Kaatumisilmaisimet hajautetuissa järjestelmissä

Markku Hanhiniemi

Tampereen yliopisto
Informaatiotieteiden yksikkö
Tietojenkäsittelytieteiden tutkinto-ohjelma
Pro gradu -tutkielma
Ohjaaja: Erkki Mäkinen
Elokuu 2016

Tampereen yliopisto
Informaatiotieteiden yksikkö
Tietojenkäsittelytieteiden tutkinto-ohjelma
Markku Hanhiniemi: Kaatumisilmaisimet hajautetuissa järjestelmissä
Pro gradu -tutkielma, 40 sivua
Elokuu 2016

Tiivistelmä

Pilvipalvelut ovat jo jonkin aikaa olleet keskeinen kaupallisesti hyödynnettävä palvelumuoto internetissä. Tästä ovat tuttuina esimerkkeinä vaikkapa internetin lukuisat hakukoneet sekä erilaiset sähköpostitiliin liitetyt sähköisen materiaalin säilytyspalvelut, kuten esimerkiksi Google Drive. Pilvipalveluiden taustalla olevana arkkitehtuurina ovat hajautetut järjestelmät. Voidaankin sanoa, että pilvipalvelut syntyivät, kun hajautettuja järjestelmiä ryhdyttiin hyödyntämään kaupallisesti.

Tässä tutkielmassa perehdytään hajautettujen järjestelmien solmujen kaatumisen kontrollointiin eli kaatumisilmaisimiin hajautetuissa järjestelmissä. Lisäksi perehdytään kahteen keskeiseen kaatumisilmaisimiin liittyvään asiaan. Nämä ovat hajautetun järjestelmän rakenne ja viestintä hajautetussa järjestelmässä.

Avainsanat: hajautettu järjestelmä, kaatumisilmaisin, kaatumisen havaitseminen, viestintä

Sisällys

1. Johdanto

Perusmääritelmän mukaan hajautettu järjestelmä koostuu joukosta itsenäisiä tietokoneita, jotka ovat yhteydessä toisiinsa viestiverkon välityksellä. Käyttäjän kannalta tämä on yksi kokonaisuus, joka tarjoaa joukon erilaisia palveluja. Yksinkertaisimmillaan tällainen järjestelmä on esimerkiksi toimistossa toimiva muutaman palvelimen yhteen kytketty verkko, jollaisia oli jo 1980-luvulla. Hajautettuja järjestelmiä on siis ollut yleisesti käytössä jo kauan ennen nykyisiä pilvipalveluita. Järjestelmän eri osat toimivat usein rinnakkain. Tämä koskee myös järjestelmän solmujen välistä viestintää. Tästä rinnakkuudesta johtuen hajautetun järjestelmän toiminta on usein epädeterminististä ja vaikeasti ennustettavaa. Tämä korostuu etenkin silloin, kun järjestelmän koko kasvaa. Tällaisen verkon toiminnassa pysymisen kontrollointi on verkon koon kasvaessa myös yhä tärkeämpää. Hajautettujen järjestelmien kasvaessa yhä suuremmiksi ruvettiin etsimään keinoa näiden järjestelmien suorituskyvyn ja vakauden parantamiseksi. Pilvipalvelut ovat syntyneet pitkälti tämän tutkimustyön seurauksena.

Hajautettujen järjestelmien kehittyminen alkaa varsinaisesti 1980-luvulta, kun PC-koneiden aikakausi alkoi. Tällöin, ja laajenevassa määrin 1990-luvulla, alettiin muodostaa ns. klustereita eli palvelinfarmeja. Nämä laajenivat niin kutsutuiksi peer to peer -järjestelmiksi eli P2P-järjestelmiksi [Rodrigues and Druschel 2010]. Ne olivat ensimmäisiä hajautettuja järjestelmiä, joissa vakavasti panostettiin skaalautuvuuteen järjestelmän solmumäärän kasvaessa. P2P-järjestelmissä useiden niitä ylläpitävien tahojen resurssit on yhdistetty ja näin on saatu aikaan skaalautuva ja joustavasti kasvava järjestelmä. Ne kehittyivät tarpeesta jakaa informaatiota (esimerkiksi musiikkia) koko internetin laajuisesti. Nämä ovat osoittautuneet erittäin suosituiksi. P2P-järjestelmien heikkouksia ovat kuitenkin olleet niiden hallitsemattomuus ja kaupallisen hyödyntämisen hankaluus. Pilvipalvelut ovat pitkälti syntyneet, kun P2P-järjestelmien heikkouksia on ryhdytty parantamaan.

Mellin ja Grancen [2011] määritelmän mukaan pilvipalvelu tai pilvi-infrastruktuuri on laitteiston ja ohjelmiston muodostama järjestelmä, joka täyttää määrätyt pilvipalvelun oleelliset vaatimukset. Näitä vaatimuksia ovat seuraavat:

- Itsepalvelu pyydettäessä. Asiakas voi automaattisesti säädellä tarvitsemiaan resursseja, kuten palvelinaikaa ja -tilaa ilman inhimillistä vuorovaikutusta palvelun tarjoajan puolelta.
- Riittävän laaja verkkoon pääsy. Palvelut ovat käytettävissä yli verkon ja niihin päästään käyttäen jotain standardimekanismia, joka tukee riittävän montaa eri laitetta ja laitealustaa, kuten kännykät, tabletit, kannettavat ja työasemat.
- Resurssien yhdistäminen. Palveluntarjoajan resurssit on yhdistetty palvelemaan montaa asiakasta kerrallaan niin sanotun monen asiakkaan mallin mukaan. Tarvittavat fyysiset ja virtuaaliset resurssit sidotaan käyttöön asiakkaiden tarpeiden mukaan. Tämän päivän pilvipalveluihin kuuluu myös resurssien sijainnista riippumattomuus käyttäjän kannalta.

Asiakkaalla ei siis ole, eikä tarvitse olla, tietoa siitä, missä hänen tarvitsemansa resurssi sijaitsee.

- Hyvä tai nopea joustavuus. Järjestelmän tarjoamat palvelut pystytään skaalaamaan nopeasti ylös- tai alaspäin asiakkaiden tarpeiden mukaan. Tällä pyritään siihen, että asiakas ei kohtaisi mitään rajoitteita ottaessaan palveluita käyttöön.
- Mitattu palvelu. Pilvipalveluissa on kontrollijärjestelmä, jonka avulla palveluiden käyttöä voidaan seurata, valvoa ja raportoida. Tämä lisää palvelujen läpinäkyvyyttä sekä palveluntarjoajille että asiakkaille.

Pilvipalvelu voidaan myös nähdä kahden eri kerroksen, eli fyysisen ja abstraktin kerroksen järjestelmänä. Fyysinen kerros sisältää kaikki laitteistoresurssit, joita tarvitaan pilvipalveluiden tuottamiseen. Näitä yleensä ovat esimerkiksi palvelimet, levytila ja verkkokomponentit. Abstrakti kerros muodostuu fyysisen kerroksen päälle muodostetuista ohjelmistoista ja bisneslogiikasta. Käsitteellisesti pilvipalvelu siis muodostuu fyysisestä kerroksesta ja tämän päällä olevasta abstraktista kerroksesta. Asiakkaan näkökulmasta pilvipalvelut koostuvat kolmesta eri palvelumallista [Mell and Grance 2011]:

- SaaS (Software as a Service). Tässä mallissa asiakas käyttää palveluntarjoajan pilvipalvelussa toimivia ohjelmia. Asiakas ei suoraan kontrolloi millään tavalla pilviympäristössä olevaa verkkoa, palvelimia, käyttöjärjestelmiä, levytilaa tai edes pilviympäristössä toimivia ohjelmia. Kaupallisia SaaS-palveluita ovat esim. Google Docs ja Microsoft Office on Demand.
- PaaS (Platform as a Service). Tässä mallissa asiakas voi sijoittaa pilveen itse tekemiään tai hankkimiaan ohjelmistoja. Ohjelmistot on tehty käyttäen palveluntarjoajan luomia ohjelmointikieliä, kirjastoja, palveluita ja työvälineitä. Asiakas voi suoraan kontrolloida pilvipalveluun toimittamiaan ohjelmistoja. Kaupallinen PaaS-palvelu on esim. Google's AppEngine.
- IaaS (Infrastructure as a Service). Tässä mallissa asiakas voi sijoittaa pilveen käytännössä mitä tahansa ohjelmistoja mukaan lukien käyttöjärjestelmät. Kaupallisia IaaS-palveluita ovat esim. Amazon WebServices ja Microsoft Azure.

Kaatumisilmaisinten teoriaa on tutkittu kauemmin kuin pilvipalveluita varsinaisesti on ollut olemassa ja kaatumisilmaisimet kuuluvat olennaisena osan tämän päivän pilvipalveluihin. Varhaisimmat artikkelit aiheesta ovat 1980-luvun lopusta. Teoreettisesti kaatumisilmaisimia ja yleensä viestintää hajautetuissa järjestelmissä alettiin voimallisesti tutkia 1980-luvulla, jolloin jopa miljoonia solmuja sisältävät P2P-järjestelmät alkoivat yleistyä. Tarve jonkinlaisen kaatumisilmaisimen käyttöön hajautetuissa järjestelmissä kehittyi vähitellen järjestelmien koon kasvaessa. Järjestelmän solmujen määrän kasvaessa myös sen epävakaus kasvoi vähitellen kestämättömälle tasolle. Lisäksi solmujen kaatuminen hajautetuissa järjestelmissä on sitä yleisempää, mitä suurempi verkko on. Oletetaan, että on käytössä 120 palvelimen muodostama järjestelmä ja jokainen näistä palvelimista menee nurin keskimäärin kerran kymmenessä vuodessa.

Tällöin aika, jonka kuluessa voidaan odottaa, että jokin verkon solmu kaatuu, on 120 kuukautta/120 eli yksi kuukausi. Kun palvelinten määrä kasvaa 12000 kappaleeseen, keskimääräinen kaatumisaika on n. 7.2 tuntia. Järjestelmässä, jossa palvelimia on 120000, keskimääräinen kaatumisaika on jo alle tunnin. Näiden ongelmien vuoksi ruvettiin tutkimaan jonkinlaisen verkon vakauden ja solmujen kaatumisen hallinnan rakentamista hajautettuihin järjestelmiin. Tämä johti kaatumisilmaisinten tutkimiseen ja kehittämiseen. Koska kaatumisilmaisinten toiminta perustuu solmujen väliseen kommunikointiin hajautetussa järjestelmässä, myös viestintää hajautetuissa järjestelmissä yleensä alettiin tämän myötä tutkia tiiviimmin. Eräänlaisia käytännön koelaboratoriota kaatumisen havaitsemiselle ja viestinnälle hajautetuissa järjestelmissä olivat pilvipalvelujen edeltäjät eli juuri P2P-järjestelmät. Näistä esimerkkinä mainittakoon Napster ja Gnutella. Näissä oli käytössä jo nykyisenkaltaisia tekniikoita solmujen kaatumisen havaitsemiseen.

Kaatumisen havaitseminen, kuten yleensä muukin informaatioliikenne, perustuu hajautetussa järjestelmässä solmujen väliseen viestintään. Solmujen välisen viestinnän kannalta keskeinen ominaisuus on hajautetun järjestelmän rakenne. Viestinnän vaativuuteen vaikuttaa keskeisesti se, onko järjestelmä synkroninen vai epäsynkroninen. Vaikeinta viestinnän toteuttaminen on epäsynkronisessa järjestelmässä. Tämä johtuu epäsynkronisen järjestelmän rakenteesta. Täysin epäsynkronisessa järjestelmässä ei määritelmän mukaisesti ole lainkaan ns. rajoja. Tällaisessa järjestelmässä esimerkiksi viestin kulkeminen solmusta toiseen voi kestää mielivaltaisen kauan. Tästä seuraa se ongelma, että tällöin solmun toiminnan tarkkailu ei voi luotettavasti perustua viestien perillemenon tutkimiseen. Käytännön toteutuksissa tämä ongelma voidaan kuitenkin yleensä kiertää. Yleensä oletetaan, että viestien läpimeno ei voi normaalissa tilanteessa kestää määrättömän kauan, kuten epäsynkronisen verkon teoreettinen määritelmä ääritapauksessa edellyttäisi, vaan aina on olemassa jokin määräaika, jonka jälkeen todetaan, että viestit jostain solmusta eivät enää kulje, vaan solmu on kaatunut. Toinen epäsynkronisen järjestelmän ongelma liittyy aikaan yleensä. Epäsynkronisessa hajautetussa järjestelmässä jokaisella solmulla voi olla oma aikansa (kellonsa). Paitsi, että jokaisella solmulla voi hajautetussa järjestelmässä olla oma paikallinen aika, tuo aika voi kulua myös eri tahtiin kuin muissa solmuissa. Tämän vuoksi eräs varsin hankala, mutta keskeinen toiminto hajautetuissa järjestelmissä on solmujen paikallisten aikojen synkronointi siten, että on olemassa jokin yleinen vertailukelpoinen aika, kun tutkitaan viestien kulkemista solmusta toiseen.

Tässä tutkielmassa esitellään kaatumisilmaisinten toiminta ja teoreettinen tausta sekä luodaan katsaus kaatumisilmaisinten erilaisiin toteutuksiin. Ensin esitellään kaatumisilmaisinten toimintaan vaikuttavat muut hajautetun verkot ominaisuudet ja tekijät. Erityisesti viestiliikenne hajautetussa järjestelmässä ja epäsynkronisen järjestelmän synkronointi käsitellään yksityiskohtaisesti, koska nämä ovat keskeisiä asioita kaatumisilmaisinten toiminnan kannalta. Tämän jälkeen analysoidaan tunnetuimmat ja ominaisuuksiltaan parhaat kaatumisilmaisimet ja pyritään löytämään niistä parhaiten teoreettista mallia vastaava.

Luvussa 2 syvennytään hajautetun järjestelmän rakenteen viestinnälle aiheuttamiin haasteisiin. Kohdassa 2.1 käsitellään tarkemmin järjestelmän synkronisuuden ja epäsynkronisuuden vaikutusta viestintään. Kohdassa 2.2 käsitellään järjestelmän epäsynkronisuuden aiheuttamia ongelmia

viestinnässä. Tässä määritellään myös hajautetun järjestelmän ns. yksimielisyysperiaate ja sen likimääräinen ratkaisu. Kohdassa 2.3 käsitellään solmujen synkronointi hajautetussa järjestelmässä.

Tutkielman kolmannessa luvussa käsitellään viestintää hajautetussa järjestelmissä yleensä. Kohdassa 3.1 käsitellään ryhmäviestinnän puumallin perusmalli ja paranneltu malli, kohdassa 3.2 syvennytään juoruprotokollan mukaiseen viestintään. Kohdassa 3.3 käsitellään ryhmäviestinnän edellyttämiä ominaisuuksia hajautetun järjestelmän solmuissa kuten jäsenlistoja ja kohdassa 3.4 käsitellään järjestys ja sen merkitys eri viestintäprotokollissa. Luvussa 4 syvennytään itse kaatumisilmaisinten toteutukseen. Tässä luvussa käsitellään kaatumisilmaisinten tärkeät ominaisuudet ja esitellään ns. teoreettinen ihannemalli kaatumisilmaisimesta. Kohdissa 4.2 ja 4.3 esitellään vastaavasti sykkeeseen ja juoruamiseen perustuvat kaatumisilmaisimet. Kohdassa 4.4 esitellään tiedon levittämiseen perustuva kaatumisilmaisin ja kohdassa 4.5 vertaillaan esiteltyjä kaatumisilmaisimia ja valitaan niistä suorituskyvyltään paras. Luku 5 on yhteenveto, jossa käydään läpi aiemmissa luvuissa syntyneet tulokset ja päätelmät.

2. Hajautetun järjestelmän rakenteen merkitys viestinnässä

Järjestelmän rakenteella on perustavanlaatuinen merkitys viestintään hajautetussa järjestelmässä. Käytännössä tämä kulminoituu siihen, onko järjestelmä synkroninen vai epäsynkroninen. Epäsynkronisessa järjestelmässä viestinnän toteuttaminen taloudellisesti ja luotettavasti on paljon vaikeampaa kuin synkronisessa järjestelmässä. Lisäksi on olemassa hajautetun järjestelmän toiminnan kannalta seuraavat kolme tärkeää ominaisuutta [Gilbert and Lynch 2012]:

- Eheys. Vaikka järjestelmässä olisi (ja yleensä on) useampia solmuja, jotka lukevat ja kirjoittavat dataa yhteiseen tietovarastoon samanaikaisesti, jokainen solmu näkee tuon tietovaraston samassa tilassa millä tahansa ajan hetkellä. Minkä tahansa solmun tekemä lukuoperaatio palauttaa aina viimeisimmän päivitetyn arvon.
- Käytettävyys. Järjestelmän sallii luku- ja kirjoitusoperaatiot kaikille solmuille milloin tahansa ja operaatiot palauttavat pyydetyn arvon nopeasti.
- Osittumistoleranssi. Järjestelmän hajotessa esim. jonkin koordinaattorisolmun kaatumisen vuoksi kahteen tai useampaan osaan, jotka eivät enää voi kommunikoida keskenään, järjestelmä toimii silti edelleen siten, että kaksi edellä mainittua vaatimusta täyttyy.

Niin sanotun CAP-teoreeman (engl. Consistency, Availability and Partition tolerance theorem) [Gilbert and Lynch 2012] mukaan hajautetussa järjestelmässä ei ole mahdollista kaikissa olosuhteissa taata näiden kaikkien ominaisuuksien optimaalista toimintaa vaan vain kaksi kolmesta. Näistä kolmesta ominaisuudesta hajautetun järjestelmän toiminnan kannalta tärkein on osittumistoleranssi. Hajautetun järjestelmän rakenteen vuoksi on välttämätöntä, että järjestelmän hajotessa kahteen tai useampaan keskenään kommunikoimattomaan osaan nuo osat pystyvät toimimaan myös itsenäisesti siten, että eheys ja käytettävyys eivät vaarannu. Tämä tarkoittaa valitettavasti sitä, että CAP-teoreemasta seuraava ongelma joudutaan kiertämään tinkimällä kahdesta muusta ominaisuudesta. Yleensä hajautetuissa järjestelmissä se ominaisuus, jonka kustannuksella osittumistoleranssi säilytetään, on eheys. Tämä tehdään noudattamalla niin kutsuttua heikkoa eheyttä. Heikko eheys tarkoittaa sitä, että järjestelmän päivitysoperaatiot pyritään aina saattamaan loppuun kaatumisista huolimatta. Tarkemmin siis niin, että jos jonkin solmun päivitysoperaatiot pysähtyvät tai epäonnistuvat, päivitettävät tiedot tallennetaan. Kun järjestelmä on toipunut sen verran, että päivitykset jälleen jatkuvat, järjestelmä jatkaa päivittämistä. Tässä siis tavallaan päivitetään ”päivitysaaltoina” jälkeenjääneitä arvoja siten, että lopputuloksena olisi ajantasainen tilanne kaikkiin solmuihin. Tämä ei käytännössä takaa sitä, että päivitykset aina toimisivat ja joka tilanteessa kaikki järjestelmän solmut aina näkisivät ajantasaisen tiedon. Heikko eheys toimii parhaiten järjestelmässä, jossa tietojen päivitysrytmi on ainakin ajoittain riittävän harva. Eheydestä tingitään siksi, että kolmas teoreeman ominaisuus eli käytettävyys on hajautetun järjestelmän käyttäjien kannalta tärkeä ominaisuus. Erityisesti pilvipalveluissa palvelun ns. reaktioajan eli asiakkaiden pyyntöihin reagointiin kuluvan ajan, joka seuraa suoraan käytettävyydestä, täytyy olla riittävän lyhyt.

