldataan. Tämä ehto näkyy kuvassa 2 rivillä 26. Suorituksen edetessä nollausmuuttujat saavat lopulta kaikki arvon $2N$. Tällöin järjestelmä on turvallisessa tilassa. Vastaavasti, jos kaikkien nollausmuuttujien arvo on alunperin $2N$, niin kaikkien naapurin tilamuuttujat saavat erota korkeintaan yhdellä toisistaan. Näin järjestelmä saavuttaa silloinkin turvallisen tilan. ■

Lemma 2.2 *Jokainen kelvollinen suoritus päättyy turvalliseen tilaan, jos jokin prosessori asettaa nollausmuuttujansa nolllaksi.*

Nyt jokin prosessori P_i asettaa nollausmuuttujansa arvoksi nollan. Seurauksena myös sen tilamuuttuja päivitetään nolllaksi. Seuraavaksi kaikki naapuriprosessorit P_j asettavat nollausmuuttujansa nollliksi tai muuten niiden nollausmuuttujat täytyvät olla alunperin pienempiä kuin yksi (kuva 2 rivit 10,11 ja 23). Täten myös naapureiden tilamuuttujat nolllataan (kuva 2 rivit 28-29). Näin edetään ja nollausaalto leviää koko verkkoon. Lopulta päädytään tilanteeseen, jossa kaikkien prosessorien nollausmuuttujat ovat pienempiä kuin $2N$ ja tilamuuttujat ovat nolllia. Eli on saavutettu turvallinen tila. Mikään prosessori ei aseta tämän jälkeen nollausmuuttujaansa nolllaksi. Edellisen lemman mukaan kelvollinen suoritus päättyy tässä tilanteessa turvalliseen tilaan. ■

2.3 β -synkronoija

Toinen klassinen synkronoija on β -synkronoija. Se toimii järjestelmissä, joissa käytetään apuna jaettua muistia prosessien välillä. β -synkronoija tarvitsee apunaan viritettävää puuta, jossa on erikseen määritelty juuri. Itsestabiloituvalla johtajan valinta-algoritmillä saadaan selvitettyä puurakenteelle juuri, kun prosesseilla on yksilölliset tunnisteet. Ja toisaalta itsestabiloivan viritettävän puun -algoritmillä saadaan luotua puurakenne.

Näin suoritettulla alustuksella viritettävässä puussa on prosessoreita, joista *juuri*-prosessori on erikoisasemassa. Jokaisella prosessorilla on oma *color*-muuttuja. Juuren tarkoituksena on värjätä koko puu toistuvasti. Juuri tarkastelee viritettävän puun osapuita, joiden juurina ovat sen omat lapset. Jos lapset antavat tiedon, että alipuut on värjätty saman värisiksi kuin itse juuri, silloin tämä valitsee itselleen uuden värin. Tämän jälkeen juuri välittää uuden väritiedon lapsilleen. Nämä juuren suorittamat toimenpiteet näkyvät kuvassa 3 riveillä 2-5.

Muut prosessorit lukevat vanhempansa tietoa toistuvasti ja asettavat vanhemman värin omaksi värikseen (kuva 3 rivit 8-10). Lisäksi prosessorit välittävät vanhemman värin eteenpäin omille lapsilleen (kuva 3 rivi 13). Toisin sanoen juuren väri leviää prosessoreille sen alipuissa ja näin koko viritettävä puu saadaan värjättyä samalla värillä.

Root:

1. **do** forever
2. **forall** $P_j \in \text{children}(i)$ **do** $lr_{ji} := \text{read}(r_{ji})$
3. **if** $\forall P_j \in \text{children}(i)$ ($lr_{ji}.\text{color} = \text{color}_i$) **then**
4. $\text{color}_i := (\text{color}_i + 1) \bmod (5n - 3)$
5. **forall** $P_j \in \text{children}(i)$ **do write** $r_{ij}.\text{color} := \text{color}_i$
6. **od**

Other:

7. **do** forever
8. **forall** $P_j \in \{\text{children}(i) \cup \text{parent}\}$ **do** $lr_{ji} := \text{read}(r_{ji})$
9. **if** $\text{color}_i \neq lr_{\text{parent},i}.\text{color}$ **then**
10. $\text{color}_i := lr_{\text{parent},i}.\text{color}$
11. **else if** $\forall P_j \in \text{children}(i)$ ($lr_{ji}.\text{color} = \text{color}_i$) **then**
12. **write** $r_{i,\text{parent}}.\text{color} := \text{color}_i$
13. **forall** $P_j \in \text{children}(i)$ **do write** $r_{ij}.\text{color} := \text{color}_i$
14. **od**

Kuva 3: β -synkronoija

Myös tietoa värjäyksen valmistumisesta välitetään vastaavasti. Tämä tieto kulkee lehdistä kohti juurta. Lehden vanhempi saa tiedon värjäyksen valmistumisesta, kun lehti on värjäynyt itsensä samalla värillä kuin juuri. Lehden vanhempi välittää saman tiedon aikanaan omalle vanhemmalleen jne. Lopulta tieto kulkeutuu aina juureen asti. Näin ollen muutkin prosessorit jäävät odottamaan tietoa lapsiltaan, onko niiden alipuut värjätty (kuva 3 rivi 11). Kun muut prosessorit saavat viimeisenkin kuittauksen kaikkien alipuidensa värjäyksestä, ne välittävät värjäystiedon vanhemmalleen (kuva 3 rivi 12).

Karkeasti ottaen juuresta lehtiin leviää väri koko virittävään puuhun ja samalla lehdistä juureen välittyy kuittaus, että puu on värjätty. β -synkronoijan edellytyksenä on, että juuri vaihtaa väriään äärettömän monta kertaa. Viestien vastaanoton aikana prosessorit lukevat naapureidensa rekistereitä alkuperäisestä verkosta eivätkä pelkästään viritetystä puusta. Nämä ylimääräiset viestit kuitenkin jäävät synkronoijan suorituksen ulkopuolelle, eivätkä siten vaikuta synkronointiin.

Lemma 3.1. *Jokaisessa kelvollisessa suorituksessa juuri vaihtaa väriään vähintään kerran jokaisen $2d + 1$ syklin aikana.*

Oletetaan, että jollain alkutilalla juuri P ei vaihda väriään ollenkaan $2d + 1$ syklin aikana. Tällöin jokaisen juuren lapsen on kopioitava juuren väri ensimmäisen syklin aikana. Samoin lapset kopioita värin lapsilleen toisen syklin aikana. Näin jatketaan ja kaikkien prosessorien väri on lopulta sama d syklin jälkeen.

Täten $d + 1$ syklin aikana jokainen lehti puussa välittää tiedon vanhemmalleen P_k siitä, että värjääminen lehden alipuussa (lehden alipuu on tyhjä) on lopetettu. Syklin $d + 2$ aikana jokainen prosessori P_k raportoi vanhemmalleen, että sen alipuu on värjätty juuren väriseksi. Näin jatketaan ja viimein jokainen juuren lapsi välittää tiedon, että sen alipuu on värjätty juuren väriseksi. Tähän kuuluu aikaa $2d$ sykliä. Näin ollen juuren on vaihdettava väriään viimeistään $2d + 1$ syklin jälkeen. ■

Lemma 3.2 *Alkutilaan, jossa kaikkien prosessorien väri on sama, päästään $O(dn)$ syklin aikana.*

Koko verkossa on n kappaletta värimuuttujia, yhteensä $2n - 2$ värikenttää viestirekistereissä ja $2n - 2$ sisäistä muuttujaa, jotka tallentavat värimuuttujan arvot rekisteristä. Täten, missä alkutilassa tahansa, värien kokonaismäärä voi olla korkeintaan $5n - 4$. Erilaisten värien määrä valitaan siten, että värimuuttuja voi saada arvot väliltä $0 - (5n - 2)$. Siten jäljelle jää vähintään yksi väri, jota ei ilmene verkossa.

Edellisen lemmän mukaan juuri vaihtaa väriään vähintään kerran jokaisen $2d + 1$ syklin aikana. Juuri vaihtaa väriään kasvattamalla sitä yhdellä(mod $5n - 3$). Nyt jokaisen $(2d + 1)(5n - 3)$ syklin aikana juuri vaihtaa värinsä sellaiseen väriin, jota ei esiinny verkossa. ■

Beta-synkronoijan kuvaukseen vaaditaan lisäksi tieto siitä, miten synkronointiaskele on toteutettu. Synkronointiaskele koostuu naapureiden viestirekisterien luvuista. Kun jokainen prosessori on lopettanut lukemisen, prosessori vaihtaa tilaa ja kirjoittaa uuden arvon viestirekisteriin sen mukaan, mitä on aikaisemmin lukenut.

