Antti Krats

PROSESSIEN VÄLINEN KOMMUNIKOINTI SUORITUSKYKYÄ VAATIVISSA JÄRJESTELMISSÄ

Tietotekniikan pro gradu -tutkielma

Ohjelmistotekniikan linja

5. elokuuta 2007

Tekijä: Antti Kalevi Krats

Yhteystiedot: antti.krats@gmail.com

Työn nimi: Prosessien välinen kommunikointi suorituskykyä vaativissa järjestelmissä

Title in English: Interprocess communication in high performance systems

Työ: Tietotekniikan pro gradu -tutkielma

Sivumäärä: 76

Tiivistelmä: Tutkielmassa tutkitaan prosessien välisiä kommunikointimenetelmiä ja niihin liittyviä tekniikoita, kuten säikeiden synkronisointia ja muistin konsistenssimalleja. Tutkielman pääpaino on jaetun muistin kautta tapahtuvassa kommunikoinnissa. Empiirisessä osassa vertaillaan jaetun muistin kautta tapahtuvaa kommunikointia ja viestinvälitysarkkitehtuuriin perustuvaa menetelmää. Tutkielma näyttää testien avulla, että jaetun muistin kautta tapahtuva kommunikointi on riittävän tehokasta, jotta ohjelmat voidaan jakaa pienempiin prosesseihin.

English abstract: This thesis presents the most usual methods to implement interprocess communication (IPC). Thesis also includes techniques essential for IPC such as thread synchronizing and memory consistency models. The emphasis is on the implementation of the shared memory IPC method. Two IPC methods are compared in the empirical section. The thesis shows with several test scenarios that the IPC implemented by shared memory is adequate to divide process into separate processes without performance degrading.

Avainsanat: Prosessien välinen kommunikointi, jaettu muisti, kontentio

Keywords: Interprocess communication (IPC), shared memory, contention

Sisältö

Kuvat

1 Johdanto

Prosessien välinen kommunikointi on tänä päivänä käymässä yhä tärkeämmäksi menetelmäksi etenkin laajoissa järjestelmissä. Ohjelmien laajentuessa ja käydessä monimutkaisemmiksi on mielekästä pilkkoa ohjelmalogiikkaa pienemmiksi moduuleiksi, jotta ohjelman rakenne selkeytyisi ja olisi helpommin ohjelmoijan ymmärrettävissä. Ohjelman pilkkomisella on hidastava vaikutus ohjelman toimintaan, sillä prosessien välinen kommunikointi on aina hitaampaa verrattuna siihen, että logiikka on yhdessä prosessissa. Tämä tutkielma esittelee jaetun muistin kautta tapahtuvan prosessien välisen kommunikointimenetelmän, joka on riittävän tehokas, jotta ohjelman pilkkominen on kannattavaa tehdä.

Jaetun muistin kautta tapahtuvassa kommunikoinnissa data on jaettu eri ohjelmien kesken. Kun ohjelmat näkevät saman muistialueen, dataa ei tarvitse kopioida prosessilta toiselle. Jotta jaettu data pysyisi eheänä, on säikeiden synkronisointi hyvin tärkeää. Tutkielmassa käydään läpi tekniikoita ja synkronisointimenetelmiä, joiden avulla prosessien synkronisointi saadaan toteutettua. Synkronisointimenetelmän on oltava tehokas, jotta siitä ei tule hidastavaa tekijää prosessien väliseen kommunikointiin. Monesti heikosti toteutetut jaetun muistin kautta tapahtuvaan kommunikointiin liittyvät menetelmät menettävät suorituskykyään kontention muodossa.

Tutkielman empiirisessä osassa esitellään yksinkertainen jaetun muistin kautta tapahtuva kommunikointimenetelmä, joka on toteutettu Windows NT käyttöjärjestelmän päälle. Menetelmän suorituskykyarvoja verrataan viestinvälitysarkkitehtuuria käyttävään sokettiliikennöintiin. Tuloksissa tullaan huomaamaan merkittäviä eroja kommunikointimenetelmien välillä.

Luvussa 2 käydään läpi synkronisointimenetelmiä, eri prosessien välisiä kommunikointimenetelmiä ja näihin molempiin liittyviä ongelmia. Luvussa käydään läpi tekniikoita, joita on välttämätöntä käyttää kommunikointimenetelmien kanssa. Luvussa 3 paneudutaan jaetun muistin kautta tapahtuvaan kommunikointiin sekä käyttöjärjestelmien muisti- ja prosessoriarkkitehtuureihin. Luvussa 4 esitellään empiirinen koe, jonka avulla saadaan mittaustuloksia jaetun muistin kautta tapahtuvasta prosessien välisestä kommunikoinnista. Ver-

tailtavana menetelmänä käytetään sokettipohjaista menetelmää. Luvussa 5 pohditaan empiirisen osan tuloksia, niiden luotettavuutta, sekä koko pro gradu -tutkielman aihealuetta. Luku 6 on tutkielman yhteenveto.

2 Prosessien välinen kommunikointi

Prosessien välisellä kommunikoinnilla tarkoitetaan ohjelmien välistä kommunikointia. Tapoja kommunikoinnin rakentamiseen on vuosien saatossa kehitelty monenlaisia. Yksinkertaisimmat menetelmät käyttävät hyväkseen tiedostoja, leikepöytää tai esim. soketteja. Monimutkaisempia menetelmiä ovat puolestaan DCOM, nimetyt putket, jaettu muisti ja CORBA. Menetelmien tarkoituksena on siirtää tietoa prosessilta toiselle, jotta ohjelmalogiikka voitaisiin pilkkoa eri prosesseihin.

Tutkielman aihealueeseen liittyy käsitteitä ja ongelmia, joita käydään läpi seuraavissa alaluvuissa. Perusasioiden jälkeen tullaan käsittelemään nykykäyttöjärjestelmissä olevia prosessien välisiä kommunikointimenetelmiä.

2.1 Transaktiot

Transaktiolla tarkoitetaan joukkoa tapahtumia, jotka käsitetään atomisena toiminnallisuuden yksikkönä. Transaktio on voimakas sovelluskehittäjän työkalu, jolla rinnakkaiset sovellukset voivat muokata yhteistä tietoa turvallisesti, ilman monimutkaisia sovellukseen kehitettäviä synkronisointimenetelmiä.

2.1.1 ACID-malli

Transaktio koostuu neljästä nk. ACID-ominaisuudesta [THa83]. Lyhenne tulee sanoista *atomisuus* (eng. atomicity), *konsistenttius* (engl. consistency), *eristettävyys* (engl. isolation) ja *kestävyys* (engl. durability). Jokainen näistä ominaisuuksista lujittaa transaktiotapahtuman eri osa-alueita.

Atomisuudella tarkoitetaan sellaista ominaisuutta, missä transaktio suoritetaan kokonaisuudessaan tai sitten mitään ei suoriteta [THa83]. Jos transaktiota suoritettaessa tapahtuu virhe, peruutetaan kaikki tehdyt tapahtumat. Onnistuneen transaktion loppuunsuoritusta kutsutaan *vahvistamiseksi* (engl. commit) ja keskeyttämistä *perumiseksi* (engl. rollback).

Konsistenttius-ominaisuus varmistaa, että myös transaktion jälkeen resurssit ovat konsistentteja [THa83].

Eristettävyydellä tarkoitetaan sitä, että transaktion suorituksen aikana tapahtuvat muutokset eivät näy muille transaktioille ennen transaktion vahvistamista [THa83]. Jos transaktio perutaan, niin tilanne pysyy muille samana – aivan kuin transaktiota ei olisi koskaan suoritettukaan.

Kestävyys-ominaisuudella vaaditaan, että transaktion vahvistamisen jälkeen transaktion aikana tehdyt muutokset ovat pysyviä [THa83]. Muutokset säilyvät tallessa tai ovat palauttamiskelpoisessa muodossa, vaikka järjestelmässä tulisi vastaan virhetilanne.

2.1.2 Transaktion osapuolet

Transaktioon liittyy eri osapuolia, jotka hallinnoivat transaktio-tapahtumaa. Seuraavaksi esiteltävät määritelmät ovat otettu Object Management Groupin tuottamasta dokumentista Transaction Service Specification [OMG03].

Transaktion aloittajaksi (engl. transaction originator) kutsutaan osapuolta, joka määrittää transaktion alkamisen ja päättymisen. Normaalisti transaktion aloittaja on asiakassovellus.

Transaktion aloittaja voi suorittaa palvelupyyntöjä transaktion alkamis- ja päättämishetken välillä *ei-transaktionaalisille* (engl. non-transactional objects)*, transaktionaalisille* (engl. transactional objects) ja *perumiskykyisille olioille* (engl. recoverable objects).

Ei-transaktionaalinen olio on olio, joka ei liity varsinaiseen transaktioon. Tämä voi olla sovelluksessa vaikkapa lokaali muuttuja, johon tallennetaan väliaikaisesti tietoa. Transaktionaalinen olio puolestaan liittyy vahvasti transaktioon. Transaktion aloittajalla on pääsy transaktionaaliseen olioon vain transaktion alkamis- ja päättämishetken välillä. Perumiskykyinen olio on transaktionaalinen olio, joka pystyy perumaan tekemänsä muutokset siinä tapauksessa, että transaktio perutaan.

Transaktionaalisien olioiden muodostamaa joukkoa ympäristöineen kutsutaan *transaktionaaliseksi palvelimeksi* (engl. transactional server). Transaktionaalista palvelinta, joka sisältää perumiskykyisiä olioita, nimitetään *perumiskykyiseksi palvelimeksi* (engl. recoverable server).

Oliota, jota välitetään transaktion aikana osapuolten välillä, kutsutaan *transaktiokontekstiksi*. Sen avulla identifioidaan transaktio, jonka kontekstissa palvelupyyntöjä tulee suorittaa.

Transaktiohallinnoija (engl. transaction manager) tarjoaa rajapinnat transaktiokontekstin käsittelemiseen. Transaktion aloittaja aloittaa transaktion transaktionhallinnoijan kautta ja sitä kautta myös transaktionaaliset oliot pääsevät käsiksi transaktiokontekstiin. Jokaisella transaktiopalvelimella on oma transaktionhallinnoija.

Transaktiot sopivat hyvin prosessien väliseen kommunikointiin. Tietokoneohjelmat ovat taipuvaisia toimimaan huonosti, johtuen ihmisen tekemistä virheistä tai jo pelkästään siitä syystä, että ohjelmat ovat nykypäivinä todella suuria ja monimutkaisia. Transaktiot, ja etenkin niiden eristettävyys-ominaisuus, pitävät huolen kahden prosessin välisen kommunikoinnin epäonnistumisen vaikutuksista prosesseille. Esimerkiksi, jos siirtoväylä prosessien välillä katkeaa, transaktiot palauttavat prosessit ennen transaktioita olleeseen tilaan.

2.2 Synkronisointi prosessien välillä

Synkronisointia prosessien välillä tarvitaan silloin, jos prosesseilla on jotain tekemistä toistensa kanssa tai ne muokkaavat samaa resurssia tai jos esimerkiksi prosessien suoritusjärjestys on ennalta määrätty.

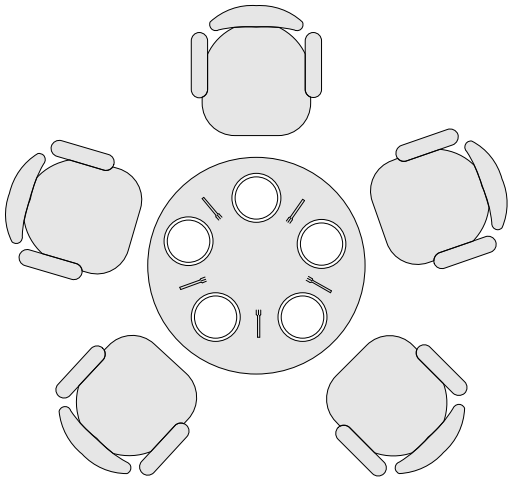
E. W. Dijkstra kirjoittaa *Cooperating sequential processes* -artikkelissa [EDi65b] prosessien välisestä synkronisoinnin ongelmasta seuraavasti: Samanaikaisesti suorittavien prosessien välisiin synkronisointeihin liittyy kommunikointiongelma ja jaetun datan pääsyongelma. Jos proseduurit ovat suunniteltu muokkaamaan jaettua dataa, ei rinnakkaisten prosessien pidä kutsua näitä proseduureja samaan aikaan, jotta jaettu data pysyy eheänä. Suojaa-

malla nämä proseduurit *kriittisellä alueella* (engl. critical section), varmistetaan, että vain yksi proseduuri suoritetaan kerrallaan.

Kommunikointiongelmassa puolestaan voidaan joutua tilanteeseen, jossa prosessi on valmiina käsittelemään informaatiota, jota ei ole vielä olemassa, koska toisen prosessin pitää ensiksi se tuottaa [EDi65b]. Tässä tapauksessa nämä proseduurit on synkronisoitava, jotta toinen prosessi ei voi aloittaa prosessointia, ennen kuin ensimmäinen prosessi on tuottanut datan. Tätä ongelmaa kutsutaan *tuottaja - kuluttaja ongelmaksi* (engl. producer - consumer problem) [DRu75].

2.2.1 Ruokailevien filosofien ongelma

Tunnetuin esimerkki säikeitten välisten synkronisoinnin ongelmakohdista tiedemaailmassa, on E. W. Dijkstran esittelemä *ruokailevien filosofien ongelma* (engl. Dining Philosophers Problem) [EDi71]. Esimerkissä tarkastellaan viittä filosofia, joiden tehtäviin kuuluvat ajatteleminen ja syöminen. Syöminen tapahtuu pyöreässä pöydässä (kuva 1), jossa jokaiselle filosofille on oma lautanen ja yksi haarukka. Ruokailua varten filosofi tarvitsee kuitenkin kaksi haarukkaa. Jokainen filosofi ottaa ensin haarukan oikeaan käteen ja sitten vasempaan käteen, yksi kerrallaan. Kun filosofi on saanut molemmat haarukat, hän syö hetkisen, laittaa haarukat takaisin lautasen molemmin puolin ja palaa ajattelemaan.



Kuva 1. Ruokailevien filosofien ruokapöytä

Ongelma syntyy, jos kaikki filosofit istuutuvat syömään samaan aikaan. Jokainen henkilö ottaa haarukan oikealla kädellä, mutta jää odottamaan vasemman käden haarukkaa kunnes se vapautuu. Tästä seuraa *kuolemanlukko* (engl. dead lock), koska kukaan filosofeista ei ole vapauttamassa haarukkaa.

Tietokonemaailmassa filosofit korvataan säikeillä ja haarukat yhteisillä resursseilla, mutta puhutaan tässä tapauksessa kuitenkin edelleen filosofeista ja haarukoista. Jos filosofi on odottamassa haarukan vapautumista vasemman puoleiselta filosofilta, voi käydä siten, että filosofin tilalle tuleekin uusi aterioitsija, joka ottaa haarukan itselleen. Jos tällainen tilanne toistuu, voi odottava filosofi *nälkiintyä* (engl. starvation).

2.2.2 Molemminpuolinen poissulkeminen

Molemminpuolinen poissulkeminen (engl. mutual exclusion) on määritelmän mukaan protokolla, joka takaa, että vain yhdellä prosessilla kerrallaan on pääsy kriittiselle alueelle [EDi65a].