2.1. Järjestelmän synkronisuuden ja epäsynkronisuuden vaikutus viestintään

Kuten aiemmin on jo todettu, tehokkaan viestintätavan ja siten myös tehokkaan kaatumisilmaisimen löytäminen riippuu aivan olennaisesti järjestelmän rakenteesta. Epäsynkroninen hajautettu järjestelmä voidaan määritellä seuraavasti [Arjomandi et al. 1983]:

- Epäsynkroninen järjestelmä muodostuu määrätystä määrästä prosesseja (solmuja).
- Jokaisella prosessilla on kulloisenakin ajanhetkenä jokin määrätty tila.
- Jokainen prosessi voi muuttaa tilaansa.
- Kun prosessin tilan muuttuu tai prosessi lähettää viestin jollekin muulle prosessille, minkä seurauksena kyseisen prosessin tila muuttuu, kutsutaan tätä tapahtumaksi.
- Jokaisella prosessilla on oma paikallinen kellonsa – prosessin sisäisille tapahtumille voidaan asettaa aikaleimat ja asettaa nämä lineaariseen järjestykseen.
- Prosessien väliset tapahtumat eivät ole järjestelmän rakenteen mukaan missään järjestyksessä – tapahtuman T prosessissa A ja tapahtuman T’ prosessissa B välistä järjestystä ei hajautetussa järjestelmässä välttämättä voi lainkaan määritellä ilman prosessien synkronointia.

Hajautetuista järjestelmistä erityisesti pilvipalvelut ovat usein epäsynkronisia. Tämä voi johtua pelkästään jo siitä, että tällaisen järjestelmän toimivia osia saattaa olla maapallon eri aikavyöhykkeillä, jolloin yllä olevan määritelmän mukaan kyse on automaattisesti epäsynkronisesta järjestelmästä.

2.2. Viestinnän ongelmat epäsynkronisessa järjestelmässä

2.2.1. Yksimielisyysongelma

Yksimielisyysongelma on yksi keskeisimpiä asioita hajautetun järjestelmän toiminnan kannalta. Se vaikuttaa suoraan ainakin seuraaviin hajautetun järjestelmän solmujen toimintoihin [Lamport 2005]:

- Luotettava ryhmäviestintä. On kyettävä varmistamaan, että kaikki solmut ottavat vastaan kaikki samat yhteiset päivitykset samassa järjestyksessä kuin muutkin solmut.
- Jäsenlistat ja kaatumisen havaitseminen. Kukin solmu pitää paikallista listaa muiden järjestelmän solmujen tilasta, ja aina kun jokin solmu kaatuu tai poistuu järjestelmästä, muille solmuille päivitetään tästä tieto samanaikaisesti.
- Koordinaattorisolmun valinta. Kun järjestelmälle valitaan uusi koordinaattorisolmu, kaikille muille solmuille on saatava tieto tästä.
- Toisensa poissulkevuus. Varmistetaan jonkin kriittisen prosessin toisensa poissulkeva (vain yksi prosessi kerrallaan) käyttö.

Tarkastellaan seuraavaksi yksimielisyysongelman muodollista määritelmää [Lamport 2005]. Yksimielisyysongelman näkökulmasta hajautetun järjestelmän voidaan ajatella jakautuvan ehdottajaprosesseihin ja oppijaprosesseihin. Ehdottajaprosessi ehdottaa tai ilmoittaa jonkin arvon, joka oppijaprosessien täytyy ”oppia” eli havaita siten, että kaikki oppijaprosessit oppivat saman

arvon. Tämä on ns. perinteinen yksimielisyysongelma. Sen ratkaisulla on neljä perusvaatimusta [Lamport 2005]:

- Epätriviaalisuus. Jokaisen opitun tai havaitun arvo täytyy olla jonkin ehdottajaprosessin ilmoittama arvo.
- Vakaus. Oppijaprosessi voi oppia vain yhden arvon kerrallaan.
- Eheys tai turvallisuus. Kaksi eri oppijaprosessia eivät voi oppia eri arvoa samasta ehdottajaprosessista. Toisen sanoen, kun ehdottajaprosessi ilmoittaa jonkin arvon, kaikkien oppijaprosessien pitää oppia tämä sama arvo.
- Elävyys (liveness). Kaikki oppijaprosessit lopulta oppivat jonkun arvon, jos sellaista on ehdotettu.

Yksimielisyysongelman ratkaiseminen on huomattavasti hankalampaa epäsynkronisessa kuin synkronisessa järjestelmässä. Tämä johtuu epäsynkronisen ja synkronisen järjestelmän rakenteellisista eroista. Synkronisessa järjestelmässä viestin kulkeminen solmusta toiseen kestää rajallisen määritellyn ajan. Toisin sanoen on olemassa jokin tietty tarkkaan määritelty aikaraja, jonka kuluessa viestit aina tulevat perille. Lisäksi tämä aikaraja on koko järjestelmän laajuinen. Epäsynkronisessa järjestelmässä tällaista aikarajaa ei ole, vaan viestien kulkeminen paikasta toiseen voi kestää mielivaltaisen kauan. Oman ongelmansa aiheuttaa myös se, että epäsynkronisessa järjestelmässä kellonaikojen ja jopa kellojen nopeudet voivat vaihdella solusta toiseen. On osoitettu, että algoritmi, jolla edellä määritelty yksimielisyys voidaan saavuttaa kaikissa tilanteissa, on mahdollista vain synkronisessa järjestelmässä [Fischer et al. 1985]. Epäsynkronisissa järjestelmissä päästään vain likimääräisiin ratkaisuihin, jotka toimivat yleensä tyydyttävästi, mutta eivät kaikissa mahdollisissa tilanteissa. Toisin sanoen, mikä tahansa protokolla tai algoritmi kehitetäänkin, aina on olemassa pahimman mahdollisen tapauksen skenaario, jossa solmujen kaatumiset ja viestien perillemenon viivästyminen ovat niin suuria haittatekijöitä, että ne estävät yksimielisyysongelman ratkaisemisen. Edelleen pohjautuen yllä olevaan on osoitettu, että jos olisi olemassa algoritmi, joka toimii yksimielisyysongelman ratkaisuna epäsynkronisessa järjestelmässä, sama ratkaisu toimii myös synkronisessa järjestelmässä, mutta ei päinvastoin [Fischer et al. 1985]. Koska lähes kaikki pilvipalvelut käytännössä ovat epäsynkronisia hajautettuja järjestelmiä, on tämän ongelman likimääräinen ratkaiseminen ollut yksi niiden toiminnan kannalta keskeinen asia. Seuraavassa alakohdassa perehdytään erääseen tapaan ratkaista tämä ongelma likimääräisesti epäsynkronisessa järjestelmässä.

2.2.2. Yksimielisyysongelman likimääräinen ratkaisu

Yksimielisyysongelman likimääräisesti ratkaiseva algoritmi on kirjallisuudessa nimetty Paxos-algoritmiksi tekijänsä mukaan [Lamport 2001]. Algoritmin yksinkertaistettuun versioon kuuluu kolme erilaista agenttia tai niin sanottua osallistujaa: ehdottajaprosessit, hyväksyjäprosessit ja oppijaprosessit. Nämä ovat siis rooleja, joita jokainen verkon solmu vuorollaan esittää. Koska algoritmi toimii epäsynkronisessa järjestelmässä, sen tulee erityisesti huomioida seuraavat asiat:

- Kaikki agenttiprosessit toimivat mielivaltaisella nopeudella ja voivat milloin tahansa kaatua, poistua järjestelmästä ja käynnistyä uudelleen. Koska mikä tahansa agenttiprosessi voi kaatua sen jälkeen, kun algoritmin ehdottama arvo on valittu, pitää sillä olla jokin mekanismi, jolla se pystyy muistamaan, mikä tuo arvo oli, kun se kaatumisen jälkeen käynnistyy uudelleen.
- Viestien välitysaika voi olla mielivaltaisen pitkä, samoja viestejä voi esiintyä useaan kertaan ja viestejä voi myös kadota matkalla.

Tässä algoritmissa ehdottajaprosessien lähettämistä arvoista pidetään kirjaa siten, että jokaiseen ehdotettuun arvoon liitetään järjestysnumero. Näin jokainen ehdotettu arvo tosiasiassa sisältää itse arvon ja tiedon siitä, monesko ehdotus tämä on. Ehdotettu arvo tulee valituksi eli oppijaprosessien oppimaksi, kun sen on hyväksynyt enemmistö hyväksyjäprosesseista. Luonnollisesti kaikissa valituissa ehdotuksissa on oltava sama arvo. Tämä saadaan aikaan ehdotettujen arvojen numeroimisella ja muutamalla lisäehdolla:

- E1. Olkoon valittu arvo v saatu ehdotuksella p. Tämän jälkeen jokainen valittu arvo, jolla korkeampi järjestysnumero kuin ehdotuksella p, on sama arvo v.
- E2. Jos hyväksytään ehdotus, jonka arvo on v, tällöin mikä tahansa järjestysnumeroltaan korkeampi ehdotus, jonka mikä tahansa hyväksyjä on valinnut, sisältää arvon v.

Ehto E1 takaa sen, että vain yksi arvo tulee valittua. Ehto E2 puolestaan takaa sen, että jonkin ehdotuksen hyväksyy vähintään yksi hyväksyjäprosessi. Kommunikoinnin asynkroninen luonne aiheuttaa kuitenkin vielä lisäongelman. Oletetaan, että jokin ehdottajaprosessi on kaatuneena ja algoritmin jonkin kierroksen ajan ja toipuu tämän kierroksen jälkeen ja lähettää uuden arvo hyväksyttäväksi korkeammalla järjestysnumerolla. Tällöin, jos tämä arvo hyväksytään, sillä on korkeampi järjestysnumero kuin jo hyväksytyillä arvoilla, eikä ehto E2 enää pädekään. Tämän vuoksi ehto E2 on muutettava seuraavaan muotoon:

> $E2_b$: Jos hyväksytään ehdotus, jonka arvo on v, niin mikä tahansa järjestysnumeroltaan korkeampi ehdotus, jota mikä tahansa ehdottajaprosessi on ehdottanut sisältää arvon v.

Lyhyesti muotoiltuna algoritmi ehdotusten esittämiseksi on seuraava:

1. Ehdottajaprosessi e_p valitsee uuden ehdotuksen, jonka järjestysnumero on x, ja lähettää viestin jokaiselle jonkun hyväksyjäjoukon jäsenelle, jossa se ensinnäkin pyytää, että hyväksyjä ei enää hyväksy ehdotuksia, joiden järjestysnumero on pienempi kuin x. Toiseksi, jos hyväksyjäprosessi on jo vastaanottanut jonkun ehdotuksen, jonka järjestysnumero on pienempi kuin x, se lähettää tämän ehdotuksen ehdottajaprosessille e_p. Tämä on niin sanottu valmistava ehdotus.
2. Jos ehdottajaprosessi e_p saa pyytämänsä vastaukset hyväksyjäprosessien enemmistöltä, se voi lähettää varsinaisen ehdotuksen, jonka arvo v on joko järjestysnumeroltaan korkeimman e_p:n vastaanottaman ehdotuksen arvo tai e_p:n alkuperäisen eli valmistavan ehdotuksen arvo, jos e_p ei vastaanottanut ehdottajaprosesseilta muita ehdotuksia.

Tässä siis muotoiltiin ehdottajaprosessin algoritmi. Kuten yllä on kuvattu, hyväksyjäprosessit ottavat siis ehdottajaprosesseilta vastaan kahdenlaisia viestejä tai ehdotuksia: valmistavia ehdotuksia ja varsinaisia ehdotuksia. Hyväksyjän toiminnassa olennaista on se, että se vastaa eli hyväksyy

varsinaisen ehdotuksen vain silloin, kun se ei ole luvannut olla hyväksymättä tällaista ehdotusta. Tällä tarkoitetaan siis tilannetta, jossa hyväksyjä ei ole vielä ehtinyt vastaanottaa jonkin muun ehdottajan ilmoitusta olla hyväksymättä järjestysnumeroltaan ko. ehdottajan ehdotuksia pienempiä ehdotuksia eli muodollisesti ilmaistuna:

- Hyväksyjäprosessi voi hyväksyä varsinaisen ehdotuksen, jonka järjestysnumero on x, jos ja vain jos se ei ole ottanut vastaan valmistavaa ehdotusta, jonka järjestysnumero on suurempi kuin x.

Kun hyväksyjäprosessin ja erottajaprosessin toiminta tässä algoritmissa summataan yhteen, saadaan arvon valitsemiseksi alla kuvattu kaksivaiheinen algoritmi:

1. Vaihe. Hyväksyjä valitsee ehdotuksen numero n ja lähettää valmistavan ehdotuksen numero x enemmistölle hyväksyjistä. Kun hyväksyjä ottaa vastaan valmistavan ehdotuksen, jonka järjestysnumero n on suurin kaikista valmistavista ehdotuksista, joihin se on jo vastannut, se vastaa tähän lupaamalla, ettei enää hyväksy varsinaisia ehdotuksia, jotka ovat järjestysnumeroltaan pienempiä kuin x ja liittää vastaukseen lisäksi järjestysnumeroltaan korkeimman ehdotuksen, jonka se on hyväksynyt.
2. Vaihe. Kun ehdottaja ottaa vastaan vastausviestin, jonka järjestysnumero on x, lähettämiinsä valmistaviin ehdotuksiin hyväksyjien enemmistöltä, se lähettää hyväksymispyyntönä kaikille näille hyväksyjille ehdotuksen, jonka numero on x. Tämän ehdotuksen arvoksi tulee sen ehdotuksen arvo, jonka järjestysluku on korkein niistä, jotka edottaja sai vastauksena valmistaviin ehdotuksiinsa. Kun hyväksyjä vastaanottaa hyväksymispyynnön ehdotukselle x, se hyväksyy tämän ehdotuksen, ellei se ole aiemmin ottanut vastaan valmistavaa ehdotusta, jonka järjestysnumero on sama kuin x.

Kun arvon valinta on suoritettu, se täytyy vielä ”oppia” eli kaikkien prosessien tulee tietää, että juuri kyseinen arvo on valittu. Tätä varten järjestelmän solmuille tarvitaan vielä kolmas rooli eli oppijat. Näiden avulla järjestelmän solmuille ”opetetaan”, minkä arvon hyväksyjien enemmistö on hyväksynyt. Oppiakseen valitun arvon oppijaprosessin täytyy saada tietoonsa, minkä ehdotuksen hyväksyjien enemmistö on hyväksynyt. Yksinkertaisin ratkaisu tähän olisi se, että jokainen hyväksyjä aina hyväksyessään jonkin ehdotuksen lähettää siitä myös tiedon kaikille oppijaprosesseille. Tällöin oppijaprosessit saisivat tietoonsa valitun arvon mahdollisimman nopeasti. Tämä on kuitenkin raskas ratkaisu, sillä se edellyttää, että jokainen hyväksyjä aina hyväksyessään jonkin ehdotuksen lähettää siitä tiedon kaikille oppijaprosesseille. Viestejä tulisi siis jokaista hyväksyntää kohden hyväksyjäprosessien määrä kerrottuna oppijaprosessien määrällä. Kevyempi ratkaisu on, että jokainen hyväksyjä hyväksyessään ehdotuksen tiedottaa tästä vain yhdelle valitsemalleen oppijalle ja tämä sitten välittää tiedon muille oppijoille.

2.3. Hajautetun järjestelmän solmujen synkronointi

Hajautetun järjestelmän solmujen synkronointi tarkoittaa jonkin määrätyn yhteisen tavan löytämistä, jolla mitataan hajautetussa järjestelmässä kuluvaa aikaa. Tämä on tärkeää monestakin syystä. Yksi

on se, että järjestelmän toiminta takaisi aiemmin määritellyn täydellisyysominaisuuden. Seuraavassa on esimerkki siitä, mitä voi tapahtua, jos epäsynkronista hajautettua järjestelmää ei synkronoida.

Oletetaan, että käytössä on lentomatkan varausjärjestelmä pilvipalvelussa. Palvelin A vastaanottaa lipunostovarauksen lennolle ABC 123. Ennen tätä varausta lennolle on vain yksi paikka jäljellä. Palvelin A merkitsee ostotapahtuman tapahtuma-ajaksi 09:15:32,15 käyttäen omaa paikallista kelloaan ja kuittaa asiakkaalle varauksen vahvistetuksi. Lento on nyt täynnä. Palvelin A lähettää palvelimelle B viestin ”Lento täynnä”. Palvelin B merkitsee omaan lokiinsa ”Lento täynnä” ja aikaleimaksi oman paikallisen kellonsa ajan 09:10:32,15. Tämän jälkeen palvelin C saa toiselta asiakkaalta pyynnön varata lippu samalle lennolle. C tutkii A:n ja B:n lokeja saadakseen selville, onko lennolle vielä paikkoja. Se löytää merkinnän varatusta lipusta ajalta 09:15:32,15. Samalla se kuitenkin havaitsee, että lento on merkitty täyteen varatuksi jo 09:10:32,15. Nyt se ei enää pysty päättelemään, onko lennolle vielä paikkoja jäljellä vai ei. Tämä voi johtaa esimerkiksi siihen, että palvelin C päättelee, että lennolle on tullut lisää paikkoja ja hyväksyy varauksen, jolloin syntyy tuplavaraus. Internetissä toimivien hajautettujen järjestelmien, kuten pilvipalvelujen, synkronointi on vaikeaa lähinnä järjestelmien epäsynkronisesta rakenteesta johtuen. Näissä järjestelmissä ei ole teoreettisesti määriteltyä rajaa sille, miten kauan viestin kulkeminen solmusta toiseen voi kestää ja miten kauan sen käsittely vie aikaa.