Toisin sanoen, ennen kuin prosessori P_i vaihtaa väriään, se lukee viestirekisterin arvot naapureiltaan, ensimmäisen synkronointiaskeleen tavoin. Ja ennen kuin ei-juuri prosessori välittää tiedon alipuun värjäytymisestä, se kirjoittaa uuden arvon viestirekisteriin, kuin toisena synkronointiaskeleena. Samoin juuri kirjoittaa uuden arvon viestirekisteriin, kun sen lapset ovat ilmoittaneet alipuiden värjäyksen olevan valmis.

3 Nimeämistekniikka

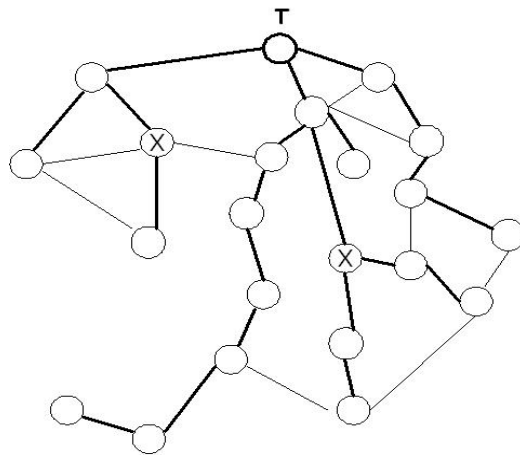
Itsestabiloivan nimeämisen tarkoituksena on saavuttaa tilanne, jossa anonyymeille järjestelmille voidaan suorittaa vastaavia toimintoja kuin systeemeille, joissa on yksilöivät tunnisteet. Tunnisteettoman järjestelmän symmetrian rikkomiseen tarvitaan apuna satunnaistamista. Ilman sitä ei esimerkiksi johtajan valinta onnistu anonyymissä verkossa.

Nimeämistekniikka pohjautuu olennaisesti itsestabiloivaan *päivitys*-algoritmiin sekä edellä esiteltyyn itsestabiloivaan β -synkronoijaan. Päivitysalgoritmi on suunniteltu stabilointiin järjestelmissä, joissa on yksilöivät tunnisteet. Lisäksi uusien satunnaisien tunnisteiden valinnassa käytetään apuna ns. Scheduler-Luck peliä. Tämä takaa sen, että hyvin suurella todennäköisyydellä valittu tunniste ei ole jo olemassa verkossa.

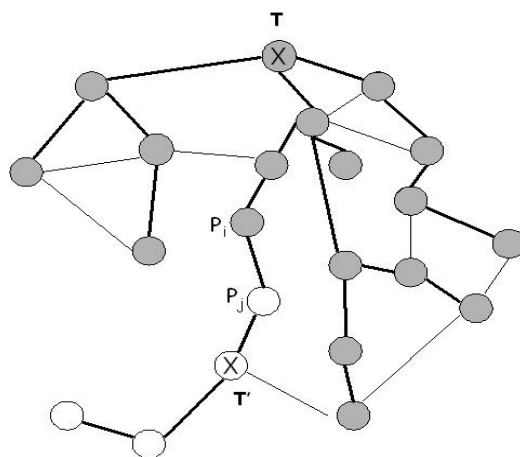
Aluksi verkolle valitaan satunnainen alkutila. Virittävää puuta rakennetaan samalla, kun tietoa lähetetään eteenpäin verkossa juuren tunnisteesta aina lehtiin asti. Vastaavasti tieto puun valmistumisesta leviää vastakkaiseen suuntaan. Lisäksi jokainen prosessori välittää tiedon alipuunsa prosessorien tunnisteista. Lopulta juuri saa listan kaikkien prosessoreiden tunnisteista, jotka ovat sen virittämässä puussa. Listalla voi olla samoja tunnisteita, kuten kuvassa 4 ilmenee. Tällöin juuri välittää tiedon alipuihin tällaisista dublikaateista. Prosessori, jolla on tällainen tuplatunniste, vaihtaa tunnisteensa uuteen sattumanvaraisesti valittuun tunnisteeseen.