Protokolla on esitelty vuonna 1962, johon E. W. Dijkstra viittaa artikkelissaan [EDi65a]. Tarkemmin ongelma määritellään kahdella ehdolla: a) Yhtäkään prosessiparia ei ole niiden kriittisellä alueella samanaikaisesti. b) Jos prosessi alkaa suorittaa algoritmia, joka suorittaa molemminpuolisen poissulkemisen, niin jossakin vaiheessa jokin toinen prosessi on yrittämässä samaan aikaan pääsyä omalle kriittiselle alueelle. Tästä ei saa seurata kuolemanlukkotilannetta.

Heikompi määritelmä molemminpuolisesta poissulkemisesta korvaa vaatimuksen lukkovapaudesta *heikolla kuolemanlukolla* (engl. weak deadlock), joka sallii kuolemanlukon kontention muodossa [EDi65a].

Tapoja molemminpuolisen poissulkemisen toteuttamiseen on monia. Ensimmäinen sovellusalgoritmi kahden samanaikaisen säikeen synkronointia varten keksi L. Dekker vuonna 1965 [EDi65b]. Tämä *Dekkerin algoritmi*ksi kutsuttu menetelmä mahdollisti ensimmäisen kerran tietorakenteiden jakamisen kahden prosessin välillä ja se vei merkittävästi eteenpäin

tietotekniikan kehitystä prosessien välisen kommunikoinnin saralla. Myöhemmin E. Dijkstra kehitti algoritmin, joka toteuttaa molemminpuolisen poissulkemisen usealle prosessille – riippumatta rinnakkaisten prosessien määrästä. Dijkstra esitteli algoritmin artikkelissaan *Solution of a Problem in Concurrent Programming Control* [EDi65a].

Ajan kuluessa on kehitetty edelleen parempia menetelmiä, joista ehkä tällä hetkellä suosituimmat ovat *Petersonin algoritmi* [GPe81] ja vähemmän tunnettu *Martinin algoritmi* [AMa86].

2.2.3 Kriittinen alue

Kriittinen alue (engl. critical section) on ohjelman koodiosa, jossa joko luetaan tai kirjoitetaan eri säikeiden kesken jaettuun resurssiin [EDi65a]. Kriittisellä alueella tarkoitetaan aluetta, jolle edellisessä luvussa kerrottu molemminpuolinen poissulkeminen halutaan tehdä.

Ruokailevien filosofien ongelmassa kriittistä aluetta voidaan käyttää esimerkiksi niin, että ruokapöytä merkitään kriittiseksi alueeksi. Kun ensimmäinen filosofi asettuu kriittiselle alueelle, jäävät muut filosofit odottamaan pöydän viereen pöydän vapautumista. Näin pöydässä pystyy syömään vain yksi filosofi kerrallaan, mutta kuolemanlukkotilannetta ei voi syntyä.

Kriittinen alue toteutetaan ohjelmakoodissa yleensä luomalla kriittistä aluetta muokkaaville säikeille yhteinen struktuuri, jossa kerrotaan onko joku säie alueella vai ei [EDi65a]. Ensimmäinen kriittiselle alueelle pyrkivä säie merkitsee struktuuriin kriittisen alueen lukituksi. Kun säie on palannut alueelta, se poistaa struktuurin lukituksen.

Kriittisen alueen synkronoinnissa on tärkeää, että alueelta poistuva säie vapauttaa lukituksen. Mikäli kriittinen alue jää lukituksi, mikään muu säie ei pääse alueelle. Jos kriittisen alueen lukitsija ei koskaan vapauta lukkoa, syntyy tällaisesta tilanteesta kuolemanlukko.

Yksi-prosessorisessa ja reaaliaikakäyttöjärjestelmällä varustetussa järjestelmässä kriittinen alue on mahdollista toteuttaa hieman eri tavalla: kun säie on etenemässä kriittiselle alueel-

le, estetään muiden säikeitten keskeytyspyynnöt (engl. disabling of interrupts) ja annetaan kaikki prosessorin suoritusaika ko. säikeelle. Tästä seuraa molemminpuolinen poissulkeminen kriittisellä alueella. [JHa01a]

2.2.4 Semafori

E. W. Dijkstran kehittämä *semafori* on suojattu mekanismi [EDi68], joka ratkaisee klassisen esimerkin ruokailevista filosofeista.

Yksinkertaisesti ajateltuna semafori on kokonaisluku, jolla on erikoistarkoitus. Semafori alustetaan ennen, kuin jaettua resurssia käyttävät rinnakkaiset säikeet käynnistetään. Tämän jälkeen rinnakkaiset säikeet muokkaavat semaforia ainoastaan kahden tarkasti määritellyn atomisen operaation kautta. Näitä operaatioita kutsutaan *P-* ja *V-operaatioiksi* [EDi68]. Kirjaimet P ja V tulevat hollannin kielen sanoista Probeer (yrittää) ja Verhoog (lisätä).

Semaforin alustusarvo on jaettujen resurssien määrä, joita rinnakkaisille säikeille on tarjolla. Jos kyseessä on vain yksittäinen resurssi, on semaforin alustusarvo 1.

P-operaatio testaa, onko semaforin arvo 1 tai suurempi. Jos näin on, operaatio vähentää semaforin arvoa yhdellä. Jos semafori on pienempi kuin 1, tarkoittaa tämä, että jaettu resurssi on käytössä ja operaatio jää odottamaan semaforin arvon kasvamista suuremmaksi kuin 0. V-operaatio kasvattaa semaforin arvoa yhdellä ja signaloi mahdolliselle odottavalle säikeelle resurssin vapautuneen. P- ja V-operaatiot oletetaan suoritettaviksi vain yhden säikeen toimesta kerrallaan. [EDi68]

Semafori eroaa kriittisestä alueesta mahdollisuudella päästää useampi kuin yksi säie muokkaamaan jaettua resurssia.

Seuraavassa pseudo-koodissa on kuvattu P-, V- ja Init-operaatiot. P:n ja V:n semaforiarvon muokkausoperaatiot, sekä P:n semaforiarvon testausoperaatio on oltava yhdessä säikeessä kerrallaan suoritettavia operaatioita.

```
P(Semaphore s)
{
     if(s>0) then                    }
     {                               }
          s = s-1;                   } suoritetaan atomisesti
     }                               }
     else
     {
          AddToWaitingList();
          await(s>0) then            }
          {                          } suoritetaan atomisesti
               s = s-1;              }
          }                          }
     }
}

V(Semaphore s)
{
     s = s + 1;                      } atominen operaatio
     SignalizeWaitingThread();
}

Init(Semaphore s, Integer v)
{
     s = v;
}
```

2.2.5 Lukkovapaat algoritmit

Lukkovapaiden algoritmien (engl. lock-free) tarkoitus on tarjota täysin erilainen lähestymistapa jaetun resurssin käsittelyyn. Sen sijaan, että toteutettaisiin molemminpuolinen poissuljenta jaetulle resurssille, tehdään jaettua resurssia käsittelevästä algoritmista sellainen, ettei molemminpuolista poissulkemista tarvita [MHe93].

Lukkovapaiden algoritmien tekeminen on huomattavan paljon vaikeampaa ja virhealttiimpaa verrattuna perinteiseen molemminpuoliseen poissulkemiseen. Tutkijoita on kuitenkin houkutellut suorituskyvyn parantaminen, joka lukkovapailla algoritmeilla saavutetaan.

Lukkovapaissa algoritmeissa säikeiden suoritusta ei rajoiteta, vaan jokainen säie muokkaa jaettua resurssia haluamallaan tavalla. Algoritmin pääperiaate on siinä, että samanaikaiset muokkaukset jaettuun resurssiin havaitaan ja osataan reagoida siihen siten, että jaettu resurssi pysyy eheänä.

Artikkelissa *Transactional Lock-Free Execution of Lock-Based Programs* [RRa02] esitetään menetelmä, jossa säie aloittaa jaetun resurssin käsittelyn uudelleen, jos konflikti toisen säikeen kanssa havaitaan. Tällaisessa tilanteessa on mahdollisuus *elävään lukkoon* (engl. live lock), jossa molemmat säikeet aloittavat jaetun resurssin käsittelyn yhä uudelleen ja uudelleen. Artikkelissa esiteltiin tämän ongelman ratkaisemiseen säikeiden priorisointia, eli määriteltiin, kumpi säie joutuu aloittamaan jaetun resurssin käsittelyn uudelleen.

Lukkovapaiden algoritmien etuna on suurempi suorituskyky, johtuen

- poistuneesta prosessoinnista, joka molemminpuolista poissuljentaa käyttävissä algoritmeissa kuluu lukkojen käsittelyyn [MHe93] ja
- lukkojen poistumisesta, jotka saattoivat kuluttaa prosessoriaikaa turhaan johtuen lukon jonotusongelmasta, katso luku 2.5 [MHe93].

Lukkovapaat algoritmit helpottavat sovellusten kehittämistä, sillä ohjelmoijan ei tarvitse ottaa kantaa säikeiden synkronointiin. Tämä on erityisen hyvä asia ohjelmoijan kannalta,

sillä ohjelmat voidaan toteuttaa hyvin oliomaisesti, eikä ylimääräisiä kytköksiä säikeiden välille muodostu.

2.2.6 Konsensus-protokolla

Usean asynkronisen säikeen muokatessa yhteistä resurssia, saattaa vikatilanteessa yhteisen resurssin eheys rikkoontua. *Konsensus-protokollat* tuovat tähän ongelmaan ratkaisun.

Konsensus-protokollat antavat usealle asynkroniselle säikeelle mahdollisuuden päästä yhteisymmärrykseen, vaikka osa säikeistä toimisi puutteellisesti tai virheellisesti [GBr83]. Toisin sanoen, konsensus-protokollan avulla tiedetään, milloin yhteinen resurssi sisältää virheitä ja miten ne korjataan.

Gabriel Brachan ja Sam Touegin kirjoittamassa *Resilient Consensus Protocols* -artikkelissa [GBr83] käsitellään kahta tapausta viallisista yhteistä resurssia muokkaavista prosesseista: *fail-stop* ja *malicious*. Ensimmäinen tapaus kuvaa tilannetta, jossa prosessi pysähtyy vikatilanteessa, eivätkä toiset prosessit tiedä onko prosessi pysähtynyt vai onko se pelkästään hidas. Jälkimmäisessä tapauksessa prosessi muokkaa jaettua resurssia virheellisesti eikä pysähdy virheen tapahduttua. Virheiden havaitseminen ja niistä selviytyminen jää toisille prosesseille.

Konsensus-protokollat perustuvat todennäköisyyslaskentaan ja deterministiseen algoritmiin. Tarkemmin tätä aihetta ei tässä tutkielmassa käsitellä.

2.3 Kontentio

Kontentio on määre odotusajalle, jonka säie joutuu odottamaan ennen pääsyä kriittiselle alueelle. [CDw97]

Kontentio tulee esiin prosessien välisessä kommunikoinnissa, kun käytetään yhteistä resurssia, jonka kautta informaatio välitetään prosessilta toiselle. Yhteinen resurssi on yleensä suojattu semaforilla tai muulla vastaavalla mekanismilla. Kun ensimmäinen säie on va-

rannut yhteisen resurssin, on toisen säikeen odotettava resurssin vapautumista. Tätä odotusaikaa sanotaan kontentioksi.

Järjestelmän luonteesta riippuen kontentiosta ei aina ole haittaa. Esimerkiksi, jos kahden tai useamman prosessin ”törmäyksiä” yhteiseen resurssiin tapahtuu prosessorin mittapuussa harvoin, on kontentioon kuluva aika niin pieni, ettei sillä ole merkittävää vaikutusta järjestelmän toimintaan. Järjestelmissä, joissa ”törmäyksiä” tapahtuu paljon, prosessit voivat joutua odottamaan pääsyä yhteiseen resurssiin huomattavan osan ajasta.

Samassa yksiprosessorisessa mikrotietokoneessa olevilla prosesseilla on kontentiolla hieman eri vaikutus. Kuvitellaanpa seuraavaa tilannetta: prosessi A saa pääsyn jaettuun resurssiin. Samaan aikaan myös prosessi B pyytää pääsyä jaettuun resurssiin ja jää odottamaan A:n poistumista alueelta. Koska molemmat prosessit käyttävät saman prosessorin suoritusaikaa, saa prosessi A enemmän suorittimen kellojaksoja. Tästä seuraa, että A selviytyy nopeammin pois yhteiseltä alueelta.

Kontentiota voidaan pienentää jakamalla kriittiset alueet useaksi pieneksi alueeksi. Tällöin todennäköisyys sille, että kaksi säiettä muokkaisi juuri samaa kriittistä aluetta, on pienempi. Hyvä esimerkki tästä tulee esiin tietokannoissa. Kun jokin säie muokkaa jonkin taulun tiettyä riviä, kannattaa lukita ainoastaan kyseinen rivi eikä koko taulua. Tällöin muut säikeet voivat muokata saman taulun muita rivejä. Jos koko taulu olisi lukittu, mikään muu säie ei voisi muokata taulua.

Hajautetuissa järjestelmissä, joissa prosessit käyttävät eri prosessoreiden suoritusaikaa, on kontentiolla suurempi merkitys suhteutettuna prosessorien kuormitusasteeseen [RAl94]. Tämä on loogista, sillä moniprosessorikoneessa jaettuun resurssiin pääsyä odottavan säikeen prosessori on tyhjäkäynnillä niin kauan, kun resurssi vapautuu.

2.4 Molemminpuolisen poissuljennan jätekuorma

Aikaisemmassa luvussa 2.2.4 esitetty semafori oli eräs tapa rajoittaa säikeiden pääsyä kriittiselle alueelle. Kun säie yrittää pääsyä kriittiselle alueelle, se tekee P-operaation, joka va-

raa kriittisen alueen säikeen käyttöön. Säikeen poistuessa kriittiseltä alueelta, säie poistaa varauksen semaforin varauslistasta ja vapauttaa kriittisen alueen toisten säikeiden käyttöön. Tämä proseduuri vie järjestelmältä hieman prosessoriaikaa ja muistia.

Molemminpuolisen poissuljennan jätekuorma (engl. lock overhead) tarkoittaa molemminpuolisen poissuljennan toteuttamiseen tehdyn mekanismin aiheuttamaa kuormaa järjestelmälle [RRa02].

Mitä enemmän resursseja suojataan kriittisillä alueilla ja mitä useammin säikeet käyttävät näitä alueita, sitä enemmän jätekuormaa syntyy. Monesti molemminpuolisen poissuljennan jätekuorma voi muodostua järjestelmän pullonkaulaksi, jos ohjelmien arkkitehtuuri on huonosti suunniteltu.

2.5 Lukon jonotusongelma

Lukon jonotusongelma (engl. lock convoying) johtuu prosessorin säikeiden aikataulutuksesta ja useiden säikeiden jonotuksesta kriittiselle alueelle [MHe93].

Ongelma muodostuu seuraavanlaisessa tilanteessa. Jaettua resurssia on muokkaamassa monta säiettä, joista yksi on parhaillaan kriittisellä alueella. Jos tämän säikeen aikaikkuna loppuu, prosessori tekee kontekstin vaihdon ja antaa aikaa kriittiselle alueelle pääsyä odottaville säikeille. Niiden ajaminen on tässä tapauksessa turhaa, sillä säikeillä ei ole mahdollisuutta tehdä mitään, ennen kuin kriittisellä alueella oleva säie on palannut alueelta pois. Prosessoriaikaa kuluu tässä tapauksessa hukkaan.