Epäsynkronisen järjestelmän kahden eri solmun kellonaikojen eroavuutta mitataan seuraavilla kahdella käsitteellä: aikaero ja ajan taajuusero. Aikaero tarkoittaa yksinkertaisesti suhteellista eroa kahden eri solmun paikallisissa kellonajoissa. Ajan taajuuserolla tarkoitetaan suhteellista eroa kahden eri prosessin kellonajan kulumisessa. Taajuusero kahden eri solmujen kellojen välillä on kuin kahden eri auton nopeusero tiellä. Samoin aikaero on kuin kahden auton välinen etäisyys tiellä. Nollasta poikkeava aikaero tarkoittaa, että solmujen kelloja ei ole synkronoitu. Taajuusero on ikävämpi ominaisuus näistä kahdesta, sillä se aiheuttaa aikaeron lisääntymistä. Tätäkin voidaan havainnollistaa autoesimerkillä: kun kaksi autoa liikkuu tiellä eri nopeudella (kellojen synkronoinnissa taajuusero), niiden välinen etäisyys (kellojen synkronoinnissa aikaero) kasvaa.

Hajautetun järjestelmän solmut voidaan synkronoida joko järjestelmän ulkoisella tai sisäisellä synkronoinnilla [Cristian 1989]. Ulkoisessa synkronoinnissa järjestelmän solmujen kellot synkronoidaan jonkin ulkoisen kellon mukaan, eli järjestelmän jokaisen solmun S^i oman kellon aika voi erota korkeintaan määrätyn arvon D verran järjestelmän ulkopuolisesta vertailukellosta C. Jos S^i on järjestelmän minkä tahansa solmun paikallinen kello ja D on vertailuaikojen yläraja ja C ulkopuolinen vertailukello, niin tällöin tulee aina päteä $|S^i - C| < D$. Sisäisessä synkronoinnissa samantapainen vertailu tehdään järjestelmän jokaiselle solmuparille. Jokaisen solmuparin S^i, S^j kellot saavat erota vain määrätyn arvon verran toisistaan eli, jos D on tuo yläraja, ehdon $|S^i - S^j| < D$ tulee aina toteutua kaikille solmuille i ja j. Seuraavissa alakohdissa käsitellään neljä erilaista tapaa synkronoida epäsynkronisen järjestelmän solmut.

2.3.1. Cristianin synkronointialgoritmi

Cristianin algoritmi [Cristian 1989] perustuu järjestelmän solmujen ulkoiseen synkronointiin. Algoritmin perusideana on verrata jokaisen solmun kellonaikaa johonkin ulkoiseen kelloon ja säätää solmujen kellonajat tämän mukaan oikein. Solmu S lähettää synkronointipalvelimelle P kyselyn, mitä kello on. Palvelin P palauttaa S:lle kellonajan T ja tämä säätää oman kellonsa tuohon aikaan T vastaanotettuaan P:n viestin. Tässä perusideassa on suurimpana puutteena se, että S:n kellonajan asetus on epätarkka. Siinä vaiheessa, kun S on saanut säädettyä kellonsa P:n ilmoittamaan aikaan T, on aikaa kulunut jo lisää, ja tämä johtaa siihen, että S:n kello on itseasiassa koko ajan jäljessä. Tämä epätarkkuus on suoraan verrannollinen viestiliikenteen viiveisiin hajautetussa järjestelmässä. Epäsynkronisessa järjestelmässä tämä muodostaa erityisen vakavan ongelman siksi, että noissa järjestelmissä viestien kulku solmusta toiseen voi kestää mielivaltaisen kauan. Tästä siis seuraa, että myös synkronoinnin epätarkkuus voi olla tässä mallissa mielivaltaisen suuri. Tämän korjaamiseksi varsinaisessa algoritmissa huomioidaan S:n P:lle lähettämän ja P:n S:lle palauttaman viestin välittämiseen kulunut kokonaisaika eli siis aika siitä, kun S lähettää P:lle viestin kysyäkseen kellonaikaa siihen, kun S on saanut P:ltä vastausviestin. Käytännön järjestelmissä tunnetaan yleensä solmusta toiseen kulkemisen minimiviive. Oletetaan, että viestin kulun S:stä P:n viive on vähintään min1 ja päinvastoin viestin kulun P:stä S:n viive on min2. Oletetaan vielä, että kokonaisaika on RTT. Tällöin kellonaika t, jonka P palauttaa S:lle, pitää todellisuudessa asettaa välille [t+min2,t+RTT-min1]. Käytännössä järkevintä on sijoittaa t:n arvo keskelle tätä väliä eli kohtaan t+(RTT+min2-min1)/2. Tällöin asetetun t:n arvon ja todellisen t:n arvon ero eli synkronoinnin virhemarginaali on siis enintään (RTT+min2-min1)/2. Virhemarginaali on sitä pienempi mitä tarkemmin minimiarvot min1 ja min2 tunnetaan. Tarkkuuteen vaikuttaa myös viestien nopeus. Mitä nopeammin viestit kulkevat, sitä pienempi on kokonaisaika RTT ja sitä pienempi synkronoinnin virhemarginaali.

2.3.2. NTP eli Network Time Protocol

Network Time Protocol [Mills 1992] on 1990-luvun alussa kehitetty hajautettujen järjestelmien synkronointistandardi. NTP-protokollassa järjestelmän solmut on järjestetty puuksi, jonka juurisolmu on solmu, jonka mukaan muiden solmujen kellot lopulta synkronoidaan. Synkronointi tapahtuu kuitenkin kerroksittain siten, että juurisolmun lapsisolmut synkronoivat kellonsa juurisolmun kellon mukaan ja juurisolmun lapsisolmujen lapsisolmut synkronoivat kellonsa juurisolmun lapsisolmujen kellojen mukaan ja niin edelleen.

NTP-protokolla toimii seuraavaan tapaan: Lapsisolmu l lähettää vanhemmalleen v algoritmin käynnistyspyynnön. Solmu v lähettää vastuksen ”pyyntö huomioitu” ja tallentaa tämän viestin viestiksi Vst1 sekä tallentaa tämän viestin lähetysajan ajanhetkeksi Ts1. Lapsisolmu puolestaan tallentaa viestin Vst1 vastaanottoajaksi Tr1. Nämä ajat ovat kummankin solmun paikallisten kellojen synkronoimattomia aikoja. Tämän jälkeen lapsisolmu lähettää vanhemmalleen kuittauksen viestissä Vst2 ja tallentaa tämän viestin lähetysajaksi Ts2 oman kellonsa mukaan. Kun vanhempi vastaanottaa tämän viestin, se tallentaa vastaanottoajaksi Tr2 oman kellonsa mukaan. Lopuksi vanhempisolmu

lähettää lapselleen tallentamansa ajat Ts1 ja Tr2. Lapsisolmulla on nyt käytettävissään seuraavat ajat kellonsa synkronoimiseen:

- Ts1, viestin Vs1 (vanhemmalta lapselle ”synkronoinnin käynnistyspyyntö huomioitu”) lähetysaika. Tallennetaan vanhempisolmun kellon mukaan.
- Tr1, viestin Vs1 vastaanottoaika. Tallennetaan lapsisolmun kellon mukaan.
- Ts2, viestin Vs2 (lapsisolmun vanhemmalleen lähettämä kuittaus viestistä Vs1) lähetysaika. Tallennetaan lapsisolmun kellon mukaan.
- Tr2, viestin Vs2 vastaanottoaika. Tallennetaan vanhempisolmun kellon mukaan.

Oletetaan, että solmujen kellojen todellinen ero on eroT. Todellinen ero eroT tulkitaan seuraavasti: jos vanhempisolmun kellonaika on t, vastaava lapsisolmun kellonaika t+eroT. Oletetaan lisäksi, että viestin Vs1 yhden suunnan verkosta aiheutuva viive on L1, ja viestin Vs2 vastaavasti L2. Tällöin ajat Tr1 ja Tr2 voidaan ilmaista seuraavasti:

Tr1=Ts1+L1+eroT

Tr2=Ts2+L2-eroT.

Näistä ratkaistaan eroT vähentämällä toinen yhtälö ensimmäisestä:

(Tr1-Ts1-L1)-(Tr2-Ts2-L2)=2eroT => eroT=(Tr1-Tr2+Ts2-Ts1)/2 +(L2-L1)/2.

Seuraa, että tällä tavoin synkronoinnin virhemarginaali on mahdollista supistaa (L1-L2)/2:n, eli synkronoinnin virhemarginaali ei voi koskaan olla suurempi kuin viestien kokonaisläpimenoaika lapsi- ja vanhempisolmun välillä.

Perusongelma sekä NTP:ssä että Cristianin algoritmissa on se, että vaikka kellojen synkronoimisen virhemarginaali saadaankin pieneksi, siitä ei koskaan päästä täysin eroon. Itse asiassa sitä ei voi säätää edes vakioksi, vaan se riippuu aina joistain verkon ominaisuuksista. Viime kädessä virhemarginaali johtuu siitä, että sitä, miten kauan viestin kulkeminen solmusta toiseen kestää, ei tiedetä tarkkaan, vaan se voidaan arvioida vain likimääräisesti. Tämän vuoksi on pyritty kehittämään menetelmiä, joilla järjestelmän kellot saataisiin synkronoitua siten, ettei niiden paikallisia aikoja tarvitsisi säätää. Yksi tällainen tapa on aikaleimojen käyttö, jota tutkitaan kahdessa seuraavassa alakohdassa.

2.3.3. Lamportin aikaleimat

Koska hajautetun järjestelmän kellojen synkronointi solmujen paikallista aikaa muuttamalla on osoittautunut vaikeaksi, ryhdyttiin pohtimaan tapaa, jolla järjestelmässä tapahtuville tapahtumille ja operaatiolle asetettaisiin aikaleimoja, joiden perusteella olisi mahdollista päätellä niiden tapahtumajärjestys. Tällöin järjestelmän solmujen paikallisia aikoja ei tarvitsisi enää muutella, vaan tapahtumien oikea järjestys kaikissa tilanteissa, jota tavoiteltiin varsinaisella synkronoinnilla, saavutettaisiin aikaleimojen avulla. Tällaisessa lähestymistavassa on olennaisen tärkeää, että nämä aikaleimat noudattavat kausaalista järjestystä: jos tapahtuma A tapahtuu kausaalisesti ennen tapahtumaa B, silloin aikaleima(A) < aikaleima(B). Tavallisessa elämässä kausaalisuus yleensä mielletään siten, että jokin tapahtuma johtaa toiseen. Klassinen esimerkki: ikkunaan heitetään kivi ja

se hajoaa. Tässä kivenheitto-tapahtuma tapahtuu aina ennen ikkunan hajoamista. Jos ikkuna hajoaa ennen kiven heittämistä, se ei voi johtua siitä, että siihen heitettiin kivi, eikä tapahtumilla ole kausaalista yhteyttä. Tapahtumien kausaalisella yhteydellä hajautetuissa järjestelmissä tarkoitetaan pitkälti tätä samaa, mutta järjestelmien monimutkaisen luonteen vuoksi kausaalisuus tässä yhteydessä on määriteltävä täsmällisemmin. Ensimmäisen tällaisen kausaalisen tavan järjestää tapahtumia kehitti Leslie Lamport 1970-luvulla [Lamport 1978]. Nykyään tällaisia on käytössä lähes kaikissa hajautetuissa järjestelmissä. Erityisesti pilvipalveluissa käytetään lähes kaikissa jonkinlaista tapahtumien loogista järjestämistä.

Lamportin aikaleimojen järjestäminen perustuu määritelmään a ”edeltää kausaalisesti” b:tä kahden tapahtuman a ja b välillä. Merkitään tätä operaatiota merkillä →. Näillä on tietynlainen tulkintaero riippuen siitä, tapahtuvatko ne samassa vai verkon eri prosesseissa.

Määritelmä 2.3.3.1. Operaatio → määritellään kolmella säännöllä [Lamport 1978]:

- Tapahtumat a ja b ovat saman prosessin sisällä eli käytetään samaa kelloa. Tällöin a→b, jos aikaleima(a) < aikaleima(b).
- Tapahtumat a ja b ovat eri prosesseissa. Tällöin ei enää voida käyttää kellonaikaa mittarina vaan pitää edellyttää, että tapahtumat ovat jonkinlaisessa yhteydessä keskenään. Tällä tarkoitetaan, että tapahtuma a on prosessin p1 lähettämä informaatio prosessille p2 ja tapahtuma b tarkoittaa tuon informaation vastaanottamista prosessissa p2. Tällöin sanotaan, että a→b, kun a ja b sijaitsevat eri prosesseissa.
- Operaatio on transitiivinen. Jos a→b ja b→c, tällöin pätee myös a→c. Tämä ominaisuus mahdollistaa pitkienkin tapahtumaketjujen kausaalisen järjestämisen monimutkaisessa hajautetussa järjestelmässä.

Määritelmän 2.3.3.1 tapa järjestää tapahtumat eli ”synkronoi” hajautetun järjestelmän solmut vain osittain. Vain saman prosessin tapahtumat ja ne eri prosessien tapahtumat, jotka ovat jollain tavoin yhteydessä keskenään, ovat operaation → perusteella jossain järjestyksessä. Esimerkiksi, jos prosessin p1 ja prosessin p2 kaikki tapahtumat ovat prosessin sisäisiä tapahtumia eli ne eivät koskaan kommunikoi keskenään, tapahtumat voidaan tällöin järjestää prosessin sisällä, mutta prosessin p1 tapahtumien järjestyksestä prosessin p2 tapahtumien suhteen tai päinvastoin ei voida sanoa mitään. Näin on, koska tapahtumat eivät ole missään tekemisissä keskenään, eikä kausaalista yhteyttä ole. Seuraavaksi käsitellään esimerkki tästä algoritmista.

Tarkastellaan prosesseja P1, P2 ja P3. Prosessissa P1 ovat tapahtumat A, B, C, D ja E. Prosessissa P2 ovat tapahtumat E_2, F ja G sekä prosessissa P3 tapahtumat H, I ja J. Prosessin P1 sisällä tapahtumien aikajärjestys on aakkosjärjestys A:sta E:hen eli operaatiolla → kuvattuna A→B→C→D→E. Prosessissa P2 vastaava järjestys on E_2→F→G ja P3:ssa H→I→J. Nämä ovat siis tapahtumien järjestykset prosessien sisällä. Oletetaan tämän lisäksi, että prosessit myös kommunikoivat keskenään ja osa viesteistä on viestien lähettämis- ja vastaanottotapahtumia eri prosessien välillä. Tapahtuma B prosessissa P1 on viestin lähetystapahtuma prosessille P2.

Tapahtuma F prosessissa P2 on tuon saman viestin vastaanottotapahtuma. Vastaavasti tapahtuma G prosessissa P2 on viestin lähetystapahtuma prosessille P1 ja tämä vastaanotetaan prosessin P1 tapahtumassa D. Edelleen vastaavasti tapahtuma H prosessissa P3 on viestin lähetystapahtuma prosessille P2 ja viesti vastaanotetaan tapahtumassa E_2. Vielä E prosessissa P1 on niin ikään viestin lähetystapahtuma prosessille P3 ja P3:ssa se vastaanotetaan ajallisesti tapahtumassa J. Prosessien sisäiset tapahtumat ovat siis prosesseittain seuraavassa järjestyksessä:

- Prosessi P1: A→B, B→C, C→D ja D→E eli A→B→C→D→E eli A→E
- Prosessi P2: E_2→F ja F→G eli E_2→G
- Prosessi P3: H→I ja I→J eli H→J.

Tämän lisäksi seuraavilla prosessien välisillä tapahtumilla on suora kausaalinen yhteys:

- B→F, koska tapahtuma B lähettää viestin prosessissa P1, jonka tapahtuma F ottaa vastaan prosessissa P2.
- G→E, koska tapahtuma G prosessissa P2 lähettää viestin, joka vastaanotetaan prosessissa P1 tapahtumassa E.
- H→E_2, koska tapahtuma H prosessissa P3 lähettää viestin, joka otetaan vastaan prosessissa P2 tapahtumassa E_2.
- E→J, koska tapahtuma E prosessissa P1 lähettää viestin, joka otetaan vastaan prosessissa P3 tapahtumassa J.

Näiden kausaalisten yhteyksien ja →-operaation transitiivisuuden perusteella voidaan päätellä kausaaliset yhteydet myös seuraavien tapahtumien välillä:

- H→G, koska H→F ja F→G prosessin P2 sisällä, joten transitiivisuuden perusteella H→G. Tämä sama pätee myös kaikkiin G:n jälkeen tapahtuviin tapahtumiin prosessissa P2.
- B→J, koska B→F, prosessin P2 sisällä F→G, G→E ja E→J, joten transitiivisuuden perusteella B→J.
- C→J, koska prosessin P1 sisällä C→D ja D→E sekä tämän lisäksi E→J, joten C→J.

Sen sijaan esimerkiksi tapahtumien C (prosessi P1) ja F (P2) järjestystä ei voida määrittää, koska tapahtuma C tapahtuu saman prosessin tapahtuman B jälkeen, joka on kausaalisessa yhteydessä tapahtumaan F toisessa prosessissa, mutta, koska prosessien kelloja ei ole synkronoitu eivätkä tapahtumat C ja F ole suorassa kausaalisessa yhteydessä, niiden välillä ei ole kausaalista yhteyttä.

Varsinaisessa Lamportin algoritmissa kaikille tapahtumille asetetaan numeroitu aikaleima. Numeroinnin tavoitteena on asettaa aikaleimat siten, että ne noudattavat kausaalisuutta. Lamportin algoritmin perusversio toimii seuraavasti:

- Jokainen prosessi käyttää omaa sisäistä kokonaislukulaskuriaan tapahtumille. Laskurin alkuarvo on 0.
- Aina, kun jokin muu tapahtuma kuin viestin vastaanottaminen toiselta prosessilta tapahtuu, prosessi kasvattaa sisäistä laskuriaan ja laittaa tämän laskurin arvon tapahtuman aikaleimaksi.
- Kun prosessi lähettää viestin toiselle prosessille, tämän tapahtuman aikaleima lähetetään mukana.

- Kun prosessi ottaa viestin vastaan se päivittää tämän tapahtuman aikaleimaksi arvon max(paikallinen laskuri, viestin aikaleima)+1.

Seuraavassa käydään kohta kohdalta läpi edellinen esimerkki yllä määritellyllä algoritmilla suoritettuna. Lopputuloksena kaikkien jossain yhteydessä toisiinsa olevien tapahtumien pitäisi olla kausaalisesti järjestyksessä. Alkutilanteessa, kun mitään ei ole vielä tapahtunut, kaikkien prosessien laskurien arvona on 0. Seuraavalla kierroksella prosessien ensimmäisille tapahtumille annetaan algoritmin mukaan aikaleimat seuraavasti:

- P1, aikaleima(A)=1.
- P2, aikaleima(E_2)=2.
- P3, aikaleima(H)=1.