Edellinen toimenpide toistetaan yhä uudestaan, kunnes juuri saa listan prosessorien tunnisteista, mitkä eroavat kaikki toisistaan. Tämä ei kuitenkaan vielä takaa, että koko verkossa ei olisi samoja tunnisteita. On mahdollista, että virittävää puuta etsittäessä löytyy useampi puu. Tällöin puiden juurilla on samat tunnisteet. Tällainen tilanne näkyy kuvassa 5.

Oletaan, että päivitystekniikan avulla on muodostunut kaksi erillistä puuta T_1 ja T_2 , joilla on sama tunniste juurenaan (kuva 5). Täten on olemassa kaksi naapuriprosessoria P_i ja P_j , jotka kuuluvat eri puihin. Puut erotetaan toisistaan β -synkronoijasta tutulla värjäysmenetelmällä. Toistuva värjäys takaa sen, että P_i ja P_j tunnistavat kuuluvansa eri puihin, vaikka niiden juurilla onkin sama tunniste. Nyt prosessorit P_i ja P_j välittävät tiedon oman puunsa juurelle tällaisesta tapauksesta. Juuri prosessori saa tiedon toisen puun olemassa olosta, jolla on sama tunniste kuin sillä itsellään. Tällöin juuri vaihtaa omaa tunnistettaan.



Kuva 4: Verkosta on luotu virittävä puu T , jota esittää paksummat viivat. Puu sisältää kuitenkin kaksi samaa tunnistetta X solmuissaan. Tämä tilanne hoidetaan valitsemalla niille uudet satunnaiset tunnisteen.



Kuva 5: Verkosta on muodostunut kaksi erillistä virittävää puuta T ja T' , joiden juurien tunnisteenä on sama arvo X . Nämä puut erotetaan toisistaan värjäämällä kumpikin puu eri värillä. Tämän jälkeen toisen puun juuren tunniste vaihdetaan uuteen satunnaiseen tunnisteseen.

Kun juuri joutuu vaihtamaan omaa tunnistettaan prosessi aloitetaan alusta. Käytännössä puiden rakentaminen ja värjäys etenevät yhtäaikaan. Ensisijaisesti siis tutkitaan, onko useita virittäviä puita olemassa värjäystekniikalla. Tämä tarkoittaa, että verkosta löytyy useampi puun juuri, jolla on sama tunniste. Tällöin ainakin yksi juurista vaihtaa tunnistetta ja puiden rakennus ja värjäys aloitetaan alusta. Näin edetään ja saavutetaan tila, jossa vain yksi puu on olemassa. Tähän prosessiin kuluu aikaa $O(d)$ syklin verran, missä d on verkon halkaisija. Jos alunperinkään ei ollut kuin yksi puu, niin ollaan vastaavassa tilanteessa.

Toissijaisena tutkinnan kohteena on, onko puussa samoja arvoja. Jos näin on pääsyt tapahtumaan, ollaan kuvan 4 tapauksessa. Nyt tunnisteille, joita esiintyy useammin kuin kerran, annetaan uudet satunnaiset arvot. Tähän toimenpiteeseen kuluu samoin $O(d)$ syklin verran aikaa.

Värien kokonaismäärä on määritelty β -synkronoijan yhteydessä. Sen sijaan erilaisien tunnisteiden määrä voi olla luokkaa N^{10} . Yhdessä scheduler-luck pelin kanssa värien määrä takaa sen, että saman värin satunnainen valinta on hyvin epätodennäköistä. Yhtenä ehtona on, että mitään tunnistetta ei vaihdeta kahdesti. Eli jos uusi satunnainen luku on jo verkossa, sitä ei yritetä enää vaihtaa vaan koko prosessi aloitetaan tarvittaessa alusta. Turvallinen tilan saavuttamisen odotusarvo on $O(d)$ sykliä. Näin toteutetulla tekniikalla saavutetaan verkko, jossa kaikilla prosessoreilla on satunnaiset yksilöivät tunnisteet.

Lähteet

- Awe85 Awerbuch, B., Complexity of network synchronization. *Journal of ACM*, 32,4(1985), sivut 804–823.
- Dol00 Dolev, S., *Self-Stabilization*. Mit Press, 2000.