2.6 Hajautetut jaetut muistit

Hajautetulla jaetulla muistilla (engl. Distributed Shared Memory, lyhennettynä DSM) tarkoitetaan prosessien kesken jaettua muistia, joka on hajautettu eri tietokoneisiin [KLi86].

Hajautetun jaetun muistin käsittely voidaan integroida käyttöjärjestelmään, jolloin jaettu muisti on läpinäkyvää sovellukselle. Tällä tapaa hajautetun jaetun muistin käsittely voidaan ymmärtää virtuaalimuistiarkkitehtuurin jatkeeksi [KLi86]. Toinen tapa on toteuttaa

hajautettu jaettu muisti ohjelmallisesti, mutta tällöin sovelluksessa on tehtävä erikseen muistinkäsittely. Jälkimmäinen tapa mahdollistaa paremman siirrettävyyden eri järjestelmien välillä.

Hajautettujen muistien käsittelylle voisi omistaa kokonaisien kirjojen verran tutkimusta, joten tästä tutkielmasta ko. aihe on jätetty pois.

2.7 Kommunikointitavat

Prosesseilla on monia mahdollisuuksia kommunikoida keskenään. Toiset tavat ovat nopeampia kuin toiset, mutta alttiimpia vioille ja sovellusvirheille. Luotettavat menetelmät puolestaan ovat poikkeuksetta hitaampia.

Seuraaviin alalukuihin on koottu tärkeimmät prosessienväliset kommunikointimenetelmät.

2.7.1 Tiedostot

Tiedosto on joukko informaatiota, joka sisältää ohjelmien käytettävissä olevaa dataa. Data koostuu perättäisistä tavuista, jonka määrän rajoittaa käytössä oleva tiedostojärjestelmä. Tiedostojen formaatti on päätetty niitä kirjoittavissa ohjelmissa, joten tiedostojen sisältämä data voi olla ihmiselle mielivaltaista. Tiedostolla on aina nimi, jolla tiedostoon voidaan viitata. Tiedostot on yleensä tallennettu johonkin massamuistille, jotta tiedostot olisivat pysyviä. Näin informaatiota saadaan talletettua, vaikka sen luonut ohjelma suljetaan.

Tiedostoja voidaan käyttää prosessien välisenä kommunikointimenetelmänä. Useat käyttöjärjestelmät sallivat, että tiedosto voidaan avata samanaikaisesti eri ohjelmiin. Tällä tavoin voidaan esimerkiksi toteuttaa yksinkertainen datan välitysmenetelmä kahden prosessin välille. Ensimmäinen prosessi kirjoittaa tiedoston loppuun, jonka jälkeen toinen prosessi voi lukea kirjoitetun datan.

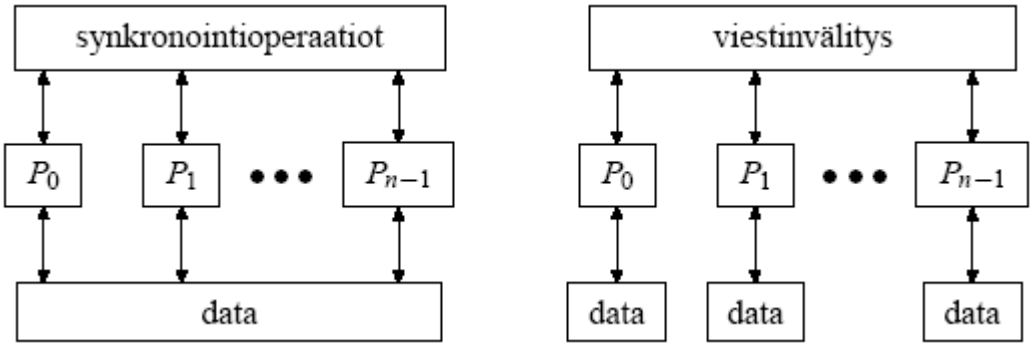
Kommunikointi tiedostojen kautta ei ole nopeaa, sillä tiedostot sijaitsevat yleensä massamuistiasemalla, jonka tiedonsiirtokyky on monin verroin hitaampaa kuin esimerkiksi tietokoneen keskusmuistilla.

2.7.2 Viestinvälittäminen

Viestinvälittämisellä (engl. Message Passing) voidaan lähettää ja vastaanottaa dataa rinnakkaisten prosessien välillä. Viestit voidaan välittää tiedonsiirtoverkon tai tiedonsiirtoväylän kautta. Viestin mukana kuljetetaan data, jota prosessin on tarkoitus muokata. [JHa01b]

Jos prosessit käyttävät jaettua muistia viestinvälittämisen ohella, voidaan viesteihin sisällyttää osoitin jaetun muistin sisällä olevaan dataan. Tosin, tällöin asynkronisessa järjestelmässä menetetään viestinvälittämisen etu, lukkovapaus.

Seuraavassa kuvassa 2 näkyvät jaettua muistia ja viestinvälitystä kommunikointiin käyttävien prosessien erot. Vasemmalla olevassa kuvassa data on yhteistä prosesseille, mutta prosessit joutuvat käyttämään synkronointioperaatioita, jotta yhteinen data ei korruptoituisi. Viestinvälitysarkkitehtuurissa data on jokaisella prosessilla lokaalissa muistissa, mutta muokattu data on välitettävä viestinvälitysväylän kautta toisille prosesseille.



Kuva 2. Jaettu muisti vs. viestinvälitys

Viestinvälitystä varten on muodostettu MPI standardi (engl. *Message Passing Interface*) [For95], jotta suuret tuotevalmistajat kehittäisivät yhteensopivia menetelmiä. Tämän standardin avulla viestinvälitys toimii eri käyttöjärjestelmien välillä ja näin ko. menetelmästä tulee hyvin voimakas väline alustariippumattomuutensa vuoksi.

2.7.3 Soketit

Soketti määritellään yksilölliseksi verkkokommunikoinnin alku- tai loppupisteeksi. [RFC793]. Kahdella soketilla voidaan muodostaa yksi TCP-yhteys [RFC793].

Soketit voidaan ajatella rajapintana, jota sovellus käyttää keskustellessaan jonkin toisen sovelluksen kanssa verkon yli. Soketti liitetään sovelluksessa valitun siirtokerroksen protokollan päälle. Mahdollisia protokollia ovat esim. TCP tai UDP.

Soketti-rajapinnasta on toteutettu useita versioita eri valmistuttajien toimesta, kuten Microsoftin ja Ciscon. Tunnetuin rajapinta on kuitenkin Berkeley socket -rajapinta (*Berkeley Sockets Layer*), joka sai alkunsa 4.2BSD Unix käyttöjärjestelmästä vuonna 1983 [JQu85]. Näinä päivinä Berkeleyn soketit muodostavat alan de facto -standardin.

Soketit liitetään yleensä verkkokommunikointiin ja lähes aina lähiverkossa ja internetissä toimivien ohjelmien käyttöön.

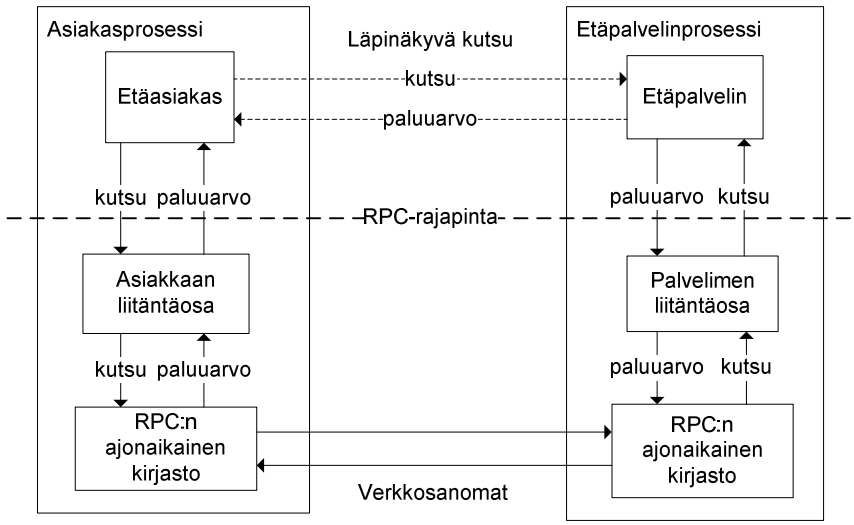
Sokettejä käyttäen voidaan toteuttaa prosessien välinen kommunikointi lokaalin koneen sisällä, mutta myös lähiverkossa tai internetissä. Sokettien suorituskykyä rasittaa sen rakenne, joka on suunniteltu internet-käyttöön. Verkkokommunikointi voi olla suorituskykyä vaativissa järjestelmissä turhan raskas saman tietokoneen sisällä.

2.7.4 Etäkutsut

Etäkutsu (engl. Remote Procedure Call, RPC) on protokolla, jolla pystytään hajauttamaan ohjelmassa suoritettava alirutiini johonkin toiseen tietokoneeseen [ABi84].

Kun etäkutsu ajetaan, pysäytetään kutsujan kontrolli etäkutsun ajaksi. Parametrit välitetään siirtotien yli ympäristöön, jossa etäkutsu suoritetaan. Vastaavasti, kun proseduuri on suoritettu, tulokset välitetään takaisin kutsuvalle osapuolelle ja kutsun tehnyt prosessi voi jatkaa suoritusta. [ABi84]

RPC-konseptiin liittyy eri komponentteja, jotka mahdollistavat läpinäkyvyyden kutsuvan ja kutsuttavan osapuolen välille. Oheinen kuva 3 selvittää komponenttien rakennetta:



Kuva 3. RPC-liityntä [ABi84]

Etäkutsua kutsuvassa ohjelmassa on seuraavat komponentit: etäasiakas, asiakkaan liitäntäosa (engl. client-stub) ja RPC:n ajonaikainen kirjasto (engl. RPC Runtime library) [ABi84]. Asiakkaalla tarkoitetaan tässä yhteydessä kutsun tekevää ohjelmaa, joka aloittaa etäkutsun tekemällä funktiokutsun. Asiakkaan liitäntäosan vastuulle jää luoda yksi tai useampia sanomia, jotka sisältävät tiedon funktiokutsusta ja sen parametreista. Tämän jälkeen RPC:n ajonaikainen kirjasto välittää kutsun argumentteineen käytössä olevan siirtotien yli kutsuttavalle etäpalvelimelle.

Kutsuttavassa päässä RPC:n ajonaikainen kirjasto vastaanottaa kutsun ja välittää datan palvelimen liitäntäosalle (engl. server-stub), joka purkaa tulevat sanomat ja muodostaa tämän saadun informaation pohjalta tavallisen funktiokutsun [Abi84]. Lopuksi palvelimen liitäntäosa herättää etäpalvelimen ja kutsuu tämän funktiota saaduilla argumenteilla. Kun funktio on suoritettu, tulokset välitetään etäpalvelimelta kutsun tehneelle ohjelmalle.

RPC-protokolla ei ota kantaa yhteyden luotettavuuteen. Luotettavuuden aste riippuu siitä, minkä siirtokerroksen päällä RPC toimii: TCP / IP on luotettavampi kuin UDP. Sovelluksen on kuitenkin viime kädessä huolehdittava itse vikatilanteet. [RFC1831]

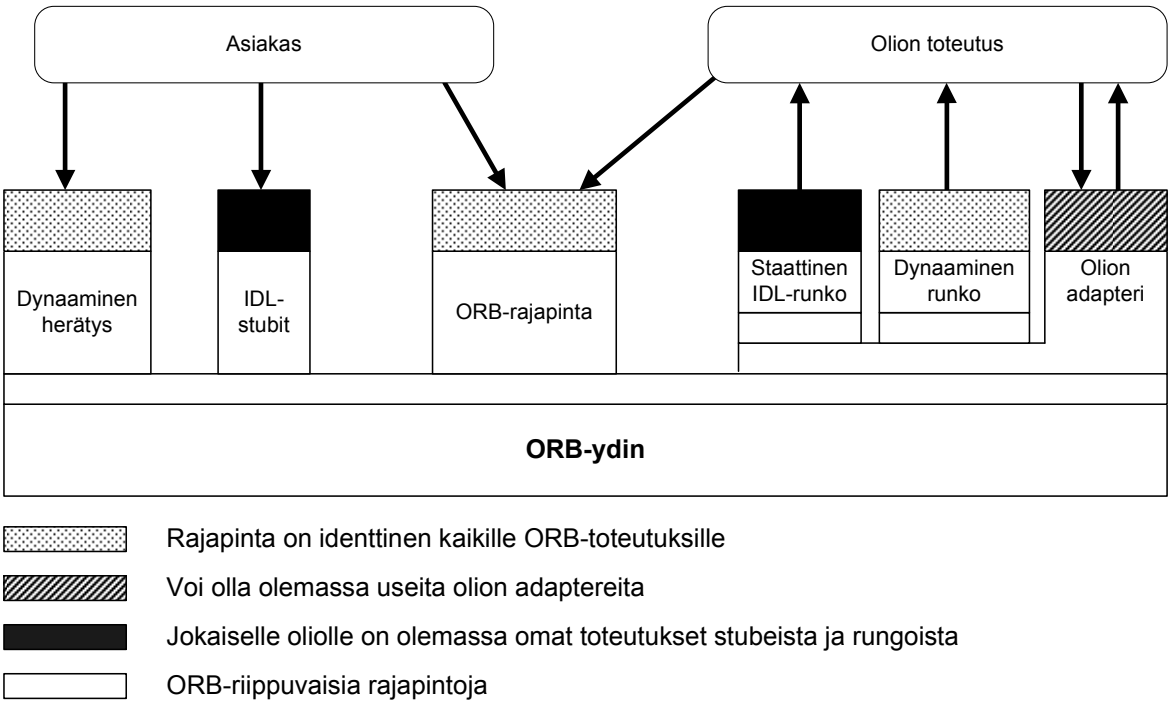
Etäkutsun hyvänä puolena on selkeä prosessien välinen rajapinta, joka muodostuu suoraan funktiokutsusta. Etäkutsun tekemä sovellus kutsuu funktiota niin kuin mitä tahansa muuta funktiota ja jää odottamaan paluuarvoa.

2.7.5 CORBA

CORBA (*Common Object Request Broker Architecture)* on Object Management Group:in määrittelemä standardi [OMG04], joka tekee mahdolliseksi eri ohjelmointikielillä kirjoitettujen ohjelmien prosessien välisen kommunikoinnin ja hajautuksen.

CORBA:n pääperiaatteena on tarjota asiakkaan ja palvelimen välille heterogeeninen rajapinta, joka läpinäkyvästi muuntaa asiakkaalta tulevan pyynnön palvelimella olevalle oliolle sopivaksi. Sovelluksen ei tarvitse ottaa kantaa siihen, missä olio sijaitsee tai kuinka pyyntö välitetään palvelimelle, vaan nämä asiat jäävät CORBA:n vastuulle.

CORBA:ssa on monia samoja piirteitä kuin etäkutsuissa, joita esiteltiin edellisessä kappaleessa. Osapuolten välinen rajapinta määritellään IDL-määrittelykielellä (*Interface Definition Language*) [OMG04]. Asiakasta varten tehdään IDL-kääntäjällä tarvittavat IDL-stubit, joita asiakassovellus sitten kutsuu. Palvelinsovelluksessa on vastaavat rajapinnat tulevia kyselyjä varten. Etäkutsuista eroten, CORBA tarjoaa myös dynaamisen herätyksen (*dynamic invocation*), jossa kutsuttavat metodit selvitetään ajon aikana. Oheisessa kuvassa 4 on CORBAn arkkitehtuurimalli.