Tapahtuma H prosessissa P3 lähettää viestin prosessille P2, joka otetaan vastaan tapahtumassa E_2. Tällöin E_2:n aikaleimaksi tulee max(0,1)+1=2, koska paikallinen laskuri on alussa arvossa 0 ja viestin mukana tullut aikaleima on 1. Tapahtumien arvoiksi voidaan prosesseittain kiinnittää seuraavat:

- P1, aikaleima(B)=2.
- P2, aikaleima(F)=3.
- P3, aikaleima(I)=2.

Tässä taas B:n aikaleima on edellisen tapahtuman aikaleima+1 eli 2, prosessin P3 tapahtumalla I samoin. Prosessissa P2 tapahtuman F aikaleimaksi tulee 3, koska se ottaa vastaan tapahtuman B viestin prosessilta P1, jonka aikaleima on 2 ja paikallinen laskuri ennen tätä oli 2, joten max(2,2)+1=3. Tämän jälkeen prosessissa P1 seuraava tapahtuma on C, jonka aikaleimaksi tulee 3. Prosessissa P2 seuraava tapahtuma on G aikaleimalla 4, koska se on ajallisesti seuraava F:stä. Tapahtuma G on viestin lähetys prosessille P1, jossa tuo viesti vastaanotetaan tapahtumassa D. Tapahtuma D saa siten aikaleimakseen max(4,4)+1=5, koska paikallinen laskurinarvo on 4 ja viestissä vastaanotettu arvo 4. Prosessissa P on vielä tapahtuma E, jonka aikaleima on paikallisen laskurin seuraava arvo eli 6. Prosessin P1 tapahtumien leimat algoritmin käsiteltyä ne ovat siis aikaleima(A)=1, aikaleima(B)=2, aikaleima(C)=3, aikaleima(D)=5 ja aikaleima(E)=6. Prosessin P2 aikaleimat ovat vastaavasti aikaleima(E_2)=1, aikaleima(F)=3 ja aikaleima(G)=4. Prosessissa P3 tapahtumien aikaleimat ovat aikaleima(H)=1, aikaleima(I)=2 ja aikaleima(J)=7. Prosessin P1 tapahtuma E on viestin lähetys prosessille P3, jonka vastaanottotapahtuma tapahtuma J on, joten J:n aikaleimaksi algoritmin sääntöjen mukaan tulee max(2,6)+1=7.

Yhteenvetona algoritmin ajamisen jälkeen tapahtumilla on prosessien sisällä seuraavat aikaleimat (→-operaatiolla kuvattuna):

- Prosessi P1: A(1)→B(2), B(2)→C(3), C(3)→D(5) ja D(5)→E(6).
- Prosessi P2: E_2(2)→F(3) ja F(3)→G(4).
- Prosessi P3: H(1)→I(2) ja I(2)→J(7).

Aikaleimat prosessien välisille tapahtumille taas ovat seuraavat:

- B(2)→F(3), prosessien P1 ja P2 välillä
- G(4)→E(6), prosessien P2 ja P1 välillä.
- H(1)→E_2(2), prosessien P3 ja P2 välillä.

- E(6)→J(7), prosessien P1 ja P3 välillä.

Tässä toteutuksessa kaikkia tapahtumia ei kuitenkaan aseteta aikaleimojen mukaan järjestykseen. Esimerkiksi tapahtuma C prosessissa P1 ja tapahtuma F prosessissa P2 eivät ole algoritmin suorituksenkaan jälkeen missään järjestyksessä keskenään. Näin on, koska C tapahtuu P1:ssä C:n jälkeen ja B→F, mutta C:llä ja F:llä ei ole mitään kausaalista yhteyttä. Tämä voidaan tulkita myös C:n ja F:n aikaleimoista, joka on molemmilla 3. Toinen tällainen tapahtumapari esimerkissä on H ja C. Lamportin määritelmän mukaan tällaiset tapahtumat ovat samanaikaisia eikä niiden välillä ole operaatiota → lainkaan määritelty. Niillä ei siis ole keskinäistä kausaalista yhteyttä kumpaankaan suuntaan.

Lamportin algoritmin muodostamat aikaleimat eivät kerro samanaikaisten tapahtumien järjestystä. Tämä nähdään yllä käsitellystä esimerkistä tapahtumista H (prosessi P3) ja C (P1). Näiden aikaleimat ovat H(1) ja C(3) eli tämä viittaisi siihen, että H tapahtuisi ennen C:tä. Nyt kuitenkin B→C eli B tapahtuu prosessissa P1 ennen C:tä ja B→F eli B tapahtuu prosessissa P1 kausaalisesti ennen tapahtumaa F prosessissa P2. Tämän lisäksi H→E_2 ja E_2→F eli H→F eli H tapahtuu kausaalisesti ennen prosessin P2 F-tapahtumaa, mutta, koska myös B tapahtuu kausaalisesti ennen F-tapahtumaa, ei sen ja H:n tapahtumajärjestyksestä voi sanoa mitään. Sama pätee myös C:n joka tulee B:n jälkeen samassa prosessissa (P1). Eli aikaleimat kertovat järjestyksen vain kausaalisessa yhteydessä olevien tapahtumien välillä. Formaalisti muotoiltuna:

Tapahtumille E1 ja E2 pätee E1→E2 => aikaleima(E1) < aikaleima(E2), mutta

aikaleima(E1) < aikaleima(E2) => { E1→E2} TAI {E1 ja E2 ovat samanaikaisia tapahtumia}.

Lamportin algoritmi on hyvä silloin, kun epäsynkronisen verkon kaikkia tapahtumia ei tarvitse saada järjestykseen. Algoritmin ongelma on kuitenkin se, ettei sitä, onko kahden eri prosessin tapahtuman välillä kausaalista suhdetta, voi päätellä pelkästään aikaleimoista, vaan tarvitaan lisätietoa siitä, ovatko nämä tapahtumat jotenkin yhteydessä keskenään. Seuraavassa alakohdassa käsitellään tapaa järjestää kaikki verkon solmut siten, että mistä tahansa kahdesta verkon solmusta voidaan tämän järjestämisen perusteella kertoa, ovatko ne samanaikaisia tai kausaalisesti yhteydessä.

2.3.4. Vektorikellot

Vektorikellojen käyttö epäsynkronisen järjestelmän solmujen synkronoimisessa käyttäen aikaleimojen on eräänlainen kaikkia tapahtumia koskeva Lamportin aikaleimamekanismin yleistys. Vektorikellojen käyttö synkronointimekanismina hajautetuissa järjestelmissä on lähes yhtä yleistä kuin Lamportin aikaleimojen. Vektorikellomekanismissa jokaisella verkon prosessilla (solmulla) on kokonaislukulaskimista muodostuva vektori, jota se ylläpitää. Oletetaan, että järjestelmässä on N prosessia. Tällöin jokaisen prosessin erikseen ylläpitämässä vektorissa on N elementtiä ja ylläpito tapahtuu seuraavasti [Baldoni and Klusch 2002]:

- Prosessi i ylläpitää vektoria $V_i[1...N]$.
- j:s elementti vektorissa $V_i[j]$ on i:n viimeisin tieto tapahtumista prosessissa j.

- Kun prosessissa i tapahtuu mitä tahansa muuta kuin viestin lähettäminen muulle prosessille tai viestin vastaanottaminen muilta prosesseilta, se vain kasvattaa oman vektorinsa i:ttä elementtiä yhdellä ja liittää tämän kyseisen tapahtuman aikaleimaksi.
- Kun prosessi i lähettää viestin toiselle prosessille, se oman i:nnen laskurinsa kasvattamisen lisäksi lähettää oman vektorinsa kokonaisuudessaan viestin mukana eli siis prosessin i koko vektori $V_i[1…N]$ menee viestin mukana.
- Ottaessaan vastaan viestiä, prosessi i ensin päivittää oman vektorinsa i:nnen elementin kasvattamalla sitä yhdellä eli $V_i[i] = V_i[i]+1$. Tämän lisäksi kaikki muut vektorin V_i elementit päivitetään valitsemalla vektorin V_i i:nnen elementin ja viestissä tulleen vektorin vastaavasta elementistä suurempi eli formaalisti $V_i[j] = \max(V_{viesti}[j], V_i[j])$ kaikilla $j \neq i$, jonka jälkeen ko. vektori liitetään kyseisen tapahtuman aikaleimaksi.

Kausaalinen riippuvuus voidaan määritellä täsmällisesti kahden tapahtuman välillä.

Määritelmä 2.3.4.1. Kausaalisen riippuvuuden määritelmä: Tarkastellaan tapahtumia VK_1 ja VK_2 ja niiden N-ulotteisia aikaleimavektoreita. Tällöin:

- tapahtumat ovat samoja eli tällöin päteeVK1=VK2, jos ja vain jos $VK_1[i] = VK_2[i]$ kaikilla i = 1…N.
- $VK_1 \leq VK_2$, jos ja vain jos $VK_1[i] \leq VK_2[i]$] kaikilla i = 1…N.
- VK_1 ja VK_2 ovat kausaalisessa suhteessa keskenään, jos $VK_1 < VK_2$. $VK_1 < VK_2$, jos ja vain jos $VK_1 \leq VK_2$ ja on olemassa sellainen elementti j, että $VK_1[j]<VK_2[j]$, missä j on jokin luku välillä i…N.
- VK_1 ja VK_2 eivät ole kausaalisessa suhteessa keskenään eli ovat samanaikaisia, jos ja vain jos NOT ($VK_1 \leq VK_2$) ja NOT ($VK_2 \leq VK_1$) eli siis vektorien leimoista ei voi päätellä tapahtumien järjestystä.

Seuraavaksi tutkitaan, miten aikaleimat päivitetään vektorikellomekanismilla Lamportin aikaleimojen käsittelystä tutussa esimerkissä.

Esimerkissä oli kolme prosessia P1, P2 ja P3. Tämän lisäksi prosesseissa oli tapahtumia tapahtumajärjestyksessä seuraavasti:

- P1: A, B, C, D ja E.
- P2: E_2, F ja G.
- P3: H, I ja J.

Prosessien väliset tapahtumat taas olivat seuraavat:

- B(P1)→F(P2)
- H(P3)→E_2(P2)
- G(P2)→D(P1)
- E(P1)→J(P3).

Alkutilanteessa kaikkien prosessien vektorien lukemat ovat (0,0,0). Prosessin P1 ensimmäisen tapahtuman A aikaleimaksi tulee (1,0,0). Tämä tapahtuma ei ollut yhteydessä mihinkään muiden

prosessien tapahtumiin, joten P1 ainoastaan päivittää omaa elementtiään vektorissaan. Prosessin P3 ensimmäinen tapahtuma H lähettää viestin prosessille P2, joka otetaan vastaan sen tapahtumassa E_2. Prosessi P3 siis laittaa tapahtuman H aikaleimaksi (0,0,1) ja lähettää viestissä P2:lle tämän vektorin. Kun P2 ottaa sen vastaan, se ensin päivittää oman elementtinsä vektorissaan kasvattamalla sitä yhdellä eli vektorin arvoksi tulee (0,1,0). Tämän jälkeen se vielä käyttää viestissä tullutta vektoria päivittääkseen oman vektorinsa. Ensimmäinen elementti sekä P2:n omassa että viestin vektorissa on 0, joten se päivittyy 0:ksi. Kolmas elementti P2:n omassa vektorissa on 0 ja viestin vektorissa 1, joten lopullinen P2:n oman vektorin arvo tapahtuman E_2 jälkeen ja E_2 aikaleima on (0,1,1). Ensimmäisten tapahtumien aikaleimat prosesseittain ovat siis P1, tapahtuma A: (1,0,0), P2, E_2: (0,1,1) ja P3, H: (0,0,1). Tämän jälkeen prosessi P1 käsittelee seuraavan tapahtuman B, jonka jälkeen sen vektorin arvoina ja tapahtuman aikaleimana ovat (2,0,0). Tämä taas vaikuttaa prosessin P2 toiseen tapahtumaan F. Tämän tapahtuman aikaleimaksi prosessissa P2 tulee (2,2,1). Näin siksi, koska ensin P2 päivittää oman elementtinsä vektorissa, joka kasvaa yhdellä eli arvo on nyt (0,2,1). Sitten päivitetään viestistä tulleen vektorin (2,0,0) arvot vertaamalla niitä edellä annettujen sääntöjen mukaan vektoriin (0,2,1), jolloin saadaan (2,2,1). Prosessin P3 seuraava tapahtuma kasvattaa vain paikallista elementtiä, joten kun toiset tapahtumat on käsitelty, tapahtumilla ovat aikaleimat P1, tapahtuma B: (2,0,0), P2, F: (2,2,1) ja P3, I: (0,0,2). Prosessin P1 kolmas tapahtuma kasvattaa vain paikallista elementtiä, joten sen vektoriksi tulee tämän jälkeen (3,0,0). Prosessin P2 kolmas tapahtuma on viestin lähetys prosessille P1, joten vektoriksi tulee (2,3,1). Prosessin P3 seuraava tapahtuma on vasta viestin vastaanotto P1:n viimeiseltä tapahtumalta, joten sen vektorin arvo pysyy edelleen samana. Kolmannen kierroksen jälkeen tapahtumien aikaleimavektorit prosesseittain ovat siis P1, tapahtuma C: (3,0,0), P2, G: (2,3,1) ja P3, I: (0,0,2). Neljäs tapahtuma P1:ssä on P2:sta tulleen viestin vastaanotto. Tämän jälkeen P1:n vektorin arvot ovat (4,3,1), koska viestissä tullut vektori oli (2,3,1) ja, kun tämä sääntöjen mukaan päivitetään, aikaleimaksi tapahtumalle D tulee tuo (4,3,1). Prosessin P1 viimeinen tapahtuma lähettää viestin prosessille P3. Ensin P1 kasvattaa vektorissa omaa laskuriaan yhdellä, jolloin tapahtuman E aikaleimaksi tulee (5,3,1) ja tämä lähetetään viestin mukana prosessille P3. Kun P3 vastaanottaa tämän kasvattaa omaa elementtiään yhdellä ja sitten päivittää viestin vektorista muut. Näin P3:n viimeisen tapahtuman aikaleimaksi tulee (5,3,3). Kun kaikki tapahtumat on käsitelty, niihin on prosesseittain liitetty seuraavat aikaleimat:

- P1: A(1,0,0), B(2,0,0), C(3,0,0), D(4,3,1) ja E(5,3,1).
- P2: E_2(0,1,1), F(2,2,1) ja G(2,3,1).
- P3: H(0,0,1), I(0,0,2) ja J(5,3,3).

Yllä olevista prosessien sisäisien tapahtumien aikaleimoista näkee suoraan, että ne noudattavat kausaalisuutta, koska yllä määritellyillä operaatiolla esim. prosessin P1 tapahtumien aikaleimoille pätee, A < B, B < C, C < D ja D < E. Prosessien välisistä tapahtumista, jotka lähettävät ja ottavat vastaan viestejä, voidaan niin ikään todeta, että ne noudattavat yllä määriteltyä kausaalisuutta. Esimerkiksi B→F, jossa B:n aikaleima on (2,0,0) ja F:n aikaleima on (2,2,1). Tässä pätee aikaleima(B) < aikaleima(F), koska B:n vektorista ei löydy yhtään elementtiä, joka olisi suurempi kuin F:n vastaava ja löytyy ainakin yksi elementti, joka on F:n vektorissa suurempi kuin B:n vastaava,

tässä tapauksessa tällaisia elementtejä on kaksi. Samalla tavoin voidaan todeta myös muiden prosessien välisten tapahtumien, joilla on kausaalinen yhteys, noudattavan kausaalisuutta tässä menetelmässä määritellyllä tavalla eli H(0,0,1) < E_2(0,1,1), G(2,3,1) < D(4,3,1) ja E(5,3,1) < J(5,3,3). Näin siis H→E_2, G→D ja E→J. Tällä menetelmällä saadaan tunnistettua myös tapahtumat. joilla ei ole kausaalista yhteyttä. Tarkastellaan esimerkkinä tapahtumia C ja F. Näistähän todettiin, ettei niillä ole kausaalista yhteyttä ja Lamportin aikaleima-algoritmi löysi niille saman aikaleiman 3 molemmille. Tässä menetelmässä C:n aikaleima on (3,0,0) ja F:n (2,2,1). Lamportin metodissa ei ole tapaa, jolla tällaiset tapahtumat voidaan leiman perusteella löytää, vaan tarvitaan leiman lisäksi erikseen tieto siitä, mitkä prosessien väliset tapahtumat kommunikoivat keskenään. Tässä C:n ja F:n kausaalinen riippumattomuus voidaan päätellä siitä, että leimat aikaleimavektoreissa menevät ristiin eli kummassakin vektorissa on vähintään yksi elementti, joka on suurempi kuin toisen vektorin vastaava elementti. Vielä parempi esimerkki on tapahtumapari H ja C. Lamportin algoritmi antaa näille aikaleimat H(1) ja C(3). Pelkkää aikaleimaa tutkimalla näyttäisi siltä, että tapahtuma H tapahtuisi ennen C:tä. Kuten aiemmin todettiin H:lla ja C:llä ei ole kuitenkaan mitään kausaalista yhteyttä, joten Lamportin aikaleimat eivät kerro tässä tapauksessa todellista tilannetta. Vektoriaikaleimat näille tapahtumille ovat vastaavasti H(0,0,1) ja C(3,0,0). Tästä nähdään heti, että leimat menevät taas ristiin, mistä voidaan päätellä, ettei tapahtumien välillä ole kausaalista yhteyttä.

Verrattuna Lamportin aikaleimamenetelmään aikaleimavektorit vaativat enemmän tilaa, koska yhden laskurin sijasta jokaista tapahtumaa kohti joudutaan ylläpitämään N kappaletta laskureita, jossa N on järjestelmän prosessien määrä. Tällöin myös laskureiden ylläpitäminen on tässä mallissa raskaampi operaatio. Tässä on kuitenkin etuna, että kausaalisen järjestyksen noudattamisen lisäksi tällä menetelmällä pystytään myös erottamaan kausaalisesti toisistaan riippumattomat eli määritelmän 2.3.4.1. mukaan samanaikaiset tapahtumat pelkästään aikaleimavektorin sisällön perusteella.

3. Viestintä hajautetuissa järjestelmissä

Viestintä ja sen luotettavuus ovat hajautettujen järjestelmien toiminnan eräitä perusedellytyksiä. Erilaisia viestintätapoja ruvettiin tutkimaan tarkemmin hajautettujen järjestelmien koon kasvaessa, jolloin tarvittiin jonkinlaista organisoitua tapaa hoitaa järjestelmän solmujen välinen viestintä. Tehokkaasti toimivan viestinnän lisäksi jokaisella solmulla tulee myös olla riittävä informaatio muiden solmujen tilasta. Eräs tehokas tapa hoitaa tämä on ns. solmujen jäsenlistat. Solmun jäsenlista on jokaisella solmulla oleva luettelo, joka sisältää tiedot kullakin ajanhetkellä järjestelmässä olevista muista toiminnassa olevista solmuista. Jäsenlistat ovat hajautetun järjestelmän toiminnan eräs tärkeä perusominaisuus. Niiden avulla voidaan solmujen kaatumisen kontrollointi toteuttaa tehokkaasti, mutta niitä käytetään myös muihin tarkoituksiin.

3.1. Ryhmäviestintä

Ryhmäviestinnän toteuttamiseksi on kehitetty lukuisia erilaisia protokollia. Yksinkertaisin lähestymistapa ryhmäviestinnän toteuttamiseksi on keskitetty malli. Keskitetyssä mallissa on yksi lähettäjäsolmu, jolla on lista vastaanottajista. Lähettäjäsolmu käy läpi kaikki vastaanottajasolmut esimerkiksi for-lauseella ja lähettää jokaiselle vuorollaan ryhmäviestinä lähetetyn viestin. Tämän mallin tunnettuja ongelmia on alhainen vikasietoisuus ja tehottomuus. Esimerkiksi, jos lähettäjäsolmu kaatuu, kun se saanut lähetettyä viestinsä vasta puolelle vastaanottajasolmuistaan, puolet verkon solmuista jää ryhmäviestinnän ulkopuolelle. Lähettäjäsolmun kuormitus on myös hyvin suuri. Tuhansien solmujen järjestelmässä solmun täytyy lähettää tuhansia viestejä, ja tämä kasvattaa myös vastaanottajasolmujen keskimääräistä viestin vastaanottoaikaa, joka on tässä mallissa lineaarinen verkon kokoon nähden. Lineaarinen eli kokoluokkaa O(n) oleva viestin vastaanottoaika ei ole kovin hyvä järjestelmissä, joissa voi olla jopa miljoonia solmuja. Näiden ongelmien ratkaisemiseksi on kehitetty puuhun perustuva malli, jota käsitellään seuraavassa alakohdassa.

3.1.1. Ryhmäviestinnän puumalli

Puumallissa verkon solmut järjestetään virittäväksi puuksi. Lähettäjäsolmu on tässä mallissa puun juurisolmu, joka lähettää kaikille seuraavassa tasossa oleville solmuille viestin. Nämä ottavat viestin vastaan, kopioivat sen ja lähettävät kopion eteenpäin omille lapsisolmuilleen. Tämä jatkuu, kunnes päästään puun alimpaan tasoon asti. Puumallissa ei siis enää ole yhtä solmua, joka hoitaisi koko viestiliikenteen, ja tämä korjaa keskitetyn mallin lähettäjäsolmun kuormitusongelman. Teoriassa myös tehokkuusongelma ratkeaa. Jos puu on tasapainossa, tällöin paras mahdollinen viestien vastaanottoaika on logaritminen verkon kokoon nähden eli siis O(log(n)). Tämä ei kuitenkaan käytännössä yleensä toteudu. Tämän mallin suuri ongelma on, että yhden solmun kaatuminen voi käytännössä lamauttaa suurenkin osan verkosta. Oletetaan, että puun juurta seuraavalla tasolla oleva solmu p_i kaatuu ennen kuin se saa kopioitua ja lähetettyä eteenpäin juurisolmulta vastaanottamansa viestin. Tällöin yksikään tämän solmun lapsisolmuista ei vastaanota mitään viestejä, kunnes solmu p_i on joko korvattu uudella solmulla tai toipunut kaatumisestaan. Ääritapauksessa, eli jos puun juurisolmulla on vain yksi lapsisolmu, käytännössä koko verkko lamaantuu yhden solmun

kaatumisesta. Koska isoissa hajautetuissa järjestelmissä solmujen kaatumisia tapahtuu koko ajan, tästä aiheutuu merkittävää tehottomuutta. Puumaisen viestintämallin perusmuodossa joudutaan koko ajan käyttämään huomattava määrä resursseja puun ylläpitämiseen. Tämä johtuu siitä, että osa puusta on jatkuvasti ryhmäviestien saavuttamattomissa, kun ko. puun osan juurisolmu on kaatuneena. Toinen ongelma on puumallin rakenteessa. Jotta tätä mallia voidaan hyödyntää, hajautetun järjestelmän solmut pitää saada järjestettyä puuksi eli käytännössä rakentaa verkon virittävä puu. Tämä ei aina ole helppoa tai välttämättä edes mahdollista.

3.1.2. Ryhmäviestinnän puumallin parannellut versiot

Puumallin perusmallia, jossa juurisolmu vain lähettää viestejä lapsisolmuilleen, jotka sitten monistavat viestiä ja välittävät sitä eteenpäin ilman mitään kontrollointia viestien perillemenosta, ei käytännössä juuri käytetä edellä lueteltujen ongelmien takia. Ongelmia ratkaisemaan on kehitetty malleja, joissa vastaanottajasolmut jotenkin ilmoittavat siitä, ovatko ne vastaanottaneet viestin vai eivät. On olemassa kaksi erilaista teoreettista mallia toteuttaa tämä. Malli, jossa käytetään negatiivista informointia, ja malli, jossa käytetään positiivista informointia. Negatiivisen informoinnin mallissa vastaanottajasolmun lähettää ns. NAK-viestin isäsolmulleen, jos se ei ole vastaanottanut ryhmäviestiä määrätyn ajan kuluessa. NAK tulee englannin kielen sanoista negative acknowledgement. Negatiivisessa informoinnissa vastaanottajasolmu siis lähettää puun juurisolmulle eli ryhmäviestin lähettäjäsolmulle ns. korjauspyynnön (eli NAK-viestin), kun se huomaa, ettei se ole vastaanottanut odottamaansa ryhmäviestiä määrätyn ajan kuluessa. Nämä korjausviestit etenevät puussa ylöspäin, ja sitä mukaa, kun ne saavuttavat ylempänä olevia solmuja, nämä solmut lähettävät alaspäin uudelleen viimeisimmän vastaanottamansa ryhmäviestin. Positiivisen informoinnin malli toimii siten, että vastaanottajasolmut lähettävät ylöspäin puussa ns. ACK-viestejä. Kuten olettaa saattaa, ACK tulee englannin kielen sanasta acknowledgement. Näitä viestejä jokainen vastaanottajasolmu lähettää säännöllisin väliajoin kohti puun juurisolmua. ACK-viesti sisältää tiedot kaikista ryhmäviesteistä, jotka ko. solmu on vastaanottanut toistaiseksi. Kutakin ACK-viestin solmua ylempänä oleva solmu tutkii ACK-viestin listasta, mitkä ryhmäviestit ovat jääneet lähettämättä, ja lähettää ne uudelleen.

Nämä kummatkin tavat on kehitetty lähinnä puumallin ryhmäviestinnän luotettavuuden lisäämiseksi. Toisin sanoen on tarvittu jokin luotettava ja riittävän nopea tapa havaita puun eri osien passivoituminen ko. osan juurisolmun kaatumisen takia. Nämä menetelmät ovat käytännössä luotettavia, mutta eivät kaikissa tilanteissa riittävän tehokkaita. Kumpikin kärsii joissain tilanteissa negatiivisten tai positiivisten viestien liian suuresta määrästä. RMTP-protokolla eli Reliable Multicast Transport Protocol on positiivista informointia eli ACK-viestejä käyttävä protokolla [Paul et al. 1997]. Tässä protokollassa puu on lisäksi jaettu useampaan alipuuhun. Ideana on jakaa puu eli käytännössä hajautettu järjestelmä useampaan osaan. Lähettäjäsolmu lähettää tässäkin mallissa ryhmäviestin kaikille vastaanottajille käyttäen globaalia ryhmäviestipuuta, mutta lähettäjäsolmu saa takaisin viestin siitä, mitkä ryhmäviestit ovat tulleet perille ja mitkä eivät, eli siis ACK-viestin ainoastaan näiden alipuiden juurisolmuilta. ACK-viestejä ei siis tule enää koko virittävän puun joka

ainoalta vastaanottajasolmulta, vaan näiden alipuiden juurisolmut toimivat ikään kuin omien alisolmujensa edustajina viestiliikenteessä. Kunkin alipuun vastaanottajasolmut lähettävät omalle juurisolmulleen ACK-viestin siitä, mitkä ryhmäviestit ne ovat vastaanottaneet ja mitä eivät. Näitä alipuun alisolmujen välittämiä ACK-viestejä juurisolmu ei enää lähetä eteenpäin lähettäjäsolmulle. Täten lähettäjäsolmu näkee tässä mallissa ainoastaan alipuiden juurisolmut ja juurisolmut näkevät alipuidensa vastaanottajasolmut. Hajauttamalla ACK-viestien käsittely tällä tavoin lähettäjäsolmun ja alipuiden juurisolmujen kesken pyritään välttämään ACK-viestien määrän räjähdysmäinen kasvu.

3.2. Jäsenlistat

Tärkeitä osia sekä ryhmäviestinnän että kaatumisilmaisinten toteutuksessa ovat jäsenlistat. Jäsenlista on hajautetun järjestelmän jokaisen solmun ylläpitämä lista muista solmuista. Tähän listaan solmu tallentaa kunkin ylläpitämänsä solmun tilan. Myös solmujen lukumäärä listassa päivittyy sitä mukaa, kun solmuja poistuu järjestelmästä tai liittyy järjestelmään tai kaatuu. Riippuen verkon rakenteesta ja listojen käyttötarkoituksesta kukin solmu pitää listaa joko kaikista tai vain tietyistä muita listan solmuista. Jäsenlistan käyttötarkoitus on ensisijaisesti se, että sen avulla järjestelmän solmut pysyvät selvillä siitä, mikä on muiden solmujen tila järjestelmässä. Luonnollisesti järjestelmässä pitää olla olemassa jokin järkevä tapa ylläpitää solmujen jäsenlistoja. Tällaista kutsutaan jäsenlistan ylläpitoprotokollaksi tai lyhyesti jäsenprotokollaksi. Pääasiassa jäsenprotokollan tulee huolehtia jäsenlistojen pysymisestä ajan tasalla. Kun järjestelmään liittyy uusi solmu, tulee jäsenprotokollan ensinnäkin havaita tämä uusi liittyminen ja informoida verkon muita solmuja tästä, jotta ne päivittäisivät omat listansa. Sama päivitys tulee myös tapahtua jonkun solmun kaatuessa tai poistuessa järjestelmästä. Vaikein ongelma luotettavasti toimivan jäsenprotokollan suunnittelussa on se, että sen pitää toimia luotettavasti järjestelmässä, jonka toiminta on järjestelmän rakenteen vuoksi jossain määrin epäluotettavaa. Näin on etenkin epäsynkronisissa hajautetuissa järjestelmissä. Kaikissa hajautetuissa järjestelmissä epäluotettavuus syntyy siitä, että kommunikointi hajautetuissa järjestelmissä solmujen välillä toimii epäluotettavasti. Se voi hukata viestejä matkalla tai niiden perillemeno voi lykkäytyä liian paljon. Epäsynkronisissa järjestelmissä tulee lisähaasteeksi vielä se, että niissä viestien kulku solmuista toiseen voi jo lähtökohtaisesti kestää mielivaltaisen kauan. Jäsenlistat ovat tärkeitä myös juoruamiseen perustuvassa viestinnässä, jota käsitellään seuraavassa alakohdassa. Jäsenlistojen käytössä viestinnän apuvälineenä on myös oma tärkeä roolinsa edellisessä luvussa käsitellyllä erilaisilla hajautetun verkon synkronointitavoilla.

3.3. Juoruprotokollat ja niiden analysointi

Juoruamiseen eli käytännössä sykkimiseen perustuva viestintä on eräs tapa parantaa puumallissa havaittuja puutteita. Hajautetun järjestelmän solmun sanotaan sykkivän, kun se lähettää säännöllisin väliajoin muille solmuille viestejä. Viestintä hajautetussa järjestelmässä voidaan myös järjestää erilaisiin sykemalleihin perustuen. Tässä kohdassa käsitellään kolme alkeellisinta sykeprotokollaa lähinnä solmun kaatumisen havaitsemisen kannalta, mutta nämä ovat myös juoruamiseen perustuvan viestinnän perusmalleja.

3.3.1. Keskitetty sykeprotokolla

Keskitetyssä sykeprotokollassa järjestelmän yksi solmu, olkoon se p_j, vastaanottaa N-solmuisen verkon kaikilta muilta N-1:tä solmulta säännöllisin väliajoin viestin, jossa on jokin järjestysnumero. Solmu p_j pitää kirjaa muilta solmuilta tulevista viesteistä, ja jos uutta sykeviestiä ei ole tullut määrätyn ajan T kuluessa, ko. solmu merkitään kaatuneeksi. Tässä siis kaikki muut verkon solmut lähettävät viestiä yhdelle solmulle, joka toimii viestien vastaanottajana. Tämä algoritmin etu on, että kaikkien viestejä lähettävien solmujen kaatuminen taatusti havaitaan. Kun mikä tahansa näistä N-1:tä solmusta kaatuu, se lopettaa viestien lähettämisen ja solmu p_j huomaa tämän ja merkitsee ajan T kuluttua ko. solmun kaatuneeksi. Eli siis täydellisyysvaatimus tulee näiden N-1 solmun osalta tyydytetyksi. Tässä algoritmissa on kuitenkin kaksi pahaa puutetta. Ensinnäkin, kaatumisilmaisimena toimivan solmun p_j kaatumista ei välttämättä havaita verkossa lainkaan. Toinen huono piirre tässä on se, että tuhansien solmujen verkossa kaatumisilmaisimeksi valittu solmu ylikuormittuu pahasti joutuessaan koko ajan käsittelemään muiden solmujen viestejä. Jonkinlainen järkevämpi tapa jakaa tuo kuorma ja varmistaa myös se, että verkon jokaisen solmun kaatuminen havaitaan kaikissa olosuhteissa, olisi siis tarpeen.

3.3.2. Rengasmainen sykeprotokolla

Rengasmaisessa sykeprotokollassa on yritetty ratkaista keskitetyn algoritmin ongelmat. Tässä versiossa kaikki solmut järjestetään renkaaksi ja jokainen solmu lähettää sykeviestin ainakin toiselle naapureistaan. Sykeviestin sisältö on sama kuin keskitetyssäkin algoritmissa. Tässä algoritmissa siis kukin solmu sekä vastaanottaa että lähettää sykeviestejä. Tämä korjaa keskitetyn version yhden solmun ylikuormittumisongelman. Tässä ei myöskään enää ole yhtä solmua p_j, jonka kaatumista ei havaita lainkaan. Tämäkään algoritmi ei kuitenkaan pysty kunnolla käsittelemään tilannetta, jossa verkon useampi solmu kaatuu samanaikaisesti tai lähes samanaikaisesti. Oletetaan, että solmun p_i molemmat naapurit renkaassa kaatuvat, ja ennen kun nämä solmut ehtivät toipua kaatumisesta tai rengas päivitetään siten, että siinä ei ole noita kaatuneita prosesseja, myös p_i kaatuu. Tällöin p_i:n kaatumista ei havaitse mikään solmu, koska sen molemmat naapurit ovat jo kaatuneet siinä vaiheessa, kun se itse kaatuu.

3.3.3. Kaikki kaikille -sykeprotokolla

Kaikki kaikille -sykeprotokollassa jokainen solmu lähettää sykeviestin verkon kaikille muille prosesseille. Muuten toimitaan kuten kahdessa edellä kuvatussakin eli, jos solmu p_i ei saa uutta viestiä solmulta p_j määrätyn ajan kuluessa, se merkitsee solmun p_j kaatuneeksi. Tämä algoritmi takaa tasaisen kuorman kaikille verkon prosesseille. Toinen ratkaiseva ominaisuus tässä on, että se tyydyttää kaatumisilmaisimen täydellisyysvaatimuksen eli jokainen kaatuminen havaitaan. Minkä tahansa verkon solmun kaatuminen havaitaan aina, jos verkossa ylipäätänsä on vielä yksikin toimiva solmu jäljellä. Tässäkin algoritmissa on kuitenkin omat puutteensa. Jos jokin solmuista on merkittävästi hitaampi vastaanottamaan ja lähettämään viestejä kuin muut, se saattaa päätyä merkitsemään kaikki tai lähes kaikki muut solmut kaatuneiksi, vaikka ne yhä olisivat toiminnassa.

Tämä tapahtuu yksinkertaisesti siksi, että se ei saa vastaanotetuksi niiltä tulevia viestejä sen ajan kuluessa, jonka jälkeen solmu tulkitaan verkossa kaatuneeksi. Tämä taas pienentää tällaisen kaatumisilmaisimen tarkkuuta eli kaikki havaitut kaatumiset eivät ole todellisia.

Todellisessa käytössä olevat juoruamiseen perustuvat viestintämallit ovat muunnelmia kaikki kaikille -sykeprotokollasta. Juoruamista viestinnässä hyväksi käyttäviä protokollia on kaksi, työnnä-protokolla ja vedä-protokolla.

Työnnä-protokollassa lähettäjäsolmu valitsee n kappaletta kohdesolmuja ja lähettää niille kopion ryhmäviestistä. Tämä tapahtuu toistuvasti jaksoittain. Kerran x:ssä sekunnissa lähettäjäsolmu valitsee y kappaletta kohdesolmuja ja lähettää niille uuden viestin. Tämä viesti korvaa vanhan viestin. Kun kohdesolmu vastaanottaa viestin, sen sanotaan ”saaneen tartunnan”. Kun kohdesolmu saa tartunnan, se alkaa itse jaksottaisesti lähettämään samaa viestiä y kappaleelle satunnaisesti valittuja kohdesolmuja.

Vedä-protokollassa jokainen solmu lähettää säännöllisin väliajoin viestin, jossa se kysyy satunnaisesti valitsemaltaan joukolta muita solmuja ”Oletko hengissä?” eli siis lähettää yksinkertaisimmillaan jonkinlaisen ping-viestin. Kun kohdesolmu saa tämän ping-viestin, se lähettää vastausviestin lähettäjäsolmulle.

3.4. Järjestys ryhmäviestinnässä

Eräs tärkeä ominaisuus ryhmäviestinnässä on viestien järjestys. Yleensä halutaan, että jokainen solmu vastaanottaa ryhmäviestinnän viestit samassa jossain määrin järkevässä järjestyksessä. On olemassa kaksi erilaista tapaa järjestää ryhmäviestinnän viestit. Nämä tavat käsitellään seuraavissa alakohdissa.