Kuva 4. CORBA:n komponenttimalli [OMG04]

2.8 Käyttöjärjestelmien tarjoamat menetelmät

Tähän alilukuun on koottu osa käyttöjärjestelmien tukemista prosessienvälisistä kommunikointimenetelmistä.

2.8.1 Tapahtumat

Windowsin tarjoamiin peruspalveluihin kuuluvat synkronisointioliot, joihin *tapahtumat* (engl. events) lukeutuvat [MSDNd]. Tapahtuma voidaan ajatella signaalina, jolla herätetään sitä odottava säie suorittamaan sille määrättyä tehtävää.

Tapahtumien avulla voidaan toteuttaa yksinkertaisen tilatiedon välittäminen toiselle prosessille. Tapahtumien avulla saadaan myös toteutettua säikeiden synkronisointi. Esimerkiksi ohjelmassa voidaan sopia, että säie voi mennä kriittiselle alueelle vasta sitten, kun tapahtuma on lauennut.

2.8.2 Leikepöytä

Leikepöydän tarkoitus on tarjota käyttäjälle helppo ja kätevä tapa siirtää ohjelmien välillä tekstiä, kuvia ja sen leikkeitä, tiedostoja ja käytännössä kaikkea, mitä kopioitavan objektin ohjelma ja kopioitavan kohteen ohjelma sallii. [MSDNe]

Leikepöydän avulla on mahdollista toteuttaa prosessien välistä kommunikointia myös siten, että käyttäjän ei tarvitse koordinoida kommunikoinnin kulkua. Esimerkiksi voidaan toteuttaa ohjelma, joka vastaanottaa informaatiota leikepöydän kautta. Toisen ohjelman tarkoituksena on tuottaa informaatiota leikepöydälle. Yksinkertainen vuonhallinta ohjelmien välille voitaisiin toteuttaa tutkimalla onko leikepöytä tyhjä vai ei. Informaation vastaanottajan olisi tyhjennettävä leikepöytä sen jälkeen kun informaatio on kopioitu ohjelman lokaaliin muistiin.

Useimmissa käyttöjärjestelmissä leikepöydän käyttö on rajoitettu yhden tietokoneen sisälle. Leikepöytä on suunniteltu pääasiassa käyttäjän palvelua varten, joten se ei sovellu varsinaiseksi prosessien väliseksi kommunikointimenetelmäksi.

2.8.3 Putket

Putket on lähtöisin UNIX-käyttöjärjestelmään kehitetystä menetelmästä, jossa prosessin tuloste pystyttiin ohjaamaan putken läpi toiselle prosessille.

Yksi Windowsin peruspalvelu tarjoaa ohjelmoijalle *nimetyn putken* (engl. named pipe), jolla voidaan siirtää dataa tehokkaasti prosessilta toiselle. *Microsoft Developer Networkin* artikkelin *Named Pipes* [MSDNa] mukaan nimetty putki käyttää sisäisesti jaettua muistia tiedon välittämiseen prosessien välillä. Menetelmään on myös toteutettu verkkotuki, joten nimetty putki toimii myös lähiverkossa ja sillä voidaan siirtää tietoa toisessa tietokoneessa sijaitsevalle prosessille. Nimetty putki on joissakin tilanteissa oivallinen työkalu, sillä tällä menetelmällä toteutetut sovellukset voidaan hajauttaa eri tietokoneisiin, jos tiedonsiirtoon kuluva aika ei ole ko. järjestelmässä kriittinen.

Putkella on aina kaksi päätä. Se prosessi, joka luo nimetyn putken on *putkipalvelin* (engl. pipe server) [MSDNa]. Luotuun putkeen liittyvää prosessia puolestaan nimitetään *putkiasiakkaaksi* (engl. pipe client) [MSDNa].

Nimetty putki luodaan antamalla sille nimi ja palvelinkoneen nimi, jossa putkipalvelin sijaitsee. Tämän jälkeen putkiasiakkaat voivat liittyä nimettyyn putkeen globaalin nimen avulla. Data voidaan kirjoittaa putkeen joko tavuvirtana tai viestivirtana. [MSDNa]

2.8.4 Postilaatikko

Windowsin peruspalvelu *postilaatikko* (engl. mail slot) tarjoaa sovelluksille asynkronisen yksisuuntaisen prosessien välisen kommunikointimenetelmän. *Microsoft Developer Networkin* [MSDNb] artikkelin *mailslots* saatavien tietojen mukaan postilaatikon kautta tapahtuvaa tiedon välitystä ei taata. Kovin luotettavana menetelmänä postilaatikkoa ei siis voida pitää.

Sitä prosessia, joka luo postilaatikon, kutsutaan *postilaatikkopalvelimeksi* (engl. mailslot server) [MSDNb]. Postilaatikkopalvelin on ainoa osapuoli, joka voi vastaanottaa postia. Dataa lähettäviä instansseja kutsutaan *postilaatikkoasiakkaiksi* (engl. mailslot client) [MSDNb]. Postilaatikkoasiakkaita voi olla useita ja postilaatikko-palvelin voi sijaita nimettyjen putkien tapaan eri tietokoneessa, mutta kuitenkin samassa verkossa. Tiedonsiirto kulkee vain postilaatikkoasiakkaalta postilaatikkopalvelimelle ja tiedonsiirto tapahtuu aina asynkronisesti.

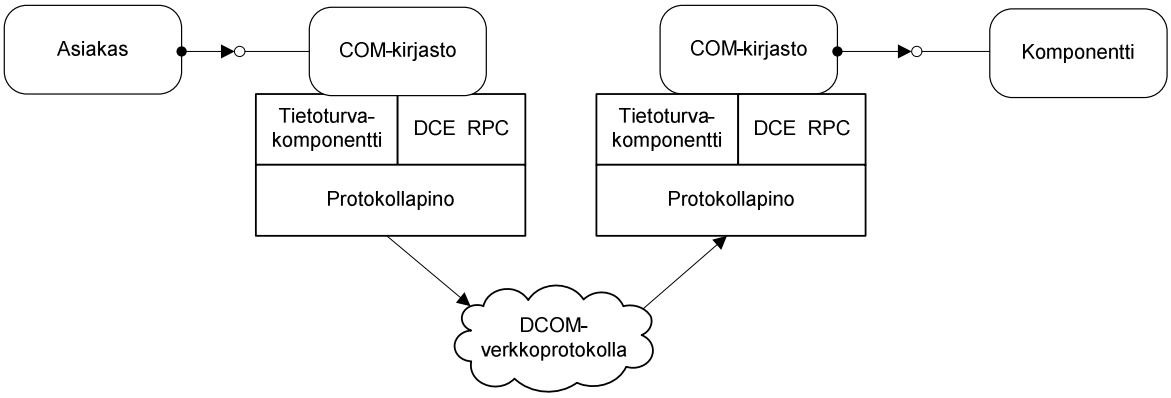
Postilaatikoita, eli postilaatikkopalvelimia, voi olla useita samannimisiä saman verkon alla [MSDNb]. Tällöin on mahdollista lähettää postilaatikko-asiakkaan toimesta broadcast-tyyppinen sanoma, joka menee samanaikaisesti kaikkiin ko. postilaatikoihin.

2.8.5 DCOM

DCOM (engl. *Distributed Component Object Model*) on Microsoftin kehittämä teknologia, joka jatkaa Microsoftin COM-alustaa lisäämällä siihen COM-olioiden hajautuksen eri tietokoneisiin [MSDNc].

DCOM:ssa on hyvin paljon samankaltaisuutta kuin CORBA:ssa ja etäkutsuissa. Asiakkaan tekemät metodikutsut välitetään läpinäkyvästi DCOM-rajapinnan yli komponentille tietämättä, missä käytettävä komponentti sijaitsee. DCOM on toteutettu virallisesti vain Microsoftin käyttöjärjestelmiin, joten menetelmä ei ole alustariippumaton CORBA:an verrattuna.

Kuva 5 kertoo DCOM-arkkitehtuurin komponentit ja niiden suhteet, kun asiakas ja sen käyttämä COM-komponentti sijaitsevat eri tietokoneissa.



Kuva 5. COM-komponentit eri tietokoneissa [MSDNc]

DCE RPC (engl. Distributed Computing Enviroment / Remote Procedure Call) vastaa COM-komponentin funtiokutsujen argumenttien marsaloinnista, sekä etäkoneessa olevan COM-olion resurssien varaamisesta ja vapauttamisesta. [MSDNc]

2.8.6 Jaettu muisti

Jaettu muisti on muistialue, joka on jaettu prosessien kesken. Loogisesti ajateltuna tämä on hyvin nopea tapa jakaa tietoa prosessien välillä, sillä prosessit käsittelevät samaa muistialuetta ja tyypillisesti muistin käsittely on nykytiekoneissa nopeaa.

Jaettu muisti ei ole Windows NT -käyttöjärjestelmissä suoraan jaettu prosessien kesken, vaan käyttöjärjestelmä joutuu tekemään työtä taustalla, jotta prosessit näkisivät toisten tekemät muutokset jaettuun muistiin.

Seuraavassa luvussa 3 selvitetään tästä aiheutuvia ongelmia ja menetelmiä, joita tarvitaan jaetun muistin kautta tapahtuvassa prosessien välisessä kommunikoinnissa.

3 Jaetun muistin kautta tapahtuva kommunikointi

Tässä luvussa käsitellään asioita, jotka liittyvät jaetun muistin kautta tapahtuvaan kommunikointiin. Ensiksi tutkitaan hieman nykykäyttöjärjestelmien ja prosessoreiden muistiarkkitehtuureja, jotta käsitetään, mitä jaettu muisti oikein tarkoittaa, mitä sen käyttämiseksi vaaditaan ja millaisiin rajoitteisiin ko. menetelmässä törmätään. Lopuksi käydään läpi muistin konsistenssimalleja, jotka etenkin moniprosessorijärjestelmissä vaikuttavat jaetun muistin kautta tapahtuvaan kommunikointiin.

Käyttöjärjestelmät, jotka ovat valittu tämän tutkielman piiriin, rajoittuvat lähinnä kotitietokoneessa käytettäviin järjestelmiin eli Windows NT:hen ja Linuxiin.

Ennen kuin pääsemme syventymään datansiirtoon jaetun muistin kautta, on selvitettävä muutamia tärkeitä asioita, jotka muodostavat rajoja kommunikoinnille.

3.1 Prosessorien muistiarkkitehtuurit

Prosessori on tietokoneen ydin, joka käytännössä tekee kaiken työn, kun ohjelmaa suoritetaan. Näin ollen prosessori muodostaa rajat sille, kuinka suurissa paloissa ohjelmaa ajetaan ja kuinka suuri muistiavaruus on mahdollista osoittaa.

Vuonna 1985 kehitettiin, nykyäänkin suosittu prosessoriarkkitehtuuri, IA-32 (Intel Architecture, 32 bit) [IA64]. Myös i386:ksi kutsuttu arkkitehtuuri mahdollistaa 32-bittisten datalohkojen käsittelyn, prosessorin tukeman moniajon, virtuaalimuistin ja muistin suojauksen. IA-32:ssa on kahdeksan 32-bittistä yleiskäyttöistä rekisteriä sovellusten käyttöön, kahdeksan liukulukupinorekisteriä sekä joukko 16- ja 8-bittisiä datarekistereitä [IA64]. Uudemmat IA-32 arkkitehtuuriin perustuvat eri valmistajien prosessorit lisäsivät rekistereiden määrää multimediatekniikoilla, kuten MMX:llä [IA64], 3DNow!:lla [AMD64], SSE:llä [IA64] ja SSE2:lla [AMD64].

32-bittinen prosessori käsittelee dataa aina 32 bitin lohkoissa [IA64]. Tämä tarkoittaa sitä, että vaikka lasketaan kaksi yhdellä tavulla ilmoitettavissa olevaa kokonaislukua yhteen, sijoitetaan luvut neljän tavun mittaisiin rekistereihin ja lasketaan luvut yhteen. Tällöin mo-

lemmissa rekistereissä on kolmen tavun verran ”tyhjää” tilaa, jota ei hyödynnetä. 64-bittisissä prosessoreissa hukkatilaa tulee entistä enemmän.

IA-32 arkkitehtuuri tukee rinnakkaisia virtuaaliosoiteavaruuksia [IA64], jolloin jokainen suoritettava prosessi saa tavallaan oman ”hiekkalaatikon”. IA-32 prosessori vaatii käyttöjärjestelmän, joka osaa käskeä prosessoria vaihtamaan rekistereiden osoittaman muistiavaruuden toiseen ja laittamaan vanhan talteen. Tällä tavalla saadaan simuloitua moniajoa, kun kontekstia vaihdetaan tarpeeksi usein – vaikkapa 1000 kertaa sekunnissa. Koska prosessorilla on rautatuki kontekstin vaihtoon [IA64], on ko. operaatio todella nopea suorittaa.

Teoriassa 32-bittinen prosessori pystyy viittaamaan 2^{32} virtuaalimuistiosoitteeseen, koska muistiosoitteen koko on 32 bittiä ja 32 bitillä voidaan ilmoittaa 2^{32} eri lukua. IA-32 arkkitehtuurin prosessoreissa näin on myös tehty käytännössä. Käyttöjärjestelmä tuo lisää rajoituksia virtuaaliosoiteavaruuden käsittelyyn, josta on kerrottu tarkemmin luvussa 3.3.

IA-32-arkkitehtuurilla saadaan käyttöön nykyaikaisten prosessorien tukema *kontekstin vaihto* (engl. context switch), eli suoritettavan prosessin vaihto lennossa. Suoritin ajaa käyttöjärjestelmän ohjaamana annettua prosessin koodia ja kun suorittimelle tulee käsky vaihtaa kontekstia, tallentaa prosessori ohjelmaosoitteen ja rekisterit talteen, hakee välimuistista toisen prosessin ohjelmaosoitteen ja rekisterit ja jatkaa suoritusta.

IA-64-arkkitehtuurin prosessori käsittelee dataa 64-bittisinä datalohkoina [IA64]. Virtuaaliosoiteavaruuden koko on teoriassa se, kuinka monta erilaista muistiosoitetta 64-bitin muuttujalla pystytään osoittamaan. Käytännössä tämä tarkoittaa lukua 2^{64}, joka on 16 eksatavua.

Seuraavassa taulukossa 1 on kerrottu eri prosessoreiden muistiarkkitehtuurien koot. [IA64] [AMD64]

Prosessori	Fyysisen muistiosoitteen koko (bittiä)	Virtuaalisen muistiosoitteen koko (bittiä)	Virtuaaliavaruuden koko
AMD Athlon 64	40	64	16 Et
AMD Opteron	40	64	16 Et
Intel Xeon	40	64	16 Et
Intel Itanium	50	64	16 Et

Taulukko 1. Prosessorien muistiarkkitehtuurien koot

3.2 Muistiavaruus

Muistiavaruudella tarkoitetaan sitä joukkoa osoitteita, joihin pystytään viittaamaan tietokoneella. Toisin sanoen tietokoneeseen liitettävien laitteiden, datamuistin ja ohjelmamuistin määrä riippuu ko. tietokoneen osoitteen bittien määrästä.

Modernit käyttöjärjestelmät (OpenVMS, Unix, Linux, Windows NT) varaavat jokaista prosessia varten oman virtuaaliosoiteavaruuden. Tällä tavalla voidaan suojautua mahdollisia prosessin tekemiä virheitä vastaan. Suojaamattomissa muistiarkkitehtuureissa koko muistiavaruus oli sellaisenaan kaikkien prosessien käytettävissä. Tästä syystä yhden prosessin tekemä virhe, esim. kirjoittaminen yli puskurin, saattoi sekoittaa tai lukita koko käyttöjärjestelmän.