3.4.1. Fifo-järjestys

Fifo-tyylinen viestien järjestäminen [Amirhossein et al. 2011] viestinnässä on yksinkertaisin tapa järjestää viestit. Oletetaan, että on lähettäjäsolmu L ja vastaanottajasolmu V. Jokaiselle viestiparille M_1 ja M_2 tulee päteä seuraava: jos M_1 lähetetään ennen M_2:ta, viestit tulee myös vastaanottaa tässä järjestyksessä. Fifo-järjestys voidaan määritellä myös jokaisen viestin yhteenlasketun läpimenoajan avulla. Oletetaan, että viestien lähtöajat ovat viestille M_1 lähtö(M_1) ja viestille vastaavasti lähtö(M_2). Oletetaan, että erotusaika on E = lähtö(M_2) - lähtö(M_1) ja lisäksi, että E > 0. Olkoon saapuu(M) viestin M saapumisaika ja matka-aika viipyy(M) = saapuu(M) – lähtö(M). Tämä viipyy(M) on siis viestin yhteenlaskettu läpimenoaika. Tällöin, jos viipyy(M_1) – viipyy(M_2) > E, viestit ovat tulleet perille eri järjestyksessä kuin ne lähetetiin eli Fifo-järjestysprotokolla on rikkoutunut. Tämä on niin kutsuttu Fifo-järjestysprotokollan toteutumisehto. Viestin M läpimenoaika voidaan myös tarkemmin ilmaista summana viipyy(M) = viipyy*(M) + viipyy$^{'}$(M). Fifo-järjestämisen eräs ominaisuus on se, että Fifo-järjestysprotokollan rikkoutumista tapahtuu sitä useammin mitä pienempi E on, eli mitä tiheämmin viestejä lähetetään.

Seuraavassa käydään läpi tämä protokolla esimerkin avulla. Tarkastellaan neljää prosessia P1, P2, P3 ja P4. Näistä P1 lähettää kaksi ryhmäviestiä M1:1 ja M1:2 ja prosessi P3 yhden ryhmäviestin

M3:1. Prosessi P1 lähettää ensin viestin M1:1 ja sen jälkeen viestin M1:2, joten jokaisen vastaanottavan prosessin tulee vastaanottaa ensin viesti M1:1 ja vasta sitten viesti M1:2, jotta Fifo-järjestysprotokollan toteutumisehto toteutuisi. Sen sijaan prosessin P1 ja prosessin P3 lähettämien viestien keskinäiselle järjestykselle tämä protokolla ei aseta mitään vaatimuksia. Ei siis ole merkitystä sillä, missä järjestyksessä kukin prosessi vastaanottaa viestit M1:1 ja M3:1 tai M1:2 ja M3:1, mutta, jos mallissa olisi myös prosessilla P3 kaksi ryhmäviestiä M3:1 ja M3:2, tulisi kaikkien muiden prosessien vastaanottaa nämäkin viestit täsmälleen siinä järjestyksessä, kun ne on lähetetty, jotta protokolla toteutuisi.

Tämän menetelmän ongelma epäsynkronisissa järjestelmissä on se, että se ei ota huomioon solmujen välistä aikaeroa. Jos solmut L ja V ovat eri prosesseissa, noissa laskelmissa pitää jälleen ottaa huomioon solmujen kellojen eri ajoista johtuva virhemarginaali. Tämän vuoksi onkin kehitetty viestien kausaaliseen järjestykseen perustuva tapa järjestää viestit, jota käsitellään seuraavassa alakohdassa.

3.4.2. Kausaalinen järjestys

Kausaalinen tapa järjestää viestit [Raynal et al. 1991] on eräänlainen Fifo-järjestämisen yleistys. Tässä menetelmässä olennainen merkitys on Lamportin ”tapahtuu ennen”-suhteella (määritelmä 2.3.3.1) kahden tapahtuman välillä. Merkitään tällaista operaatiota, kuten aikaisemminkin, merkinnällä →. Kausaalisessa järjestämisessä viestien järjestys on määritelty operaation → perusteella. Tapahtumat, jotka voidaan laittaa keskenään järjestykseen operaation → perusteella, ovat kausaalisessa yhteydessä keskenään. Kausaalisen järjestyksen perusvaatimus on seuraava: keskenään kausaalisessa yhteydessä olevat ryhmäviestit, jotka on lähetetty määrätyssä kausaalisessa järjestyksessä, pitää myös ottaa vastaan samassa kausaalisessa järjestyksessä. Tämä voidaan kuvata muodollisemmin näin: Olkoon viestin lähettäjäsolmu L ja vastaanottajasolmu V. Solmu L lähettää viestin M solmulle V. Olkoon M’ mikä tahansa solmun L lähettämä ryhmäviesti, joka lähetetään viestin M jälkeen eli M→M’. Tällöin vastaanottajasolmun V tulee myös vastaanottaa nuo viestit samassa kausaalisessa järjestyksessä eli määritelmän M→M’ pitää päteä myös solmussa V ja edelleen kaikissa muissa verkon solmuissa, jotka vastaanottavat tuon viestin.

Yksinkertaisin tapa toteuttaa viestien kausaalinen järjestäminen ryhmäviestinnässä on käyttää vektoreita. Jokainen ryhmäviestejä vastaanottava prosessi ylläpitää vektoria, jossa on jokaista lähettäjäprosessia vastaava järjestysluku. Siis N:n prosessin järjestelmässä jokainen prosessi ylläpitää vektoria Pi[1…N], jonka alkiot alustetaan nolliksi. Vektoriin tallennetaan kunkin lähettäjäprosessin viesteissään lähettämät järjestysluvut, toisin sanoen Pi[j] on viimeisin järjestysluku, jonka prosessi Pi on vastaanottanut prosessilta Pj. Vektorien päivityssääntö on määritelty seuraavassa määritelmässä.

Määritelmä 3.4.2.1. Vektorien päivittämisen säännöt:

- Viestin lähettäminen. Kun prosessi Pi lähettää ryhmäviestin, se kasvattaa oman vektorinsa i:ttä elementtiä yhdellä eli P[i] = P[i] + 1. Tämän jälkeen se liittää uuden vektorin kokonaisuudessaan mukaan eteenpäin lähettämäänsä ryhmäviestiin.
- Viestin vastaanottaminen. Prosessi Pj vastaanottaa ryhmäviestin prosessilta Pi, jossa on prosessin Pi viestivektori viestin lähetyshetkellä, eli viestin vektori M[1…N] = P[1…N]. Prosessi Pj tallentaa vektorin M muistiinsa ja lähettää viestin eteenpäin vasta, kun seuraavat kaksi ehtoa toteutuvat:
    1. Tämä viesti on seuraava viesti, jota Pj odottaa Pi:ltä, eli M[i] = Pj[i]+1.
    2. Prosessi Pj on vastaanottanut kaikki muut ryhmäviestit kaikkialta verkosta, jotka kausaalisesti edeltävät viestiä M eli nämä viestit → M. Täsmällisemmin ilmaistuna, kaikille k ≠ i, jossa k = 1…N ja i on prosessin Pi indeksi ryhmäviestien vektoreissa, pätee M[k] ≤ Pj[k].

Näiden kahden ehdon toteuduttua Pj päivittää oman vektorinsa viestin mukaiseksi eli asettaa Pj[i] = M[i].

Käydään yllä olevan protokollan toteutus läpi esimerkin avulla. Tarkastellaan neljää prosessia P1, P2, P3 ja P4. Tällöin jokainen prosessi ylläpitää neljän alkion vektoria. Olkoot nämä vektorit PV1, PV2, PV3 ja PV4. Vektorien alkuarvot, kun mitään viestiä ei vielä ole lähetetty, ovat (0,0,0,0). Prosessi P1 lähettää ryhmäviestin M1:1, P2 lähettää viestin M2:1 ja P4 viestin M4:1. Prosessien sisällä viestien lähetys- ja vastaanottotapahtumien järjestys on seuraava. Prosessi P1 lähettää ensin oman viestinsä ja ottaa sitten vastaan viestin M2:1 ja tämän jälkeen viestin M4:1. Merkitään viestien vastaanottamista merkinnällä V(viesti) ja lähettämistä merkinnällä L(viesti). Tällöin prosessin P1 sisällä viestitapahtumien kausaalinen järjestys on seuraava: L(M1:1) →V(M2:1) →V(M4:1). Prosessi P2 ottaa ensin vastaan prosessin P1 ryhmäviestin, lähettää sen jälkeen omansa, ja ottaa sitten vastaan ryhmäviestin prosessilta P4 eli järjestys on V(M1:1) →L(M2:1) →V(M4:1). Prosessi P3 ottaa ensin vastaan viestin M2:1 sitten viestin M4:1 ja lopuksi viestin M1:1 eli järjestys on V(M2:1) →V(4:1) →V(M1:1). Prosessi P4 vastaanottaa ensin ryhmäviestin prosessilta P1, lähettää sen jälkeen oman ryhmäviestinsä ja vastaanottaa lopuksi viestin prosessilta P2 eli V(M1:1) →L(M4:1) →V(M2:1). Selvyyden vuoksi kerrataan vielä viestien lähetys- ja vastaanottotapahtumien järjestys prosessien sisällä prosesseittain:

- Prosessi P1: L(M1:1) →V(M2:1) →V(M4:1).
- Prosessi P2: V(M1:1) →L(M2:1) →V(M4:1).
- Prosessi P3: V(M2:1) →V(4:1) →V(M1:1).
- Prosessi P4: V(M1:1) →L(M4:1) →V(M2:1).

Aluksi siis prosessi P1 lähettää oman ryhmäviestinsä eteenpäin ja liittää viestiin mukaan vektorin MV1(1,0,0,0). Prosessi P2 ottaa vastaan ryhmäviestin M1:1 ja toteaa, että tämä on seuraava ryhmäviesti, jota se odottaa prosessilta 1, koska sen omassa vektorissa ensimmäinen elementti 0 ja viestin vektorissa (MV1:1) vastaava elementti on 1. Tämän lisäksi se toteaa, että kaikki muut

elementit sen omassa vektorissa ovat nollia eli samoja kuin viestin vektorissa eli myös määritelmän 3.4.2.1 toinen ehto eli ns. kausaalisuusehto toteutuu, joten P2 voi käsitellä vastaanottamansa ryhmäviestin. Prosessi P4 toteaa aivan samalla tavalla, että sen P1:ltä vastaanottama viesti on ensinnäkin seuraava sen P1:ltä odottama viesti ja kaikki muut viestit, jotka se on vastaanottanut, edeltävät tätä viestiä kausaalisesti, joten se voi käsitellä viestin. Tämän jälkeen prosessi P2 lähettää oman ryhmäviestinsä ja kasvattaa samalla oman viestivektorinsa toista elementtiä yhdellä eli vektorin arvo on PV2 = (1,1,0,0) sekä lähettää tämän vektorin ryhmäviestin aikaleimana. Ottaessaan vastaan P2:n ryhmäviestiä M2:1 prosessi P1 toteaa, että tämä on sen P2:lta odottama viesti, koska viestin vektorin toinen elementti on 1 ja P1:n oman vektorin vastaava elementti on 0. Edelleen, koska kaikki muut vektoreiden elementit ovat identtisiä (PV1(1,0,0,0) ja MV2:1(1,1,0,0)), myös kausaalisuusehto toteutuu ja P1 voi välittää viestin eteenpäin sekä päivittää oman vektorinsa vastaavan elementin samaksi kuin viestin vektorissa. Ottaessaan vastaan viestiä M2:1 prosessi P3 toteaa, että tämä on seuraava viesti, jota se P1:ltä odottaa, koska jälleen vektorin MV2:1(1,1,0,0) toinen elementti on yhden suurempi kuin prosessin P3 oman vektorin (PV3(0,0,0,0)) toinen elementti. Sen sijaan kausaalisuusehto ei tällä kertaa toteudukaan. Viestin vektorin ensimmäinen elementti on 1, kun taas P3:n oman vektorin vastaava elementti on 0 eli prosessi P3 ei ole vielä ottanut vastaan kaikkia viestejä, jotka sen tulisi ottaa vastaan, jotta kausaalisuusehto toteutuisi. Tämä johtuu siitä, että prosessi P2 otti vastaan P1:n ryhmäviestin M1:1 ennen kuin se lähetti omansa (M2:1) eli kausaalisesti M1:1 edeltää viestiä M2:1. Nyt kuitenkin prosessi P3 ottaa ensin vastaan viestin M2:1 ja vasta sen jälkeen viestin M1:1. Tämän vuoksi P3 laittaa viestin M2:1 odottamaan ja käsittelee sen vasta, kun se on ottanut vastaan myös viestin M1:1. Käsiteltyään prosessilta P1 tulleen viestin P4 päivittää vektoriinsa arvoon PV4(1,0,0,0). Tämän jälkeen se lähettää oman ryhmäviestinsä, päivittää vektorinsa arvoksi (1,0,0,1) ja lähettää tämän viestin mukana. Kun prosessi P1 ottaa sen vastaan, se toteaa jälleen, että se ensinnäkin on viesti, jota se odottaa P4:ltä, koska viestin vektori on MV4:1(1,0,0,1) ja P1:n oma vektori vastaanottohetkellä on PV1(1,1,0,0). Myös kausaalisuusehto toteutuu, sillä P1:n vektorissa PV1(1,1,0,0) ei ole neljännen elementin lisäksi yhtään elementtiä, joka olisi pienempi kuin vastaava elementti viestin mukana tulleessa vektorissa MV4:1(1,0,0,1) eli P1 voi käsitellä viestin. Vastaavalla tavalla myös prosessi P2 toteaa tämän viestin käsittelykelpoiseksi. Sen sijaan prosessi P3 laittaa tämänkin odottamaan samasta syystä kuin viestin M2:1, eli se ei ole vieläkään ottanut vastaan viestiä M1:1, joka edeltää kausaalisesti näitä molempia. Kun P3 viimein ottaa vastaan viestin M1:1, se toteaa, että tämä on sen prosessilta P1 odottama viesti, koska sen oma vektori on tässä vaiheessa vielä PV3(0,0,0,0) ja viestin M1:1 vektori on MV1:1(1,0,0,0). Näin ollen myös kausaalisuusehto toteutuu ja P3 käsittelee viestin M1:1. Samalla se myös käsittelee puskuroimansa viestit M2:1 ja M4:1 ja, kun nämä kaikki on käsitelty, prosessin P3 vektorissa on arvot PV3(1,1,0,1).

4. Kaatumisilmaisimet

Kaatumisilmaisinten toiminta ja toteutus riippuvat paljon siitä, toimivatko ne synkronisessa vai asynkronisessa järjestelmässä. Synkronisessa järjestelmässä kaatumisen havaitsemien voidaan kehittää täysin luotettava tapa, mutta asynkronisessa järjestelmässä tämä ei järjestelmän rakenteen vuoksi ole mahdollista. Asynkronisessa järjestelmässä joudutaan tyytymään siihen, että kaatumisilmaisin toimii jollain todennäköisyydellä oikein. Tätä kutsutaan probabilistiseksi tai realistiseksi kaatumisen havaitsemiseksi. Valitettavasti lähes kaikki nykyään käytössä olevat hajautetut järjestelmät ovat rakenteeltaan asynkronisia, joten myös kaatumisilmaisimet joudutaan rakentamaan vain niin täydellisiksi kuin mahdollista.

4.1. Kaatumisilmaisimen tärkeät ominaisuudet

Kuten jo aiemmin on todettu, kaatumisilmaisimia tarvitaan, koska järjestelmän solujen kaatumista silloin tällöin ei voi estää. Lisäksi kaatumisia tapahtuu sitä useammin, mitä enemmän järjestelmässä on solmuja. Kaatumisilmaisimen kaksi tärkeää ominaisuutta ovat [Raynal 2005]:

- täydellisyys eli, kun jokin solmu kaatuu, jokainen pystyssä oleva solmu saa lopulta tiedon tämän solmun kaatumisesta. Tätä kutsutaan joskus myös vahvaksi täydellisyydeksi.
- tarkkuus eli jokainen havaittu kaatuminen on aito. Havainnot tehdään siis vain todella tapahtuneista kaatumisista.

Kaatumisilmaisinten toteutuksen kannalta ikävä asia on, että näiden kahden ominaisuuden toteuttaminen täydellisesti on mahdotonta. Tämä johtuu suoraan tutkielmassa aiemmin käsitellystä yksimielisyysongelmasta. Kaatumisilmaisinten toteutuksissa joudutaan siis aina tinkimään jommastakummasta ominaisuudesta. Koska näistä tärkeämpi ominaisuus on täydellisyys, yleensä ilmaisimen tarkkuus on se, josta tingitään. Täydellisyys on tärkeämpi ominaisuus siksi, että kaatumisilmaisimen tärkein ominaisuus on se, että se havaitsee mahdollisimman suuren osan kaikista kaatumisista, mieluiten kaikki. Tämän vuoksi kaatumisilmaisimet käytännössä toteutetaan niin, että niiden täydellisyys pyritään takaamaan sataprosenttisesti tai lähes sataprosenttisesti ja tarkkuus toteutetaan niin hyvin kuin se kussakin tapauksessa on mahdollista toteuttaa. Kaksi muuta kaatumisilmaisimen toivottavaa ominaisuutta ovat:

- nopeus. Tämä tarkoittaa käytännössä sitä, että aika hetkestä, jolloin kaatuminen tapahtuu siihen hetkeen, jolloin jokin toiminnassa oleva solmu havaitsee kaatumisen, tulisi olla mahdollisimman lyhyt.
- skaalautuvuus. Tämä tarkoittaa, että järjestelmän koon eli solmumäärän kasvaessa järjestelmään ei synny toimintaa merkittävästi hidastavia pullonkauloja. Tämä pyritään takaamaan suunnittelemalla kaatumisilmaisin siten, että se jakaa kuorman mahdollisimman tasaisesti järjestelmän solmujen kesken.

4.2. Sykkeeseen perustuvat kaatumisilmaisimet

Sykkeeseen tai juoruamiseen perustuvan kaatumisilmaisimen perusperiaate on yksinkertainen. Tässä mallissahan jokainen järjestelmän solmu lähettää säännöllisin väliajoin laskurilla varustetun sykeviestin kaikille muille solmuille. Toiminnassa oleva solmu s_j havaitsee tai merkitsee solmun s_i kaatuneeksi, kun se ei enää ole ottanut tältä vastaan sykeviestejä riittävän monen peräkkäisen sykejakson aikana. Sykeprotokolla takaa käytännössä sen, että kaatunut solmu aina havaitaan määrätyn ajan kuluessa jonkin toimivan solmun toimesta, sillä solmu, joka kaatuu, lopettaa myös viestien välittämisen. Näin yksinkertaisella tavalla toimivan kaatumisilmaisimen ongelmana on tehottomuus ja viestien suuri määrä. Tällainen kaatumisilmaisin tuottaa verkkoon neliöllistä suuruusluokkaa olevan ylimääräisen kuorman seuraavan määritelmän mukaan:

Määritelmä 4.2.1.
Oletetaan, että T on yksittäisen solmun kaatumisen havaitsemiseen vaadittu aika ja N verkon solmujen lukumäärä. Tällöin ylimääräisen kuorman suuruusluokka on $O(n^2/T)$ viestiä aikayksikössä.