Virtuaaliosoiteavaruuden koko määräytyy käytössä olevan mikrotietokoneen prosessoriarkkitehtuurista. Seuraava taulukko 2 kertoo osoitettavien muistipaikkojen määrän järjestelmän arkkitehtuurin mukaan.

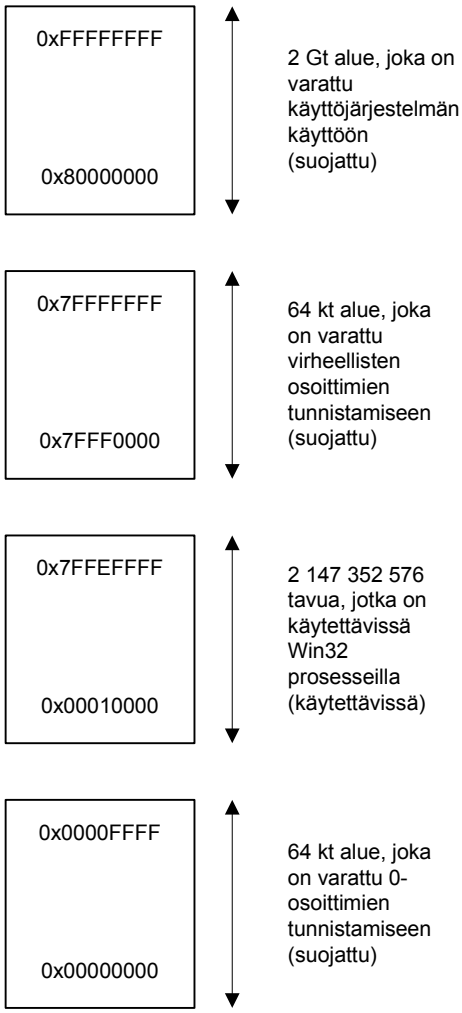
Osoitteen koko (bittiä)	Osoitettavien muistipaikkojen määrä
8	256
16	65 536
32	4 294 967 296
64	18 446 744 073 709 551 616

Taulukko 2. Muistiavaruuden koko suhteessa osoitteen leveyteen

3.3 Windows NT:n muistialueet

Microsoft on päättänyt jakaa Windows NT käyttöjärjestelmissä prosessin näkemän virtuaalimuistiavaruuden eri segmentteihin [JRi96]. Tällä tavalla on pystytty piilottamaan käyttöjärjestelmän komponentit suojatulle virtuaalimuistialueelle, joten sovellus ei voi tehdä harmia kuin itselleen.

Jeffrey Richter kertoo kirjassaan *Advanced Windows* [JRi96] Windows NT:n muistiarkkitehtuurista ja selvittää kuinka virtuaalimuistiavaruus toimii prosessin näkökulmasta. Kuvassa 6 kerrotaan Windows NT:ssä käytettyjen muistisegmenttien tarkoitukset ja alueet.



Kuva 6. Windows NT:ssä ajettavan prosessin muistiavaruus

Prosessin virtuaaliosoiteavaruuden pohjimmainen osoitealue (0x00000000 - 0x0000FFFF) on suojattu kaikilta luku- tai kirjoitusoperaatioilta. Jos sovellus koettaa päästä käsiksi ko. alueeseen, käyttöjärjestelmä estää tämän ja lopettaa ohjelman suorituksen. Tällä estetään 0-osoittimien käyttö.

Seuraava, noin kahden gigatavun kokoinen osoitealue (0x00010000 - 0x7FFEFFF), on prosessin käsiteltävissä. Tämä alue on se, johon Windows NT järjestelmässä suoritettava ohjelma kirjastoineen sijoitetaan. Muistin allokoiminen tapahtuu käyttöjärjestelmän kautta ja saadut varatut muistilohkot sijoitetaan aina tälle kyseiselle muistialueelle. Windows XP

ja Windows Server 2003 käyttöjärjestelmissä on mahdollista suurentaa tämän segmentin koko noin kolmeen gigatavuun, jolloin sovellusten käyttöön saadaan enemmän virtuaaliosoiteavaruutta. Tällöin seuraavan segmentin koko pienenee yhteen gigatavuun.

Edellisen alueen yläpuolella, alueella (0x7FFF00000 - 0x7FFFFFFF), on jälleen suoja-alue pohjimmaisen muistialueen tapaan, jotta sovelluksessa mahdolliset vialliset puskurin ylittämiset ja tästä seuraavat luvattoman muistipaikan luku- ja kirjoitusoperaatiot havaitaan. Jos tällainen tapahtuu, käyttöjärjestelmä lopettaa ohjelman suorituksen.

Viimeisimpänä, ylimpänä ja suurimpana muistialueena (0x80000000 - 0xFFFFFFFF) on prosessilta suojattu kahden gigatavun kokoinen segmentti, joka sisältää käyttöjärjestelmän ytimen ja laiteajureiden tarvitsemaa koodia.

3.3.1 Ohjelman virtuaaliosoiteavaruus ja ohjelman suoritus

Moderneissa käyttöjärjestelmissä (Windows NT, Linux) tapahtuu monia asioita kun prosessi käynnistetään. Prosessia varten tehdään oma virtuaalimuistiavaruus, jossa ohjelma ajetaan. Jos ohjelma suorittaa laittoman toiminnon, kuten käyttää muistiosoitetta, johon ei ole allokoitu muistia, ko. toiminto ei hajota käyttöjärjestelmän muistiavaruutta. [JRi96]

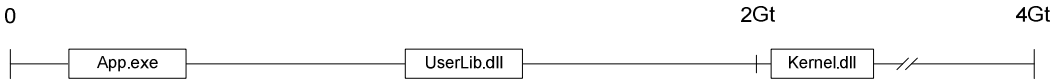
Ohjelman suoritus alkaa luomalla prosessia varten virtuaalimuistiavaruus. Tämän avaruuden koko määräytyy käyttöjärjestelmän käyttämän muistiosoitteen leveydestä (tällä hetkellä yleisimmät koot ovat 32 ja 64 bittiä). Oheinen kuva 7 ko. tilanteesta on toteutettu 32-bittisessä järjestelmässä.

Kuva 7. Prosessin virtuaaliosoiteavaruus

Seuraavaksi ajettavan prosessin konekielikoodi liitetään ohjelman virtuaaliosoiteavaruuteen [JRi96], kuva 8.

Kuva 8. Suoritettava koodi on linkitetty virtuaaliosoiteavaruuteen

Viimeisenä askeleena liitetään käyttöjärjestelmän ytimen ja ohjelman tarvitsemat kirjastot virtuaaliosoiteavaruuteen [JRi96]. Kuva 9 kertoo virtuaaliosoiteavaruuden tilan.



Kuva 9. Ytimen ja ajettavan ohjelman kirjastot virtuaaliosoiteavaruudessa

Lopuksi käyttöjärjestelmä luo säikeen ajamaan ohjelmaa konekielikoodin alusta. Kaikki ohjelman tekemät muistinvaraukset tapahtuvat käyttöjärjestelmän kautta. Varattu muisti, eli muistisivut, liitetään käyttöjärjestelmän määräämiin osoitteisiin virtuaaliosoiteavaruuteen, kuva 10.

Kuva 10. Ohjelman allokoimat muistilohkot virtuaaliosoiteavaruudessa

Jos prosessi koettaa käyttää sellaista osoitetta, jota ei ole varattu käyttöjärjestelmän toimesta, tai ko. osoite sijaitsee koodisegmentin tai kirjastojen muistialueella, estää käyttöjärjestelmä tämän ja lopettaa ohjelman suorituksen.

3.4 Muistin varaaminen

Nykykäyttöjärjestelmät käsittelevät kaiken muistin virtuaalimuistina. Virtuaalimuisti voi olla fyysistä muistia tai kiintolevyltä varattua tilaa. Tarkoituksena on piilottaa sovelluksilta muistin todellinen sijainti, jolloin käyttöjärjestelmä pystyy siirtämään fyysistä muistia levylle ja päinvastoin sovelluksen siitä tietämättä.

Muistilohkot, joita sovellukselle annetaan sen varatessa muistia, ovat oikeammin paloja *järjestelmän sivutiedostosta* (engl. system page file), jotka ovat liitetty prosessin virtuaalimuistiavaruuteen. Edellisessä aliluvussa oleva kuva 10 esittää tilannetta, jossa sivutiedostosta on liitetty varattuja muistilohkoja prosessin virtuaalimuistiavaruuteen.

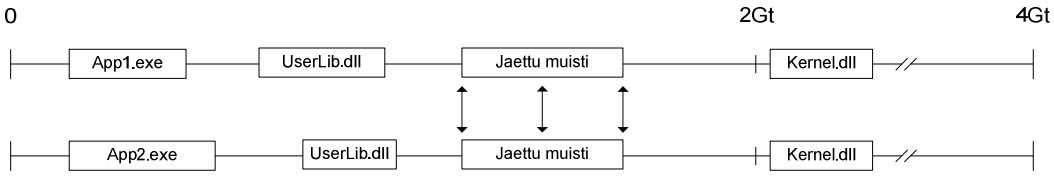
Koska pienien muutamien tavujen kokoisten lohkojen siirtäminen fyysisestä muistista levylle ja päinvastoin on hidasta, optimoivat käyttöjärjestelmät tätä pakottamalla pienimmäksi muistiblokin kooksi 4 kt. Esimerkiksi, jos sovellus varaa neljä tavua muistia, antaa käyttöjärjestelmä sille 4 kt. Seuraava sovelluksen tekemä muistinvaraus on nopeampi, sillä sovelluksella on jo muistia reservissä.

3.5 Jaetun muistin linkitys virtuaaliosoiteavaruuteen

Jaetun muistin käsittelyyn liittyvät hyvin vahvasti muistiosoitteet ja käytettävän järjestelmän muistiosoitteen koko. Kun kaksi prosessia jakaa muistialueen, tarkoittaa tämä sitä, että molempien prosessien virtuaaliosoiteavaruudessa on viittaukset jaettuun muistialueeseen.

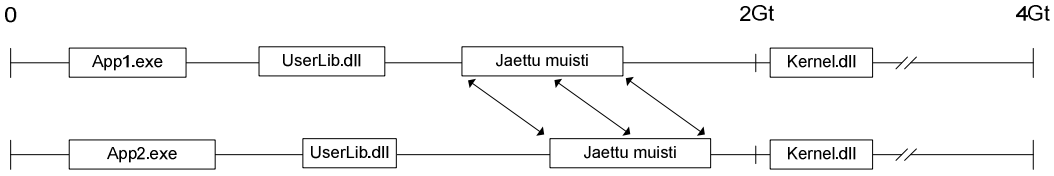
Windows NT käyttöjärjestelmissä on API-komento, jolla jaettu muisti saadaan linkitettyä prosessin virtuaaliosoiteavaruuteen alkaen halutusta muistiosoitteesta. Toinen vaihtoehto on antaa käyttöjärjestelmän linkittää jaettu muistialue johonkin prosessin vapaaseen kohtaan virtuaaliosoiteavaruudessa [JRi96]. Jälkimmäisessä tavassa on se heikkous, ettei prosessien kesken voida käyttää samoja muistiosoitteita, vaan on toimittava suhteellisilla osoitteilla. Esimerkiksi *Standard Template Libraryn* (STL, kirjasto tietorakenteita ja algoritmeja yleiskäyttöisessä muodossa) säiliöluokat käyttävät tietorakenteiden käsittelyyn suoraan muistiosoitteita, joten tällaista tietorakennetta ei voi sijoittaa jaettuun muistiin siten, että jaettu muisti olisi eri muistiosoitteissa kahden prosessin virtuaalimuistiavaruuksissa.

Seuraavassa kuvassa 11 jaettu muisti on liitetty prosessien virtuaaliosoiteavaruuteen käyttäen absoluuttista liittämistä.



Kuva 11. Jaettu muisti käyttäen absoluuttista linkitystä

Vaihtoehtoinen tapa, jossa annetaan käyttöjärjestelmän etsiä sopiva kohta prosessin virtuaaliosoiteavaruudesta jaettua muistia varten, on selvennetty kuvassa 12. Tällöin jaettu muisti voi sijaita eri osoitteissa kahden eri prosessin virtuaaliosoiteavaruuksissa.



Kuva 12. Jaettu muisti käyttäen suhteellista linkitystä

Tärkeäksi asiaksi 32-bittisissä järjestelmissä muodostuu virtuaaliosoiteavaruuden pienuus. Tästä johtuen teoreettinen jaetun muistin suurin mahdollinen koko on

$$S_{tot} = 2^{32} - S_{exe} - S_{dll,}$$

jossa S_{tot} on jaetun muistin suurin mahdollinen koko, S_{exe} on jaettua muistia käsittelevän ohjelman koko ja S_{dll} on jaettua muistia käyttävän ohjelman käyttämien kirjastojen koko. Lisäksi jäljelle jääneestä tilasta kuluu osoitteita vielä ohjelman pinoa varten riippuen käyttöjärjestelmästä. 32-bittisissä Windows NT järjestelmissä muistiosoitteet 0x80000000 - 0xFFFFFFFF sisältävät käyttöjärjestelmän komponentit, kuten ytimen, laiteajurit ym., joten käytettävä avaruus pienenee puoleen teoreettisesta suurimmasta koosta. Nämä seikat huomioon ottaen S_{tot} voi suurta jaetun muistin kapasiteettia vaativissa järjestelmissä olla liian pieni.

64-bittisessä käyttöjärjestelmässä virtuaalisoiteavaruus verrattuna 32-bittiseen järjestelmään kasvaa yli 4 miljardikertaiseksi. (Tätä tutkielmaa kirjoitettaessa 18 446 744 073 709 551 616 muistiosoitetta pitäisi olla riittävästi hieman laajempaakin järjestelmää varten. :-)

3.6 Muistin konsistenssimallit

Tähän mennessä on käsitelty kaikki perusasiat, joita tarvitaan jaetun muistin kautta kommunikointiin. Ensiksi käytiin läpi synkronisointi- ja erilaisia prosessien välisiä kommunikointitapoja, sekä lopuksi selvitettiin jaetun muistin luomiseen tarvittavia asioita muistinhallinnan näkökulmasta.

Seuraavaksi selvitetään muistin konsistenssimalleja, jotka vaikuttavat jaetun muistin käyttöön ja kerrotaan niiden aiheuttamista hyödyistä ja haitoista.

3.6.1 Taustaa konsistenssimalleista

Nykyaikaisissa tietokoneissa on käytössä monia eri muisteja kuten keskusmuisti, kiintolevyvälimuisti ja prosessorin L1- ja L2-välimuistit. Moniprosessorikoneessa jokaisella prosessorilla on omat välimuistinsa, jotta prosessoreiden ei tarvitsisi kilpailla samasta muistista. Tästä seuraa kuitenkin ongelmia, kun kaksi prosessoria käsittelee samaa dataa. Prosessorien pitää jollain keinoin saada tietoon toisen prosessorin tekemät laskutoimitukset.

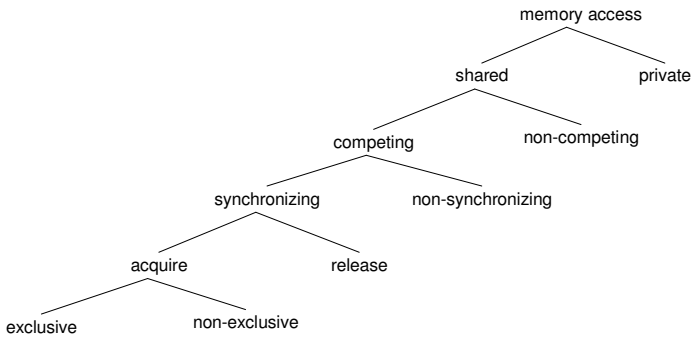
Moniprosessori- ja myös yksiprosessorijärjestelmissä, käytetään erilaisia muistinkonsistenssimalleja, jotta jaetut resurssit pysyvät konsistenttina eri säikeiden lukiessa ja kirjoittaessa niihin [DMo93].

Perinteisesti *muistin konsistenssimallit* (engl. Memory Consistency Models, MCM) ovat olleet laitevalmistajien ongelma-aluetta. Nykyään ne ovat siirtyneet myös ohjelmankehittäjien osaksi johtuen optimoinneista, joita laitevalmistajat ovat tehneet prosessoreihin kasvattaakseen prosessorien suorituskykyä. Nämä optimoinnit monimutkaistavat muistin konsistenssimallia ja tekevät ohjelmoinnin vaikeammaksi [DMo93].

Ensimmäinen kehitetty muistin konsistenssimalli oli *peräkkäinen konsistenssi* (engl. Sequential Consistency, SC). Tässä mallissa taataan, että minkä tahansa usean prosessorin suoritus on sama, kuin että prosessorien tekemät operaatiot ajettaisiin jossain tietyssä perättäisessä järjestyksessä, joka on määritelty ajettavassa ohjelmassa [DMo93]. Tämän kaltainen malli rajoittaa merkittävästi optimointimahdollisuuksia. Esimerkiksi järjestelmässä, jossa on hidas, korkealatenssinen muisti, olisi järkevää muodostaa liukuhihnamainen käsittely muistioperaatiolle, sekä käyttää puskureita muistin kirjoittamista varten.

Muistin konsistenssimalleja voidaan luokitella niiden *heikkouden* (engl. weakness) tai *tiukkuuden* (engl. strictness) mukaan. Heikoin mahdollinen malli antaa lukuoperaation paluuarvona jonkin aikaisemmin kirjoitetun arvon, tai arvon joka tullaan kirjoittamaan tulevaisuudessa [DMo93]. Tällainen malli on teoreettinen, sillä todellisuudessa on vaikeaa toteuttaa ohjelma, joka palauttaisi arvon tulevaisuudesta.

Mallien luokitteleminen voidaan tehdä esimerkiksi mallin muistinsaannin rajoittamiseen liittyvien attribuuttien määrän mukaan. Näitä attribuutteja voivat olla muistinsaannin sijainti, muistinsaannin suunta (luku, kirjoitus tai molemmat), muistinsaannissa sijoitettava arvo, muistinsaannin kausaalisuus ja muistinsaannin kategoria. Muistinsaannin kategorisointi voidaan tehdä kuvan 13 osoittamalla tavalla, joka on jatke [KGh90] esitellylle kategorisoinnille. Tämä jatke on määritelty David Mosbergerin artikkelissa Memory Consistency Models. [DMo93]



Kuva 13. Muistinsaantikategoriat [DMo93]

Muistinsaanti on joko jaettu tai yksityinen. Yksityiset muistinsaannit ovat triviaaleja käsitellä, joten jatketaan seuraavaksi jaettuihin muistinsaanteihin. Jaetut muistinsaannit voidaan kategorisoida *kilpaileviin* (engl. competing) ja *ei-kilpaileviin* (engl. non-competing) muistinsaanteihin. Muistinsaannit ovat kilpailevia, jos ne yrittävät pääsyä samaan paikkaan, vähintään yksi niistä on kirjoituspyyntö ja muistinsaannit eivät ole järjestettyjä. Esimerkiksi samanaikaisesti suoritetut jaetun muuttujan muistinsaannit käytettäessä kriittistä aluetta ovat ei-kilpailevia, koska molemminpuolinen poissulkeminen takaa saantien järjestyksen.

Kilpailevat muistinsaannit voidaan jakaa edelleen kahteen ryhmään, *synkronisoivaan* (engl. syncronizing) ja *ei-synkronisoivaan* (engl. non-syncronizing) [DMo93]. Synkronisoivaa muistinsaantia käytetään muistinsaantien järjestyksen pakottamiseen viivyttämällä muita saanteja niin pitkään, että meneillään oleva saanti on suoritettu loppuun.

Synkronisoivat muistinsaannit voidaan jakaa *omistussaanteihin* (engl. acquire) ja *vapautussaanteihin* (engl. release) [DMo93]. Omistussaanti liittyy aina luku-synkronisoivaan saantiin, kun taas vapautussaanti puolestaan liittyy kirjoitettavaan synkronisoivaan saantiin [DMo93]. Lopuksi omistussaanti voi olla *eksklusiivinen* (engl. exclusive) tai *ei-eksklusiivinen* (engl. non-exlusive) [DMo93]. Useita ei-eksklusiivisia saanteja voidaan sallia, mutta eksklusiivisia omistussaanteja viivytetään, kunnes kaikki aikaisemmat saannit ovat vapautettu. Esimerkiksi, jos kriittinen alue sisältää vain lukusaanteja jaettuihin muuttujiin, silloin lukko voidaan toteuttaa ei-ekslusiivisesti.

3.6.2 Atominen konsistenssi

Tiukin mahdollinen muistin konsistenssimalli on *atominen konsistenssi* (atomic consistency) [DMo93].

Atomisessa konsistenssimallissa jokainen suoritettava operaatio suoritetaan tiettynä *operaatiointervallina* [DMo93]. Operaatiointervalli voidaan ajatella jakamalla aikajana ei-päällekkäisiin aikalokeroihin, jonka sisällä operaatio suoritetaan. Atominen konsistenssi määrää, että kirjoitusoperaatio näkyy välittömästi kaikille prosessoreille.

Tarkemmat mallit atomisesta konsistenssista ovat *staattinen atominen konsistenssi* (engl. static atomic consistency) ja *dynaaminen atominen konsistenssi* (engl. dynamic atomic consistency) [DMo93]. Atominen konsistessimalli sallii samanaikaiset luku- ja kirjoitusoperaatiot, joista seuraa ongelmia, jos esimerkiksi samaa objektia käsitellään kahdessa prosessorissa samanaikaisesti. Staattinen atominen konsistenssimalli määrittää luku- ja kirjoitusoperaatioille oman ajanhetken, jolloin yhtaikaista luku- ja kirjoitusoperaatioita ei tapahdu. Dynaamisessa atomisessa konsistenssimallissa mikä tahansa luku- ja kirjoitusoperaatio voi tapahtua milloin tahansa.

3.6.3 Peräkkäinen konsistenssi

Leslie Lamport määritti *peräkkäisen konsistenssimallin* (engl. sequential consistency) vuonna 1979. Määrittely menee näin:

> ”…minkä tahansa suorituksen tulos on sama, kuin jos kaikkien prosessoreiden operaatiot suoritetaan jossain perättäisessä järjestyksessä ja jokaisen prosessorin operaatiot, jotka ovat tässä sekvenssissä, ajetaan siinä järjestyksessä kuin prosessorin ajama ohjelma ne määrittää.” [LLa79]

Toisin sanoen, järjestelmä tarjoaa peräkkäisen konsistenssimallin, jos kukin prosessori näkee kirjoitusoperaatiot samassa muistilohkossa samassa järjestyksessä. Peräkkäinen konsistenssi ei pakota operaatioita suoritettavan samassa järjestyksessä kuin ajettavissa ohjelmissa vaan sen, että jokainen prosessori näkee operaatiot samalla tavalla.

Peräkkäinen konsistenssimalli on heikompi kuin atominen konsistenssi. Atominen konsistenssi pakottaa muistioperaatioiden näkyvän kaikille prosessoreille täsmälleen silloin, kun ne prosessorin toimesta ajetaan.

3.6.4 Heikko konsistenssi

Heikkoa konsistenssimallia (engl. weak consistency) sanotaan tiukaksi hybridimalliksi. Konsistenssimallin esitteli M. Dubois vuonna 1986 [MDu86]. Järjestelmä tarjoaa heikon konsistenssimallin, jos seuraavat ehdot ovat voimassa:

1. Saannit synkronoituihin muistiobjekteihin ovat järjestettyjä,

2. saantia synkronoituun objektiin ei sallita ennen kuin kaikki aikaisemmat saannit ovat käsitelty ja

3. saantia mihinkään objektiin ei sallita, ennen kuin aikaisemmin saatu synkronointiobjekti on käsitelty.

Tämä malli takaa sen, että kun saantia ollaan suorittamassa, aikaisemmat saannit on suoritettu ja sen, ettei tulevia saanteja vielä ole suoritettu. Synkronoituja muistisaanteja voidaan ajatella eräänlaisina *aitoina* (engl. fence). [DMo93]

Käytännössä tämä malli takaa, että kaikki synkronoituihin objekteihin kohdistuvat saannit näkyvät kaikille prosessoreille samassa järjestyksessä. Saannit muihin kuin synkronoituihin objekteihin voivat näkyä eri järjestyksessä.

3.6.5 Vapautuskonsistenssi

Vapautuskonsistenssin (engl. release consistency) määritteli Kourosh Gharachorloo heikon konsistenssimallin tarkennukseksi [KGh90]. Vapautuskonsistenssissa kilpailevat saannit määritellään omistaviksi (engl. acquire), vapauttaviksi (engl. release) ja synkronoimattomiksi (engl. non-synchronizing). Kilpaileville saanneille on lisäksi annettu oma nimitys spesiaali (engl. special), jotta ne voidaan helposti erottaa ei-kilpailevista operaatioista [DMo93].

Heikossa konsistenssimallissa on se ongelma, ettei siinä tiedetä ollaanko synkronoitua objektia lukemassa vai kirjoittamassa. Jälkimmäisiä operaatioita viivästetään turhaan, kun jokin prosessori on tekemässä lukuoperaatiota synkronoituun objektiin.

Vapautuskonsistenssimallissa omistussaanti toimii kuten synkronoitu saanti heikossa konsistenssimallissa, paitsi että aita viivästyttää ainoastaan tulevia saanteja. Vapauttavat saannit puolestaan toimivat samalla tavalla kuin heikon konsistenssimallin vapautussaannit, mutta aita viivästyttää tulevia operaatioita kunnes aikaisemmat saannit on saatu käsiteltyä.

Tällä sallitaan osittainen päällekkäisyys kriittisten alueiden välillä, jota heikko konsistenssimalli ei salli [DMo93].

Järjestelmän sanotaan toteuttavan vapautuskonsistenssimallin silloin, kun yhden prosessorin tekemät kirjoitusoperaatiot nähdään toisissa prosessoreissa sen jälkeen, kun ensimmäinen vapauttaa objektin ja ennen kuin jälkimmäiset tekevät objektille omistussaannin.

3.7 Transaktiomuisti

Transaktiomuisti (engl. transactional memory) on mekanismi, jolla kontrolloidaan pääsyä jaettuihin muistialueisiin monisäikeisissä ohjelmissa [MHe93].

Transaktiomuistin periaate esiteltiin ensimmäisen kerran vuonna 1993 Maurice Herlihyn ja J. Eliot B. Mossin toimesta. Transaktiomuistin pääperiaatteena on toteuttaa jaetun muistialueen pääsy optimistisena [MHe93]. Jokainen jaettua muistialuetta käsittelevä säie muokkaa muistialuetta muista säikeistä riippumatta. Toisin sanoen menetelmä on lukkovapaa, kts. luku 2.2.5. Menetelmä toimii transaktiomallin mukaisesti, joten kunkin säikeen tekemät muokkaukset näkyvät toisille säikeille vasta kun transaktio vahvistetaan.

Jos kaksi rinnattaista säiettä muokkaa samaa jaettua muistilohkoa samaan aikaan, havaitaan tämä siinä vaiheessa kun transaktiota ollaan vahvistamassa. Tällöin käy niin, että molemmat transaktiot aloitetaan alusta ja tätä suoritetaan niin kauan, että vahvistaminen onnistuu.

Herlihyn ja Mossin menetelmässä transaktiomuistit vaativat prosessorilta tuen seuraaville prosessorin tukemille toiminnoille:

- *Lue-transaktionaalinen* (engl. load-transactional, LT) lukee arvon jaetusta muistilohkosta privaattiin rekisteriin.
- *Lue-transaktionaalinen-poissulkeva* (engl. load-transactional-exlusive, LTX) lukee arvon jaetusta muistilohkosta privaattiin rekisteriin samalla antaen vihjeen, että muistilohko tullaan todennäköisesti päivittämään.

- *Tallenna-transaktionaalinen* (engl. store-transactional, ST) yrittää kirjoittaa arvon privaatista rekisteristä jaettuun muistilohkoon. Tämä uusi arvo ei tule näkymään muille säikeille, ennen kuin transaktio onnistuneesti vahvistetaan.

Lisäksi transaktiot vaativat seuraavat toiminnot, joilla manipuloidaan transaktion tilaa:

- *Vahvistus* (engl. commit) yrittää tehdä ST:n muutoksista pysyviä. Se onnistuu ainoastaan silloin kun yksikään muu transaktio ei ole päivittänyt transaktion yhtäkään muistialuetta, eikä yksikään muu transaktio ole lukenut yhtäkään transaktion kirjoitettavista muistilohkoista. Jos vahvistus onnistuu, transaktion tekemät muutokset tulevat näkyviksi muille säikeille.

- *Peruminen* (engl. abort) hylkää kaikki transaktion vahvistamattomat muutokset.

- *Validointi* (engl. validate) testaa senhetkisen transaktion tilan. Tällä voidaan tarkistaa, onko transaktio peruttu vai ei.

Näillä toiminnoilla voidaan toteuttaa mikä tahansa lue-muokkaa-kirjoita -operaatio jaettuun muistilohkoon.

Herlihyn ja Mossin mukaan transaktiomuisteilla saadaan rinnattaisessa laskennassa reilusti parempia tuloksia kuin molemminpuolista poissuljentaa käyttävillä algoritmeilla, kun käytössä on moniprosessorikone, jossa prosessoreiden määrä on yli neljä kappaletta. [MHe93]

Molemminpuolista poissuljentaa käyttävien algoritmien heikkoutena on ylimääräinen prosessointi, jota suoritetaan lukinnoissa silloinkin, kun niitä ei tarvittaisi. Jaettua muistilohkoa käsiteltäessä uhrataan koneen resursseja lukintoihin, vaikka sillä hetkellä ei olisi vaaraa, että toinen säie tulisi samaan aikaan käsittelemään samaa muistilohkoa.

4 Sovellusesimerkki

Tutkielman empiirisessä osassa haetaan sovellusesimerkin avulla tukea teoriaosassa esiteltyihin asioihin. Testeissä vertaillaan kahden prosessien välisen kommunikointimenetelmän tehokkuutta. Samalla myös paneudutaan näiden menetelmien käytännön ongelmakohtiin.

Mittaustuloksia varten tehdään kaksi ohjelmaa, joilla simuloidaan datanvälitystä prosessien välillä. Ohjelmilla on tarkoitus simuloida kahta eri kommunikointimenetelmää ja mitata niiden suorituskykyarvot samalla datamäärällä. Näin saadaan vertailukelpoiset mittaustulokset, joista voidaan arvioida kommunikointimenetelmän tehokkuutta.