Määritelmän 4.2.1 mukaisen kuorman keventämiseksi on kehitetty protokolla, jossa sykeviestit välitetään käyttäen jonkinlaista juoruamiseen perustuvaa tapaa. Tässä protokolassa solmut eivät enää lähetäkään sykeviestiään kaikille, vaan vain osalle muista solmuista [Van Renesse et al. 1998] eli seuraavan määritelmän mukaan:

Määritelmä 4.2.2.
Järjestelmän jokainen solmu lähettää ajanjakson T_g välein satunnaisesti valitulle joukolle kohdesolmuja N-kokoisen listan, jossa on viimeisimmät sykelaskurien arvot, jotka solmu on vastaanottanut. Koska juoruviestit etenevät logaritmisesti, uuden sykeviestin odotettavissa oleva aika, jona se saavuttaa kohdesolmun on kokoluokkaa $O(\log(N)*T_g)$ ajanyksikköä. Määritelmän 4.2.2 mukaista protokollaa käsitellään seuraavassa alakohdassa.

4.3. Juorutyylinen jäsenlistoihin perustuva kaatumisilmaisimen perusmalli

Juoruiluun perustuva kaatumisilmaisin on muunnelma kaikki kaikille -sykeprotokollasta, joka on kuitenkin tehokkaampi kuin puhtaasti kaikki kaikille -sykeprotokollaan perustuva kaatumisilmaisin olisi [Van Renesse et al. 1998]. Tällä kaatumisilmaisimen toteutustavalla on pyritty ratkaisemaan yksinkertaisten sykeprotokollien tarkkuusongelma. Tämä on myös tehokkaampi kuin pelkkään kaikki kaikille -sykeprotokolaan perustuva kaatumisilmaisin. Juoruamisen tehokkuudesta johtuen tämän kaatumisilmaisimen verkolle aiheuttama kuorma on myös huomattavasti kevyempi kuin kaikki kaikille -sykeprotokollassa. Seuraavassa on kuvattu juoruamiseen perustuvan kaatumisilmaisimen perusmalli. Jokainen solmu verkossa ylläpitää omaa jäsenlistaa muista solmuista. Tässä listassa on id-numero, jota käytetään kaatumisen tunnistamisessa. Tätä kutsutaan sykelaskuriksi. Sykelaskurin lisäksi kukin solmu ylläpitää omaan paikalliseen kelloon perustuvaa aikaleimaa, jota päivitetään aina, kun sykelaskuria päivitetään sekä päivitettävän jäsensolmun osoite.

Kaatumisilmaisin toimii juoruperiaatteella siten, että kukin jäsensolmu lähettää säännöllisin väliajoin satunnaisesti valitsemilleen n:lle muulle solmulle oman jäsenlistansa. Kun solmu vastaanottaa jäsenlistan, se vertaa sitä omaan jäsenlistaansa. Se tutkii saamastaan jäsenlistasta jokaisen solmun kohdalta, onko sen omassa jäsenlistassa oleva ko. solmun sykelaskurin lukema pienempi kuin vastaanotetussa jäsenlistassa. Jos lukema on pienempi, solmu päivittää sen omaan jäsenlistaansa. Näitä listoja kukin solmu siis päivittää jatkuvasti vastaanottamistaan viesteistä. Kaatumisilmaisimen parametriksi asetetaan aika T_f. Tämä on niin sanottu kaatumisen odotusaika. Kun solmu havaitsee omasta jäsenlistastaan, että jonkin solmun sykelaskuri ei ole päivittynyt ajan T_f kuluessa, se merkitsee ko. solmun kaatuneeksi. Tämän jälkeen solmu odottaa ajan T_c ennen kuin se poistaa kaatuneeksi merkityn solmun listastaan. Mihin tarvitaan tätä T_c aikaa? Jos sitä ei ole, on mahdollista, että kaatunut solmu, joka pitäisi poistaa listasta, ei koskaan häviä sieltä.

Perustellaan tämä esimerkin avulla. Tarkastellaan järjestelmän solmuja S1 ja S2. Solmu S1 lähettää sykeviestinsä järjestelmän solmuille S2, S3 ja S4, solmun S2 viestit menevät solmuille S1, S3 ja S4. Oletetaan vielä, että T_f on 24 sekuntia. Oletetaan lisäksi, että S1 ja S2 taulukot ovat kuvassa 1 esitettyä muotoa. Kun solmu S2 vastaanottaa solmun S1 jäsenlistan, se päivittää solmut S1 ja S3, koska niiden sykelaskurin lukema on vastaanotetussa listassa suurempi ja tuloksena on kuvassa 2 esitetty taulukko. Tässä listassa päivitettyjen solmujen aikaleimaksi tulee solmun S2 paikallinen leima, joka on 75. Tämän lisäksi S2 merkitsee solmun S3 kaatuneeksi, koska sen viimeisin päivitysaika S2:n listassa oli 50 ja senhetkinen paikallinen aikaleima on 75, joten aika T_f, joka on 24 sekuntia, on mennyt umpeen. Oletetaan nyt, että odotusaika T_c solmun kaatuneeksi toteamisen jälkeen ei ole käytössä. Tällöin S2 poistaa listastaan solmua S3 koskevan rivin eli toteaa solmun kaatuneeksi eli S2:n taulukko on nyt kuvassa 3 esitettyä muotoa. Kuitenkin solmun 1 taulukossa tämä rivi säilyy vielä, koska siellä solmun S3 aikaleima on 55 ja senhetkinen paikallinen aikaleima sen verran pienempi, että, kun T_f oli 24 sekuntia, se ei tässä tapauksessa ole vielä umpeutunut.

S1	10120	66
S2	10103	62
S3	10098	55
S4	10111	65

S1	10118	64
S2	10110	62
S3	10090	50
S4	10111	65

Kuva 1. Solmujen S1 ja S2 jäsenlistat alkutilanteessa. Oikealla solmu S1 ja vasemmalla solmu S2.

Algoritmin mukaan solmu S1 lähettää jälleen solmulle S2 uuden listan. Tässä listassa on kuitenkin mukana myös solmu S3, joten solmu S2 lisää sen uudelleen omaan listaansa omalla paikallisella aikaleimallaan eli taulukon sisältö on nyt kuvassa 4 esitettyä muotoa. Tämä voi johtaa käytännössä siihen, että solmujen aikaleimojen asynkronisuuden aiheuttamasta epätarkkuudesta johtuen solmun S3 kaatumista ei koskaan havaita. Lopputulos olisi sama myös silloin, jos tuo uudelleen solmun S3 rivin sisältävä taulukko tulisi joltain muulta solmulta kuin S1. Tällaisen tilanteen estämiseksi tarvitaan aika T_c, joka jokaisen solmun pitää odottaa, ennen kuin se voi poistaa kaatuneeksi havaitsemansa solmun jäsenlistastaan. Aika T_c on valittava niin, että todennäköisyys, että se korjaa paikallisten aikaleimojen aiheuttaman poikkeaman mahdollisimman täydellisesti, on suurin mahdollinen.

S1	10120	75
S2	10110	62
S3	10098	75
S4	10111	65

Kuva 2. Solmun S2 taulukko sen vastaanotettua solmun S2 jäsenlistan.

S1	10120	75
S2	10110	62
S4	10111	65

Kuva 3. Solmun S2 taulukko, kun solmua S3 koskeva rivi on siitä poistettu.

S1	10120	75
S2	10110	62
S3	10098	78
S4	10111	65

Kuva 4. Solmun S2 taulukko, kun se on saanut solmulta S1 uuden jäsenlistan.

4.4. Tiedon levittämiseen perustuva kaatumisen havaitseminen

Sykkeeseen perustuvien kaatumisilmaisinten huono skaalautuvuus on luonut tarpeen kehittää ilmaisintyyppi, jonka verkolle muodostama ylimääräinen kuorma pysyy vakiona tai lähes vakiona yksittäisen solmun kaatumisen havaitsemisen nopeuden tai kaatumisten havaitsemisen tarkkuuden kärsimättä tästä liiaksi. Tämän lisäksi edellisessä alakohdassa käsitellyn protokollan toinen ongelma on solmujen jäsenlistojen mahdollisimman tehokas ylläpitäminen. Jäsenlistojen tehokas ylläpito on tärkeää, koska se vaikuttaa suoraan varsinaisen kaatumisen havaitsemisen tehokkuuteen. Erityisesti tämä ei saisi hidastua solujen määrän kasvaessa. Tämä taas on ongelmallista pelkästään sykkeeseen

perustuvissa kaatumisilmaisimissa, koska solmujen määrän kasvaessa jäsenlistan koko vääjäämättä kasvaa. Tiedon levittämiseen perustuvassa kaatumisten havaitsemisessa käytetään juoruamiseen perustuvaa viestintää vielä enemmän hyväksi kuin edellisessä alakohdassa käsitellyssä perusprotokollassa. Lisäksi kantavana ajatuksena on käyttää eri komponentteja kaatumisen havaitsemiseen ja jäsenlistojen ylläpitämiseen kuin viestien levittämiseen verkossa. Tässä mallissa jokainen solmu lähettää kaikuviestejä joukolle satunnaisesti valitsemiaan solmuja.

4.5. Esimerkki tiedon levittämiseen perustuvasta kaatumisilmaisimesta

Eräs tiedon levittämiseen perustuva kaatumisilmaisin on ns. Swim-kaatumisilmaisin. Swim-kaatumisilmaisimen nimi tulee englannin kielen sanoista Scalable Weekly consistent Infection style Membership Protocol eli suomeksi skaalautuva heikosti yhdenmukainen tartuttamistyylinen jäsenprotokolla [Abhinandan et al. 2002]. Tämä protokolla sisältää seuraavat kaksi erillistä komponenttia:

1. Varsinainen kaatumisilmaisinkomponentti. Tämä havaitsee jäsensolmujen kaatumisen ja pitää yllä solmujen jäsenlistoja.
2. Informaation levityskomponentti. Nimensä mukaisesti tämä huolehtii tiedon levittämisestä solmuista, jotka ovat joko kaatuneet, lähteneet järjestelmästä tai liittyneet siihen uudestaan.

Jäsenprotokolla perustuu ns. kaikuviestien lähettämiseen järjestelmän solmulta toiselle. Jokainen solmu lähettää satunnaisesti valitsemalleen solmulle säännöllisin väliajoin viestin.

Seuraavaksi kuvataan Swim-ilmaisimen perusalgoritmin toiminta. Ajan T_p kuluessa eli yhden ns. protokollaperiodin kuluessa tapahtuu seuraavaa: Satunnainen solmu p_i lähettää satunnaisesti vallitsemalleen solmulle p_j kaikuviestin. Kun solmu p_j vastaanottaa kaikuviestin, se lähettää takaisin vastausviestin tuohon kaikuviestiin. Jos p_i saa takaisin vastausviestin, se toteaa, että solmu p_j on hengissä eikä kiinnitä tähän huomiota enää ko. protokollaperiodin aikana. Jos vastausviestiä ei tule, p_i valitsee satunnaisesti K solmua ja lähettää näille ns. epäsuoran kaikuviestin. Saatuaan tämän viestin nämä K solmua lähettävät kukin solmulle p_j suoran kaikuviestin ja saatuaan tähän vastauksen, ne välittävät solmulta p_j saamansa hyväksymisviestin solmulle p_i. Jos p_i ei saa suoraa hyväksymisviestiä p_j:ltä eikä yhtään epäsuoraa hyväksymisviestiä, se merkitsee p_j:n kaatuneeksi. Toisin kuin kaikki kaikille -sykeprotokollaan perustuvassa kaatumisen havaitsemisessa, tässä solmun p_i lähettämien viestien määrä on vakio (jokin satunnainen lukumäärä K) eikä siis riipu verkon koosta.

Tämän kaatumisilmaisimen tehokkuutta analysoidaan seuraavassa. Kaatumisen havaitsemisen todennäköisyys on tässä mallissa sitä lähempänä vakioarvoa, mitä suurempi verkon koko on. Seuraava yksinkertainen skenaario kertoo tarkemmin, miksi näin on: Oletetaan aluksi, että yksi solmu on kaatunut ja tämä solmu on kaikkien muiden solmujen jäsenlistalla ja jokainen niistä poimii sattumanvaraisesti yhden kaikuviestin. Tällöin todennäköisyys, että tämä solmu valitaan kaikuviestin kohteeksi minkä tahansa solmun p_i toimesta on $\frac{1}{N}$. Todennäköisyys, että sitä ei valita solmun p_i toimesta on $1 - \frac{1}{N}$. Todennäköisyys, että mikään verkon muista solmuista ei valitse kaatunutta solmua

kaikuviestin kohteeksi on $(1 - \frac{1}{N})^{N-1}$, joten todennäköisyys, että ainakin yksi solmu lähettää kaikuviestin tälle kaatuneelle solmulle on $1 - (1 - \frac{1}{N})^{N-1}$. Kun N lähestyy ääretöntä, lähestyy lausekkeen $(1 - \frac{1}{N})^{N-1}$ arvo vakioarvoa e^{-1}, jossa e on eksponenttifunktion kantaluku eli ns. Napierin luku. Näin siis verkon koon kasvaessa yksittäisen kaatumisen havaitsemisen todennäköisyys lähestyy vakioarvoa $1 - e^{-1}$. Yleisemmässä tapauksessa todennäköisyys, että jokin mielivaltainen solmu tulee valituksi kaikuviestin kohteeksi, on $1 - (1 - Q_f * \frac{1}{N})^{N-1}$, jossa N on edelleen solmujen määrä verkossa ja Q_f toiminnassa olevien solmujen määrä. Kuten yllä käsitellyssä yksinkertaisemmassa tapauksessa, edellä esitetyn lausekkeen raja-arvoksi, kun N lähestyy ääretöntä, tulee $1 - e^{-Qf}$. Edellä esitetyn seurauksena saadaan yläraja odotusajalle, joka kuluu mielivaltaisen solmun kaatumisesta siihen, kun jokin verkon toiminnassa oleva solmu sen viimeistään havaitsee. Tämä aika on $T_p * 1/(1 - e^{-Qf})$, jossa T_p on protokollaperiodin pituus. Swim-kaatumisilmaisimen perusmallissa on kuitenkin edelleen jotain puutteita. Ongelmia voi syntyä esimerkiksi silloin, kun jotkin verkon solmuista ovat merkittävästi hitaampia kuin toiset. Tällöin teoreettisesti hyvä kaatumisilmaisimen tarkkuus voi kärsiä, kun kaikuviestit ja niiden vastaukset viipyvät liian kauan matkalla ja aiheuttavat toiminnassa olevien hitaiden solmujen merkitsemisen kaatuneiksi. Tämän vuoksi on kehitelty Swim-ilmaisimen paranneltu versio, jossa on mukana ns. epäilymekanismi, jota esitellään seuraavassa.

Swim-kaatumisilmaisimen paranneltu versio toimii muuten kuten perusversio, mutta tämän lisäksi protokollaperiodin kuluessa kerätään talteen tietoa aiemmin havaituista solmujen kaatumisista. Kun jonkin solmun kaatuminen havaitaan, siihen liittyvät kaikuviestit, epäsuorat kaikuviestit, vastausviestit ja epäsuorat vastausviestit kerätään talteen. Tämä tehdään satunnaisesti valitulle osalle havaituista kaatumisista. Kun solmut protokollaperiodin aikana vastaanottavat viestejä, ne toimivat kuten perusversiossakin, mutta tämän lisäksi ne myös tutkivat, mitä on aiemmista kaatumishavainnoista kerätty talteen ja päivittävät näillä tiedoilla jäsenlistaansa. Tässä kaatumisilmaisimessa on ns. epäilyvaihe perusversion normaalin protokollaperiodin lisäksi. Kun järjestelmän jokin solmu ei enää vastaa suoraan eikä epäsuorasti saamiinsa kaikuviesteihin, sitä ei tässä algoritmissa merkittäkään heti kaatuneeksi, vaan se merkitään ensin mahdollisesti kaatuneeksi ennen kuin se tulkitaan kaatuneeksi.

Solmu p_i käyttää tilakonetta tarkkailemalleen prosessille p_j. Alussa ja oletusarvoisesti tämän tilakoneen tila prosessin p_j kohdalla on ”elossa”. Kun solmu p_i ei enää saa solmulta p_j suoraa eikä epäsuoraa hyväksymisviestiä, se siirtää solmun p_j tilakoneessa tilaan ”epäilyksenalainen” ja lähettää järjestelmän muille solmuille viestin ”p_j epäilyksenalainen”. Jos määrätyn ajan T_f aikana solmu p_j lähettää p_i:lle hyväksymisviestin tai epäsuoran hyväksymisviestin tai kaikuviestin tai joku muu verkon solmu lähettää viestin, että p_j on elossa, p_i siirtää p_j:n takaisin tilaan ”elossa” tilakoneessaan. Jos mitään näistä viesteistä ei ajan T_f kuluessa tule, p_i merkitsee p_j:n kaatuneeksi ja lähettää muille solmuille viestin ”p_j kaatunut”. Tässä mallissa oleellista on huomata, että solmu p_j voi siirtyä edellä kuvatulla tavalla monta kertaa protokollaperiodin aikana tilasta ”elossa” tilaan ”epäilyksenalainen” ja päinvastoin. Aina, kun lähetetään viesti verkon muille solmuille, siitä, että solmu p_j on siirretty tilasta ”elossa” tilaan ”epäilyksenalainen” tai päinvastoin, tuohon viestiin sisältyy ns.

inkarnaatiotunnus. Jokaisella solmulla on siis oma inkarnaatiotunnuksensa, jota vain kyseinen solmu voi muuttaa. Inkarnaatiotunnuksen tarkoitus on, että sillä erotetaan useaan kertaan tilasta ”elossa” tilaan ”epäilyksenalainen” siirtyneiden solmujen eri siirtymiskerrat toisistaan. Alussa solmun inkarnaatiotunnus on 0. Esimerkiksi, kun solmu p_j vastaanottaa viestin ”epäilyksenalainen” itsestään, ja se on edelleen toiminnassa, se kasvattaa tuota inkarnaatiotunnusta yhdellä, ja ryhtyy lähettämään muille solmuille uuden inkarnaatiotunnuksen sisältävää viestiä. Kun muut solmut saavat tämän ”elossa”-viestin, jolla on korkeampi inkarnaatiotunnus kuin solmulla p_j aiemmin oli, ne tietävät, että ko. solmu onkin edelleen elossa, ja päivittävät sen omiin jäsenlistoihinsa. Sen lisäksi, että korkeampi inkarnaatiotunnus aina syrjäyttää matalamman, myös samoilla inkarnaatiotunnuksilla on oma sisäinen järjestyksensä. Esimerkiksi solmua p_j koskeva viesti ”epäilyksenalainen” samalla inkarnaatiotunnuksella kuin ”elossa”-viesti syrjäyttää tuon ”elossa”-viestin ja kaikki solmut, jotka viestin saavat, merkitsevät solmun p_j tämän jälkeen tilaan ”epäilyksenalainen”. Viesti ”kaatunut” syrjäyttää aina kaikki muut samalla inkarnaatiotunnuksella tulleet muut viestit.