Seuraavaksi kerrotaan lisää simulaattorista ja tutkittavista asioista.

4.1 Datanhajautussimulaattori

Datanhajautussimulaattorin tehtävänä on jakaa dataa usealle prosessille niin nopeasti kuin asiakasprosessit ehtivät dataa vastaanottaa. Kun sama testi tehdään eri kommunikointimenetelmillä, saadaan kulutetun prosessoriajan perusteella selville kunkin menetelmän tehokkuus.

Prosessit, jotka vastaanottavat dataa, ovat nimeltään asiakasprosesseja. Asiakasprosessit ottavat dataa vastaan niin nopeasti kuin sitä vain on saatavilla. Kuitenkin kaikki data käsitellään, eli dataa ei jätetä väliin, jos sitä ei ehditä käsitellä. Tästä muodostuu vuonhallinta datan tuottajan ja vastaanottajan välille. Datan lähettäjä lähettää uuden datan vasta, kun kaikki asiakasprosessit ovat käsitelleet jo lähetetyn datan.

Ohjelmien kommunikointi on rajattu saman mikrotietokoneen sisälle, koska käytetyn käyttöjärjestelmän muistiavaruus ei näy toiselle mikrotietokoneelle ilman sitä tukevia ohjelmia.

4.2 Tutkittavat asiat

Simulaattori rakennetaan siten, että sillä voidaan tutkia siirtoväylän tuomia rajoitteita datanvälitykseen. Ensisijainen tutkimuskohde on siirtoväylän siirtämiskyky ja siirtoväylän

hyötykuorma prosessoriaikaan nähden. Tärkeänä tutkimuskohteena pidetään myös prosessien määrän kasvamisesta tulevaa vaikutusta kokonaissiirtokapasiteettiin ja prosessorikuormaan. Kontention osuuden pitäisi kasvaa, kun kommunikoivien prosessien määrää kasvatetaan.

Toisarvoisina tutkimuskohteina pidetään siirtomenetelmän helppokäyttöisyyttä, vikasietoisuutta ja rakenteen monimutkaisuutta. Näillä arvioinneilla ei ole merkitystä kommunikointimenetelmän paremmuuteen, vaan tämä empiirinen tutkimus antaa arvoa sille menetelmälle, jolla dataa saadaan nopeimmin siirrettyä prosessilta toiselle.

Mittaustuloksissa kiinnitetään huomioita eri kommunikointimenetelmien eroihin, sillä absoluuttiset mittaustulokset indikoisivat vain testilaitteiston tehokkuutta.

Tutkittavat asiat on jaoteltu seuraaviin alalukuihin. Kussakin luvussa selvennetään mitä mitataan, miten mitattava asia mitataan ja lopuksi kerrotaan olettamuksia mittaustuloksista.

4.2.1 Keskimääräinen tavujen määrä sekunnissa

Ensimmäinen tutkittava kohde on siirtoväylän kapasiteetti siirtää dataa prosessilta toiselle. Mitattava suure on tavua sekunnissa, jolla nähdään siirtoväylän kulloinenkin nopeus.

Mittaus suoritetaan tarkastelemalla siirtoväylään kirjoitettua / siirtoväylältä vastaanotettua datamäärää ja vertaamalla datamäärää kuluneeseen aikaan. Tästä voidaan laskea tarkasti keskimääräinen tavujen määrä sekunnissa. Siirron alkukellonaika merkitään talteen sillä hetkellä, kun siirtoväylään kirjoitetaan ensimmäisen kerran tai kun vastaanottavassa päässä vastaanotetaan siirtoväylästä dataa ensimmäisen kerran. Tunnuslukuja tallennetaan tiedostoon sekunnin välein.

4.2.2 Tavujen määrä sekunnissa

Tämä kohde mittaa hetkellistä tavujen määrää sekunnissa.

Mittausarvo saadaan jakamalla siirretty tavumäärä siihen kulutetulla ajalla. Arvoja tallennetaan sekunnin välein, joten laskeminen tapahtuu jakamalla siirretty tavumäärä mittausajanhetkien välisellä ajalla.

4.2.3 Viestien määrä sekunnissa

Viestien määrällä tarkoitetaan lähetettyjä / vastaanotettuja viestejä sekunnissa. Yhdellä viestillä tarkoitetaan yksittäisen datalohkon lähetystä vastaanottajalle. Kasvattamalla tai pienentämällä datalohkon kokoa saadaan yhdellä kertaa välitettävän datan kokoa kasvatettua tai pienennettyä. Käyttämällä suurta datalohkon kokoa pitäisi teoriassa kontention osuus pienentyä, sillä molemminpuolisen poissuljennan aiheuttamaa jätekuormaa pitäisi syntyä vähemmän.

Viestien määrä sekunnissa riippuu asiakkaiden määrästä ja datalohkon koosta.

4.2.4 Kontentio sekunnissa

Kontentio sekunnissa tarkoittaa lähetyspäässä sitä odotusaikaa, joka odotetaan kunnes lähetettävä data päästään siirtämään siirtoväylälle yhden sekunnin aikana. Vastaanottopäässä kontentio sekunnissa tarkoittaa sitä aikaa, joka odotetaan kunnes päästään siirtämään dataa pois siirtoväylältä.

4.2.5 Asiakkaiden määrä

Asiakkaiden määrää mitataan vain lähettävässä päässä. Tällä tarkoitetaan dataa vastaanottavien prosessien määrää.

4.3 Testisovelluksien rakenne

Testausta varten tehdään kaksi erilaista sovellusta, jotka osaavat kommunikoida keskenään. Toinen sovellustyyppi osaa jakaa dataa ja toinen vastaavasti vastaanottaa sitä.

Ohjelmat esitellään seuraavissa aliluvuissa ensin prosessitasolla ja sen jälkeen tarkemmin komponenttitasolla.

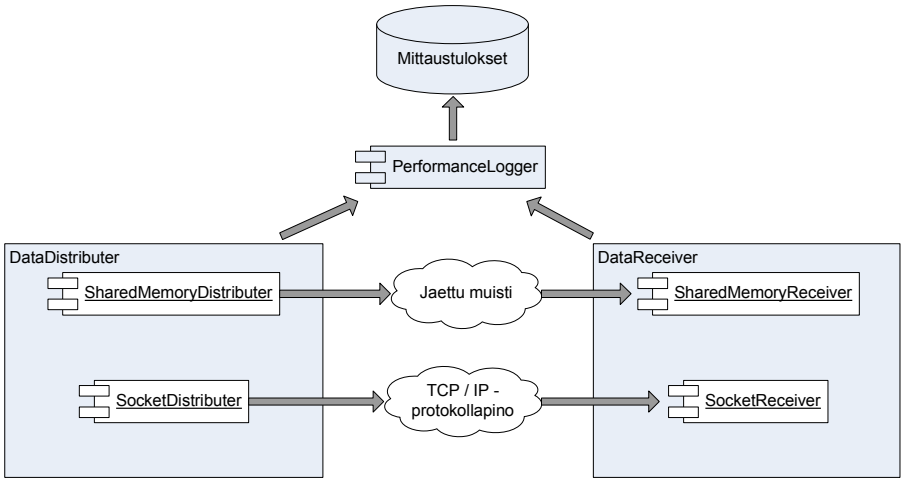
4.3.1 DataDistributer

DataDistributer on prosessi, jonka on tarkoitus toimia palvelintyyppisenä datanhajautussimulaattorina.

DataDistributer-prosessissa on kaksi tilaa, alustustila, jossa alustetaan siirtoväylä ja odotetaan asiakkaiden rekisteröitymistä, sekä varsinainen datanjakamistila, jossa dataa jaetaan. Prosessin suorituskykyä mitataan vasta datanjakamistilassa. Näin saadaan eliminoitua mittaustuloksista alustuksesta johtuvat viiveet prosessin käynnistyksen ja asiakkaiden rekisteröitymisen aikana.

Prosessia voidaan ajaa kahdella eri tavalla, joko simuloiden jaetun muistin kautta tapahtuvaa kommunikointia, tai käyttäen siirtotienä verkkosanomia. Kummatkin kommunikointitavat on toteutettu omiin komponentteihinsa samalla rajapinnalla, joten ohjelman liikennettä ohjaavan komponentin toiminta on samanlainen kummallakin prosessien välisellä kommunikointimenetelmällä.

Seuraavassa kuvassa 14 esitellään DataDistributerin sisäiset komponentit:



Kuva 14. Simulaattorin komponenttirakenne

PerformanceLogger-komponenttia käytetään prosessin suorituskyvyn mittaamiseen sekä DataDistributerissa, että DataReceiverissä.

Asiakkaiden rekisteröitymistä varten DataDistributerissa on molemmille kommunikointimenetelmille oma tapansa. Käytettäessä jaettua muistia rekisteröinti tapahtuu erillisen rekisteröintimuistin kautta, jonne asiakkaat laittavat oman yksilöllisen tunnisteensa. Käyttäjän on kerrottava DataDistributerille datan vastaanottajien lukumäärä, jotta DataDistributer tietää odottaa tarvittavaa määrää asiakkaita. Kun kaikki asiakkaat ovat rekisteröityneet, aloittaa DataDistributer datan jakamisen. Verkkosanomapohjaisessa kommunikoinnissa rekisteröityminen tapahtuu verkkosanomien kautta, mutta logiikka on sama.

4.3.2 SharedMemoryDistributer

Jaetun muistin kautta tapahtuvan kommunikoinnin datanjakaja on toteutettu SharedMemoryDistributer-komponenttiin.

Komponentti luo kommunikointiin tarvittavat objektit, jaetun muistin sekä signaloimiseen tarvittavat tapahtumasignaalit, joilla kerrotaan ohjelman tiloja toisille prosesseille. Jaettu muisti luodaan kappaleen 3.5 kertomalla tavalla siten, että useat prosessit saavat saman muistialueen näkymään omaan virtuaaliavaruuteensa. Jaetun muistin koko määräytyy paketin koosta niin, että yksi paketti mahtuu kokonaisuudessaan kerralla jaettuun muistiin. Tapahtumasignaaleja luodaan kaksi kutakin datanvastaanottajaa kohti. Ensimmäisellä signaalilla kerrotaan DataReceiverille uuden paketin saapuneen jaettuun muistiin. Toista signaalia käytetään vastakkaiseen suuntaan. DataReceiver kertoo, että paketti on luettu jaetusta muistista.

Jaetunmuistin kautta tapahtuvan datanjakamisen algoritmin voi yksinkertaistaa muutamaan ydinkohtaan:

```
1. Kopioidaan jaettuun muistiin uusi paketti
2. Resetoidaan DataReceiverien valmissignaali
3. Asetetaan DataReceiverien herätyssignaalit signaloitu-tilaan
```

```
4. Odotetaan DataReceiverien valmissignaaleja

5. Takaisin kohtaan 1.
```

4.3.3 SocketDistributer

Toinen vertailtavana oleva prosessien välinen kommunikointimenetelmä on toteutettu SocketDistributer-komponenttiin.

Soket-liikennettä varten SocketDistributer avaa kuuntelevan portin 10000, johon SocketReceiverit ottavat yhteyden rekisteröimisvaiheessa. Soket-liikennöintiä varten käytetään IP-osoitetta 127.0.0.1, jolloin Windowsin Winsock2-kirjastolle kerrotaan, että vastapää on samassa koneessa ja kirjasto pystyy näin optimoimaan kommunikointia, koska liikennettä ei tarvitse ohjata TCP / IP -protokollapinon fyysisen kerroksen läpi.

Jokaisella SocketReceiverillä on yksilöllinen TCP-yhteys SocketDistributeriin. SocketDistributer lähettää paketin jokaiselle TCP-yhteydelle erikseen. Jokaisen paketin lähetyksen jälkeen odotetaan SocketReceiveriltä kuittaussanomaa takaisin, jotta tiedetään, että jokainen SocketReceiver on saanut vastaanotettua paketin. Seuraavan paketin lähetys alkaa vasta sitten, kun jokainen SocketReceiver on lähettänyt SocketDistributerille kuittaussanoman.

Sokettien kautta tapahtuvan prosessien välisen kommunikoinnin algoritmi voidaan pelkistää seuraavasti:

```
1. Luodaan uusi paketti

2. Lähetetään paketti jokaiselle SocketReceiverille omaa TCP-yhteyttä
   pitkin

3. Odotetaan, että jokaiselta SocketReceiveriltä on tullut
   kuittaussanoma

4. Takaisin kohtaan 1.
```

4.3.4 DataReceiver

Datan vastaanottajana toimii prosessi nimeltä *DataReceiver*.

Prosessi ottaa dataa vastaan joko jaetun muistin kautta tai verkkosanomien kautta. Molemmissa tavoissa pääperiaatteena on vastaanottaa data ja lähettää kuittaus, jotta DataDistributer voi lähettää seuraavan datan.

DataReceiver on jaettu kahteen toiminnalliseen osaan kuten DataDistributerkin. Kun DataDistributer käynnistetään, menee se ensin alustustilaan. Tässä tilassa se luo yhteyden siirtoväylään ja rekisteröi itsensä DataDistributerille. Tämän jälkeen DataReceiver jää odottamaan dataa DataDistributerilta.

Suoritusarvojen mittaukseen käytetään PerformanceLogger-komponenttia samalla tavalla kuin DataDistributerissakin. Seuraavassa luvussa kerrotaan tarkemmin PerformanceLogger-komponentin toiminnasta.

Käyttäjän on kerrottava DataReceiverille siirtoväylän tyyppi, jotta prosessi osaa rekisteröityä ja odottaa dataa oikealla tavalla. DataReceiver lähettää asetetulla kommunikointimenetelmällä oman yksilöllisen tunnisteen DataDistributerille, jotta DataDistributer osaa yksilöidä datan vastaanottajat oikein.

4.3.5 SharedMemoryReceiver

SharedMemoryReceiver on SharedMemoryDistributer-komponentin pari, joka vastaanottaa dataa jaetusta muistista.

SharedMemoryReceiver liittää SharedMemoryDistributerin tekemän jaetun muistin omaan virtuaaliosoiteavaruuteen, sekä ottaa käyttöön tapahtumasignaalit, joiden avulla synkronisoidaan SharedMemoryDistributerin kirjoittaminen ja SharedMemoryReceiverin lukeminen. SharedMemoryReceivereitä voi olla useita rinnattain ajossa. Tällöin SharedMemoryReceiverit voivat lukea jaetusta muistista dataa samaan aikaan, mutta tästä ei muodostu ongelmia kuten luvussa 3.6 kerrottiin. Kirjoittaminen ja lukeminen jaettuun muistiin on aina synkronisoitava jollain menetelmällä, tai muuten jaetusta muistista lukevan prosessin luettu data voi olla korruptoitunutta.

SharedMemoryReceiverin algoritmi jaetusta muistista lukiessa voidaan esittää seuraavasti:

```
1. Odotetaan signaalia SharedDataDistributerilta, että paketti on
   jaetussa muistissa

2. Luetaan paketti jaetusta muistista

3. Signaloidaan SharedDataDistributerille, että paketti on luettu

4. Takaisin kohtaan 1.
```

4.3.6 SocketReceiver

SocketReceiver-komponentti muodostaa TCP-yhteyden DataDistributerissa sijaitsevaan SocketDistributer-komponenttiin.