Tämä mekanismi vähentää merkittävästi virheellisten kaatumisen havaitsemisten määrää, mutta sekään ei kokonaan poista niitä. Epäilymekanismista huolimatta on edelleen mahdollista, että jokin solmu on niin hidas, ettei se ehdi reagoida kaikuviesteihin siinäkään ajassa, jolloin se on epäilyksenalaisena jonkin muun solmun listalla. Myös perusversion takaama vahva täydellisyysominaisuus pysyy edelleen voimassa. Esimerkiksi, jos solmu p_i kaatuu ja solmu p_j on solmun p_i epäilyksenalaisten solmujen listalla, tämä pidentää solmun p_i kaatumisen havaitsemisaikaa, mutta lopulta solmun p_i kaatuminen havaitaan. Swim-kaatumisilmaisimen perusversiokin takaa sen, että mielivaltaisen solmun p_i kaatuminen lopulta havaitaan verkon jokaisen toimivan solmun taholta. Sillä, missä ajassa tämä tapahtuu, ei kuitenkaan ole algoritmin perusversiossa mitään teoreettista ylärajaa.

4.6. Paras kaatumisilmaisin

Tärkeimmät kaatumisilmaisimen ominaisuudet ovat täydellisyys, nopeus, tarkkuus ja skaalautuvuus. Teoreettisesti täydellisessä kaatumisilmaisimessa sekä täydellisyys että tarkkuus ovat sataprosenttisia, toisin sanoen, tällaisessa kaatumisilmaisimessa kaikki toiminnassa olevat solmut havaitsevat aina jokaisen kaatumisen ja jokainen havaittu kaatuminen on todella tapahtunut eli virheellisiä havaintoja ei tehdä. Johtuen epäsynkronisen hajautetun järjestelmän rakenteesta tällaista ei kaatumisilmaisinta ei pystytä toteuttamaan. Lisäksi ihanteellisessa kaatumisilmaisimessa pitäisi kuorma pystyä jakamaan mahdollisimman tasaisesti verkon solmujen kesken, kaatumiset tulisi havaita mahdollisimman nopeasti ja kaatumisilmaisimen viestiliikenteen verkolle aiheuttama kuorma tulisi saada mahdollisimman pieneksi. Yksi tärkeimmistä ominaisuuksista kaatumisilmaisinten toiminnassa on kuitenkin viestien mahdollisimman tehokas välittäminen solmulta solmulle. Tämä on erityisen tärkeää tiedon levittämiseen perustuvissa kaatumisilmaisimissa, kuten Swim-kaatumisilmaisin. Tässä tehokkaimmaksi tavaksi on osoittautunut juoruamiseen perustuva tapa levittää viestejä verkossa. Kuten jo luvussa 3 kerrottiin, tätä kommunikointitapaa kuvataan myös siten, että se on kuin huhun levittäminen tiiviissä ihmisyhteisössä. Toinen luonnehdinta juoruamiseen

perustuvasta viestinnästä on, että se on kuin jonkin tarttuvan taudin leviäminen tiiviissä ihmisyhteisössä. Viestien levittämisen tehokkuudesta tällä tavoin on laadittu tarkka matemaattinen malli [Abhinandan et al. 2002]. Tätä on alun perin tutkittu tartunnan leviämisenä ihmisyhteisössä, mutta informaation levittäminen hajautetun järjestelmän solmuissa tällä tavoin tartunnanomaisesti perustuu samaan malliin. Oletetaan, että on olemassa N-solmuinen järjestelmä, jossa on alun perin yksi tartunnan saanut solmu. Hajautettujen solmujen viestinnässä tämä tarkoittaa sitä, että tuo yksi solmu on se, joka ensimmäiseksi ryhtyy viestejä lähettämään. Tällöin voidaan määritellä, montako jäsensolmua on saanut tartunnan eli hajautetussa järjestelmän tapauksessa vastaanottanut viestejä ajan t funktiona. Olkoon x tartunnan saaneiden solmujen määrä ajanhetkellä t, N solmujen kokonaismäärä ja β tartuntanopeus aikayksikköä kohti. Tällöin tartunnan saaneiden määrä ajan funktiona on

$$\frac{dx}{dt} = \beta * x *(N - x) \qquad (1).$$

Tätä voidaan soveltaa juoruamisperustaiseen viestintään siten, että ajan yksikkönä käytetään yhtä protokollaperiodia ja tartuntanopeus on tartunnan todennäköisyys minkä tahansa kahden solmun välillä, joista toinen saanut tartunnan ja toinen ei. Puhuttaessa tartunnan saamisesta tarkoitetaan viestinnässä mitä tahansa kahta solmua, jotka ovat yhteydessä keskenään. Kahden solmun välisen kontaktin todennäköisyys voidaan laskea seuraavasti: Tarkastellaan kahta solmua p_i ja p_j. Todennäköisyys, että p_j poimii p_i:n lähettämän viestin, on $\frac{1}{N}$. Todennäköisyys, että näin ei käy on $1 - \frac{1}{N}$. Todennäköisyys, että kumpikaan ei poimi toistensa lähettämiä viestijä on $(1 - \frac{1}{N})^2$. Todennäköisyys, että jompikumpi poimii toistensa lähettämän viestin, eli solmut ovat yhteydessä keskenään, on siis $[1 – (1 - \frac{1}{N})^2] = (\frac{2}{N} – 1/N^2)$. Tämä sovellettuna lausekkeeseen (1) antaa tuloksen $x = N/(1 + (n -1) * e^{-(2 – 1/n)t}$. Tällainen viestin levittäminen verkossa on siis eksponentiaalisen nopeaa. Tällä hetkellä olemassa olevista kaatumisilmaisimista todennäköisesti paras ja lähimpänä teoreettisen kaatumisilmaisimen ihannetyyppiä on yllä kuvatun kaltainen paranneltu Swim-kaatumisilmaisin epäilymekanismeineen.

5. Yhteenveto

Solmujen kaatumisen havaitseminen ja yleensä kaatumisen kontrollointi on toteutettavissa helposti tai vaikeasti riippuen siitä, onko ko. järjestelmä synkroninen vai epäsynkroninen. Täysin synkronisessakaan järjestelmässä kaatumisen kontrollointia ei saada toimimaan teoreettisesti täydellisesti. Tämä voi johtua pelkästään jo siitä, ettei viestin kulkemisen viemää aikaa solmusta toiseen lopulta pystytä ennustamaan täysin tarkkaan edes synkronisessa verkossa. Viestintähän on monen muun hajautettuihin järjestelmiin liittyvän toiminnon ohella myös kaatumisilmaisinten kunnollisen toiminnan perusta. Viestintä on tärkeimpiä toimintoja hajautetuissa järjestelmissä ylipäätään siksi, että se on kaiken informaation kulun perusta ko. järjestelmissä. Lähinnä tämän vuoksi viestintää on tutkittu paljon. Nykyään hyvin paljon tutkittu viestinnän on muoto tämän tutkielmankin 3. luvussa käsitelty ns. juoruamiseen perustuva viestintä. Tämän keskeisenä ideanahan on, että viestien levittämiseen osallistuu järjestelmän jokainen solmu. Tämä on suoraan verrattavissa juorun leviämiseen ihmisjoukossa. Aivan kuin ihmisjoukossa leviävän juorun tapauksessa, kun jokin solmu lähettää viestin liikkeelle, jokainen vastaanottava solmu lähettää sen edelleen muille solmuille. Juoruamiseen perustuvan viestintätavan suuri etu hajautetuissa järjestelmissä on, että se on hyvin tehokasta. Viestien leviämisen vaativuus on yleensä luokkaa O(log(N)), jossa N on järjestelmän solmujen määrä, eli siis leviämisen tehokkuus verkon solmujen määrään nähden on logaritminen. Viestinnän toteuttaminen on hoidettava tässäkin mallissa niin, että solmujen kuormitus jakautuisi mahdollisimman tasaisesti. Toinen asia, josta täytyy huolehtia, on, ettei järjestelmä, jonkin solmun kaatumisen tuloksena jakaudu kahteen tai useampaan osaan, jotka eivät enää ole yhteydessä toisiinsa. Pahin tapaus on, että yhden solmun kaatumisen vuoksi koko järjestelmä on käytännössä viestien saavuttamattomissa. Tämä ongelma voi tulla vastaan yksinkertaisimmassa puumalliin perustuvassa viestinnässä. Jos puun juurta seuraavalla tasolla oleva solmu kaatuu, ennen kuin se ottaa vastaan juurisolmun viestin, eivät puun muutkaan solmut vastaanota mitään viestejä, jos puun juurta seuraavalla tasolla oleva solmu on tämän tason ainoa solmu. Tällöin viestiliikenne puun juurisolmun ja muun verkon välillä on käytännössä poikki. Yksinkertaiseen puumalliin perustuvaa viestintätapaa ei tämän ja viestien kuormituksen epätasaisuuden vuoksi juuri käytetäkään käytännön järjestelmissä. Käytännön toteutukset yleensä perustuvat puumallin paranneltuun versioon, jossa kontrolloimalla viestien perillemenoa pyritään estämään järjestelmän osittuminen muutamaan keskenään kommunikoimattomaan osaan.

Juoruamiseen perustuvassa viestinnässä on sopivan viestintämallin lisäksi tärkeää, millaista viestien käsittelyprotokollaa käytetään ja missä järjestyksessä viestejä käsitellään. Juoruamiseen perustuvan viestinnän kolme perusprotokollaa ovat keskitetty sykeprotokolla, rengasmainen sykeprotokolla ja kaikki kaikille -sykeprotokolla. Näistä kaikki kaikille -sykeprotokolla on käytännössä se, johon varsinaiset todellisissa järjestelmissä olevat toteutukset perustuvat. Keskitetyn sykeprotokollan pahin ongelma on, että viestien vastaanottajana toimiva solmu voi ylikuormittua pahasti varsinkin suurissa tuhansien tai miljoonien solmujen järjestelmissä. Rengasmaisessa

sykeprotokollassa taas rengasmallin vierekkäisten solmujen kaatumiset saattavat jäädä havaitsematta. Viestien järjestys on viestintämallien lisäksi toinen olennainen osa hajautetun järjestelmän viestinnässä. Viestien käsittelyjärjestys on tärkeää siksi, että sillä on oma vaikutuksensa verkon eheyteen. Järjestelmän toiminnan kannalta voi olla hyvinkin tärkeää, että viestit vastaanotetaan ja käsitellään siinä järjestyksessä kuin ne lähetettiinkin. Lähetettävät peräkkäiset viestit voivat esimerkiksi liittyä loogisesti toisiinsa siten, että niitä ei ole järkevää vastaanottaa muussa kuin niiden lähetysjärjestyksessä. Viestien järjestyksen käsittelyssä kaksi tärkeintä menetelmää ovat fifo-tyylinen järjestäminen ja kausaalinen järjestäminen. Näistä kausaalinen järjestäminen on optimaalisempi ja yleisesti käytetty. Fifo-tyylisen viestien järjestämisen suurin ongelma on, että siinä ei huomioida solmujen välistä aikaeroa, joten pelkästään fifo-tyylinen järjestäminen ei välttämättä toimi kovin hyvin epäsynkronisissa järjestelmissä. Kausaalinen järjestäminen on fifo-tyylisen järjestämisen yleistys, joka perustuu Lamportin ”tapahtuu ennen”-suhteeseen kahden tapahtuman välillä (määritelmä 2.3.3.1). Tämän avulla voidaan määrittää viestien järjestys luotettavasti myös solmujen välillä.

Edellä käsitellyt asiat ovat olennaisimpia osia optimaalisesti toimivissa kaatumisilmaisimissa. Käytännön järjestelmissä toimiva ja optimaalinen kaatumisilmaisin tarkoittaa toimivaa ja optimaalista kaatumisilmaisinta epäsynkronisessa järjestelmässä. Näin suuressa määrin epäsynkronisuus on käytännön toteutuksissa nykyään edustettuna. Kaatumisilmaisimien kaksi tärkeintä ominaisuutta ovat täydellisyys ja skaalautuvuus. Täydellisyys eli, että jokainen kaatuminen havaitaan, on ehkä tärkein ominaisuus, koska se takaa, että kaatumisilmaisin tekee sen, mihin se on suunniteltu, eli havaitsee kaatumiset. Käytännön toteutuksissa kaatumisilmaisimen täydellisyys pyritäänkin takaamaan mahdollisimman hyvin jopa muiden ominaisuuksien kustannuksella. Skaalautuvuus on tärkeää siksi, että kaatumisilmaisimen pitää toimia yhtä hyvin sekä pienissä että suurissa järjestelmissä. Tätä ominaisuutta tarvitaan nykypäivän valtavissa pilvipalvelujärjestelmissä yhä enemmän. Kaatumisilmaisimia on kahta tyyppiä: sykkimiseen ja tiedon levittämiseen perustuvat kaatumisilmaisimet. Sykkimiseen perustuvien kaatumisilmaisinten perustyyppi on kaikki kaikille -sykeprotokollaan perustuva kaatumisilmaisin. Tällaisen kaatumisilmaisimen puutteina ovat kuitenkin suuri viestien määrä ja kaatumisilmaisimen tuottama neliöllistä suuruusluokkaa oleva ylimääräinen kuorma (määritelmä 4.2.1). Tiedon levittämiseen perustuva kaatumisilmaisintyyppi kehitettiin ratkaisemaan sykeilmaisimissa esiintyvät puutteet ja ongelmat. Tiedon levittämiseen perustuvan kaatumisilmaisintyypin tärkein toteutus on tutkielman luvussa 4 kuvattu Swim-kaatumisilmaisin. Swim-kaatumisilmaisimen perusalgoritmissa on pyritty saamaan kaatumisen havaitsemisen todennäköisyys mahdollisimman lähelle jotain vakioarvoa ja tässä on myös onnistuttu. Lisäksi sykkeeseen perustuvia kaatumisilmaisimia vaivaava viestien suuri määrä on myös tässä mallissa korjaantunut. Kaatumisilmaisimen perusmalli ei kuitenkaan osaa käsitellä hyvin esimerkiksi sellaista tilannetta, jossa jotkin verkon solmuista ovat hitaampia kuin toiset. Tällainen tilanne voi johtaa ilmaisimen tarkkuuden heikkenemiseen, kun viestit viipyvät matkalla niin kauan, että kaatumisilmaisin merkitsee toiminnassa olevan hitaan solmun kaatuneeksi. Swim-ilmaisimen parannellussa mallissa on tätä korjaamassa ns. epäilymekanismi. Tämän lisäksi parannellussa

algoritmissa kerätään talteen aikaisemmin kaatuneisiin solmuihin liittyvät viestit. Viestien talteen keräämisellä pyritään varmistamaan, että muita hitaammin toimivia solmuja ei suotta merkitä kaatuneiksi.

Viiteluettelo

[Abhinandan et al. 2002] Abhinandan Das, Indranil Gupta, and Ashish Motivala. Swim: Scalable weakly-consistent infection-style process group membership protocol. In *Proceedings of the International Conference on Dependable Systems and Networks, 2002. DSN 2002*, pp. 303-312. IEEE, 2002.

[Amirhossein et al. 2011] Amirhossein Malekpour, Antonio Carzaniga, Giovanni Toffetti Carughi, Probabilistic fifo ordering in publish/subscribe networks. In *Proceedings of the 10th IEEE International Symposium on Network Computing and Applications (NCA)*. IEEE, 2011.

[Arjomandi et al. 1983] Arjomandi, Eshrat, Michael J. Fischer, and Nancy A. Lynch. Efficiency of synchronous versus asynchronous distributed systems. *Journal of the ACM (JACM)* 30.3 (1983): 449-456.

[Baldoni and Klusch 2002] Baldoni, Roberto, and Matthias Klusch. Fundamentals of distributed computing: A practical tour of vector clock systems. *IEEE Distributed Systems Online* 2 (2002)

[Cristian 1989] Cristian, Flaviu. Probabilistic clock synchronization. *Distributed Computing*, 3.3 (1989): 146-158.

[Fischer et al. 1985] Fischer, Michael J., Nancy Ann Lynch, and Michael S. Paterson.. Impossibility of distributed consensus with one faulty process. *Journal of the ACM* 32.2 (1985): 374-382.

[Gilbert and Lynch 2012] Gilbert, Seth, and Nancy Ann Lynch. *Perspectives on the CAP Theorem.* IEEE, 2012.

[Gubta et al. 2001] Gupta, Indranil, Tushar D. Chandra, and Germán S. Goldszmidt. On scalable and efficient distributed failure detectors. *In Proceedings of the Twentieth Annual ACM Symposium on Principles of _Distributed Computing*. ACM, 2001.

[Lamport 1978] Lamport, Leslie. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM* 21.7 (1978): 558-565.

[Lamport 2001] Lamport, Leslie. Paxos made simple. *ACM Sigact News* 32.4 (2001): 18-25.

[Lamport 2005] Lamport, Leslie. Generalized consensus and Paxos. *Technical Report MSR-TR-2005-33*, Microsoft Research, 2005.

[Mell and Grance 2011] Mell, Peter, and Tim Grance. The NIST definition of cloud computing. *Communications of the ACM* 53.6 (2011): 20-23.

[Mills 1992] Mills, David L. Network Time Protocol (Version 3) specification, implementation and analysis.

[Paul et al. 1997] Paul, Sanjoy, Krishan K. Sabnani, John C.-H. Lin, and Supratik Bhattacharyya. Reliable multicast transport protocol (RMTP). *IEE Journal on Selected Areas in Communications,* 15.3 (1997): 407-421.

[Raynal et al. 1991] Raynal, Michel, André Schiper, and Sam Toueg. The causal ordering abstraction and a simple way to implement it. *Information Processing Letters* 39.6 (1991): 343-350.

[Raynal 2005] Raynal, Michel. A short introduction to failure detectors for asynchronous distributed systems. *ACM SIGACT News* 36.1 (2005): 53-70.

[Rodrigues and Druschel 2010] Rodrigues, Rodrigo, and Peter Druschel. Peer-to-peer systems. *Communications of the ACM* 53.10 (2010): 72-82.

[Van Renesse et al. 1998] Van Renesse, Robbert, Yaron Minsky, and Mark Hayden. A gossip-style failure detection service. *In Proceedings of Middleware'98*, pp. 55-70. Springer London, 1998.