SocketReceiverin datan vastaanottaminen on hieman erilainen verrattuna SharedMemoryReceiveriin, sillä paketin siirtämiseen tarvitaan väliaikaisia puskureita. Kun SocketDistributer lähettää paketin verkkoon, kopioi Winsock2-kirjasto paketin TCP / IP -protokollapinon sisäiseen siirtopuskuriin. Tämän jälkeen Winsock2-kirjasto purkaa puskurin sisällön SocketReceiverin lokaaliin muistiavaruuteen. Tässä menetelmässä on yksi muistikopiointi enemmän verrattuna jaetun muistin kautta tapahtuvaan kommunikointiin.

SocketReceiverin algoritmi datan vastaanottamiseen SocketDistributerilta voidaan kertoa seuraavasti:

```
1. Odotetaan verkkosanomaa SocketDistributerilta

2. Kopioidaan vastaanotettu paketti lokaaliin muistiin

3. Lähetetään kuittaussanoma SocketDistributerille

4. Takaisin kohtaan 1.
```

4.3.7 PerformanceLogger

PerformanceLogger on irrallinen komponentti, joka tarjoaa joukon rajapintoja, joilla komponenttia käyttävä olio voi merkitä suorituskykyarvoja mitattavaksi.

PerformanceLogger-komponentti on suunniteltu siten, että se häiritsee mahdollisimman vähän varsinaisen simulaattorin suoritusta. PerformanceLogger laskee sekunnin välein suorituskykyarvot ja merkitsee nämä tiedostoon. Suorituskyvyn laskenta on toteutettu

omassa säikeessään, joten laskennalla ei suoranaisesti ole vaikutusta työsäikeen toimintaan. Todellisuudessa PerformanceLoggerissa tapahtuva laskenta suoritetaan tietokoneen prosessoriajalla, joten pieni hidastava vaikutus suorituskyvyn laskennalla kuitenkin on yksiprosessorikoneessa. Moniprosessorikoneessa PerformanceLoggerin suoritus ajettaisiin eri prosessorilla, joten se ei hidastaisi varsinaisen työsäikeen ajamista, tosin prosessoriaika olisi kuitenkin pois joltain toiselta samassa tietokoneessa olevalta säikeeltä. PerformanceLoggerin prosessoriajankulutus tallennetaan tiedostoon, joten se voidaan ottaa jälkianalyysissä huomioon.

Komponentin tarjoamia datankirjausrajapintoja ovat:

StartLogging,

kerrotaan komponentille testin alkavan

StopLogging,

kerrotaan komponentille testin loppuvan

AccessTransmissionConduitBegin,

kerrotaan komponentille siirtoväylän käsittelyn alkaneen

AccessTransmissionConduitEnd,

kerrotaan komponentille siirtoväylän käsittelyn päättyneen

MessageProcessingBegin,

kerrotaan komponentille viestin odotuksen alkavan

MessageProcessingEnd,

kerrotaan komponentille viestin odotuksen päättyvän

WaitPeerBegin,

kerrotaan komponentille vastapään odottamisen alkavan

WaitPeerEnd,

kerrotaan komponentille vastapään odottamisen loppuvan

UpdateDataProcessed,

kerrotaan komponentille käsitellyn datalohkon koko

Suhteellisten kellonaikojen laskentaan käytetään Windowsin tarjoamaa QueryPerformanceCounter-apikutsua, jolla saadaan kyseltyä tietokoneen laitteistolta korkearesoluutioisia ajastimen arvoja. QueryPerformanceFrequency-apikutsu kertoo ajastimen laukeamistaajuuden, jonka jälkeen tiedetään, mitä QueryPerformanceCounterin palauttama arvo tarkoittaa ajassa. [WHDC]

Tietokoneella, jolla testit ajettiin, on käytössä ACPI-laitteistoajastin, jonka päivitystaajuus on 3,57MHz. Tämä asettaa varsinaisen ajastimen resoluutioksi 0,28 mikrosekuntia. Vaikkakin ajastimen resoluutio on reilusti alle mikrosekunnin, ei tätä tarkkuutta saada käyttöön sovelluksissa, sillä ajastimen lukemiseen kuluva aika on myös otettava huomioon. Testejä ajettavalla tietokoneella ajastimen lukemiseen kului aikaa 1,7 mikrosekuntia, joka saatiin selvitettyä testaamalla.

Ajastimelta pyydettiin perättäin kaksi arvoa, joista ensimmäinen vähennettiin toisesta. Näin saatiin selvitettyä yksittäisen ajastimen kyselyyn kuluva aika. Tätä toistettiin 1000 kertaa, jotta saatiin eliminoitua yksittäisten virheiden vaikutus tulokseen. Tämän tutkielman empiiriseen testiin 1,7 mikrosekunnin resoluutio riittää vallan mainiosti, joten tarkempia menetelmiä suhteellisien aikojen mittaamiseen ei lähdetty etsimään.

4.4 Testausympäristö

Testit ajettiin kannettavassa mikrossa siten, ettei samanaikaisesti ole käynnissä muita aktiivisia prosesseja, jotka kuormittaisivat tietokoneen prosessoria, keskusmuistia tai I/O-

väylää. Näin varmistetaan se, että tietokoneesta saadaan testien aikana hyödynnettyä kaikki teho vain ja ainoastaan testejä varten.

Tietokoneen laitteiston testeihin vaikuttavat tarkat tiedot on koottu seuraavaan taulukkoon:

Osa	Seloste
Käyttöjärjestelmä	Microsoft Windows XP SP2
Prosessori	Intel ® Pentium ® M 1,86 GHz
Keskusmuisti	Infineon 1 GB DDR2 533MHz
Emolevyn piirisarja	Mobile Intel 915PM Express Chipset

Taulukko 3. Testilaitteiston tiedot

4.5 Mittaustulokset

DataDistributerin ja DataReceiverin tulokset kerättiin Microsoft Excel -arkkeihin, joissa tuloksista jalostettiin tunnuslukuja. Kaikki tunnusluvut on laskettu keskiarvoistamalla mitattuja suorituskykyarvoja. Näin on saatu eliminoitua pienet heilahtelut simulaattorin toiminnassa.

Seuraavissa alaluvuissa käydään läpi mittauksista saatuja tuloksia.

4.5.1 Simulaattorin tunnuslukuja

Seuraavaan taulukkoon 4 on koottu molempien menetelmien suorituskykyarvot. Paketilla tarkoitetaan DataReceiverielle jaettavaa datalohkoa. Simulaattorissa käytetyn paketin koko oli 512 kilotavun kokoinen. Siirtoväylän käyttöaste kertoo prosentteina kuinka suuri osa prosessin testin aikana olleesta ajasta on kulunut siirtoväylän käsittelyyn.

Kontentiolla tarkoitetaan sitä aikaa prosentteina, joka on kulunut vastapään odottamiseen ennen kuin seuraavaa pakettia päästään käsittelemään. Paketin käsittely -sarakkeeseen on

merkitty prosessin kuluttama aika, joka on kulunut pelkästään paketin käsittelyyn. Tällä tarkoitetaan sitä aikaa, jonka prosessi on kuluttanut paketin siirtämiseen siirtoväylälle tai pois sieltä suhteessa koko simulaation suorittamiseen kuluneeseen aikaan.

CPU-ajankulutus kertoo prosessin kuluttaman prosessoriajan suhteessa koko koneen prosessoriaikaan. DataReceiverien kohdalla CPU-ajankulutussarakkeeseen on merkitty sulkeisiin kaikkien kolmen DataReceiverien yhteenlaskettu suhteellinen prosessoriaika.

Komponentti	**Paketteja (á 512 Kt) / s**	**Siirtoväylän käyttöaste (%)**	**Kontentio (%)**	**Paketin käsittely (%)**	**CPU-ajankulutus (%)**
DataDistributer (jaettu muisti)	680	52,9	46,5	0,25	33,1
DataDistributer (soket)	34,6	99,2	0,11	0,02	28,5
DataReceiver (jaettu muisti)	680	42,8	56,9	0,00	21,7 (65,1)
DataReceiver (soket)	34,6	2,80	94,3	1,49	9,53 (28,6)

Taulukko 4. Simulaattorin suorituskykyarvot

Testin tulokset kertovat, että jaetun muistin kautta tapahtuva kommunikointi on 20 kertaa nopeampaa kuin sokettipohjainen kommunikointi. Koska yhden paketin koko oli 512 kilotavua, saadaan ensimmäiselle siirtomenetelmälle datansiirtonopeudeksi 340 Mt / s ja jälkimmäiselle menetelmälle 17,3 Mt / s.

Siirtoväylän operointi vei noin puolet jaettua muistia käyttävästä DataDistributerin ajasta, kun taas sokettiväylää käyttävältä jakajalta se vei melkein kaiken ajan. Tästä voidaan päätellä, että jaetun muistin kautta tapahtuvassa kommunikoinnissa olisi vielä optimoitavaa,

sillä DataDistributer oli 46,5 % ajasta odottamassa vastapäiden toimintaa. Sen sijaan soketti-pohjaisessa kommunikoinnissa DataDistributer ei enää juurikaan olisi voinut toimia nopeammin. Molempien datanjakajien viestin prosessointiin kului sama vakiomittainen aika, sillä prosessoinnilla tarkoitetaan uuden paketin muistikopiointia lähetyspuskuriin. Jaettua muistia käyttävällä DataDistributerilla paketin käsittelyyn kuluu suhteessa koko simulaatioon suurempi aika, koska paketteja kulki suurempi määrä sekunnissa verrattuna soketti-pohjaiseen DataDistributeriin.

Datan vastaanottopäässä jaettua muistia käytettäessä kontention osuus oli vielä suurempi (56,9 %) kuin lähetyspäässä. Tämä on loogista, sillä datanjakajalla on enemmän työtä, koska sen on tarkasteltava useiden datan vastaanottajien signaaleja.

Sokettia käyttävän DataReceiverin siirtoväylän käyttöaste on alhainen, koska TCP-yhteydestä ei saatu tietoa siitä, milloin soketti on vastaanottamassa ja milloin odottamassa dataa. Datan vastaanottaminen ei kuluttanut DataReceiverin prosessin prosessoriaikaa vaan työ tehtiin käyttöjärjestelmän ydinprosessilla nimeltä System.

Siirtoväylän käyttöaste -sarakkeeseen on laskettu se osa siirtoprosessista, jolloin soketissa on dataa ja se kopioidaan vastaanottopuskuriin.

4.5.2 Siirtomenetelmien tehokkuus

Menetelmillä oli pelkästään siirtonopeutta katsottaessa suuri ero. Seuraavaan taulukkoon 5 on laskettu siirtomenetelmille tehokkuuskerroin, joka kertoo menetelmän hyötysuhteen prosessoriaikaan nähden. Tehokkuuskerroin on laskettu jakamalla siirtonopeus siirtoon kuluvalla prosessoriajalla.

Testi tehtiin käyttämällä 512 kilotavun kokoisia paketteja ja kolmea DataReceiveria.

Menetelmä	Siirtonopeus (Mt / s)	CPU aika (s)	Tehokkuuskerroin (Mt / CPU s)
DataDistributer (jaettu muisti)	340	0,338	1006
DataDistributer (soket)	17,3	0,285	60.8
DataReceiver (jaettu muisti)	340	0,223	1528
DataReceiver (soket)	17,3	0,101	171

Taulukko 5. Menetelmien tehokkuus

Tuloksista nähdään, että jaetun muistin kautta tapahtuva kommunikointi on dataa lähetettäessä noin 16 - 17 kertaa tehokkaampi kuin sokettien kautta tapahtuva kommunikointi. Vastaanottopäässä ero oli noin yhdeksänkertainen.

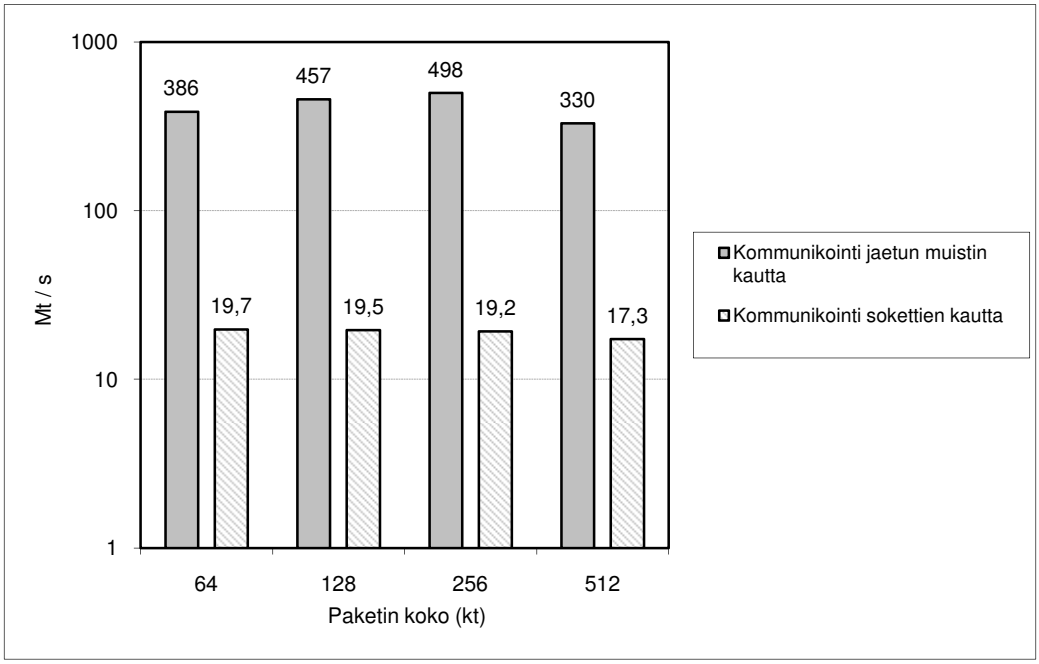
4.5.3 Paketin koon vaikutus siirtonopeuteen

Lopuksi suoritettiin testejä, joissa katsottiin kuinka simulaattorin yksittäisen siirrettävän paketin koko vaikuttaa kummankin siirtomenetelmän nopeuteen. Testit tehtiin kolmella DataReceiverillä. Testissä mitattiin myös kontentiota, sillä paketin koon kasvaessa, pitäisi siirtomenetelmien hyötykuorman kasvaa.

Paketin koko (kt)	Kommunikointi jaetun muistin kautta		Kommunikointi soketin kautta	
	Siirtonopeus (Mt / s)	Kontentio (%)	Siirtonopeus (Mt / s)	Kontentio (%)
64	386	65,4	19,7	0,42
128	457	51,9	19,5	0,23
256	498	52,2	19,2	0,16
512	330	46,5	17,3	0,11

Taulukko 6. Paketin koon vaikutus siirtonopeuteen

Seuraava kaavio selventää kommunikointimenetelmien siirtonopeuksien muutoksia paketin koon muuttuessa.



Kaavio 1. Paketin koon vaikutus siirtonopeuteen

Tässä testistä kävi ilmi, että paketin koolla on selvästi vaikutusta siirtomenetelmien tehokkuuteen. Jaetun muistin kautta tapahtuvassa kommunikoinnissa löytyi optimaalinen paketin koko, jossa siirtonopeus oli korkeimmillaan. Kontention osuus putosi sitä mukaan mitä suuremmalla paketilla dataa siirrettiin prosessien välillä. Siirtyminen 256 kilotavun paketista 512 kilotavun pakettiin pudotti siirtonopeutta reilusti, vaikka kontentionkin osuus putosi. Tähän voi olla syynä paketin kopioimisen hidastuminen jaettuun muistiin, kun paketin koko käy suuremmaksi. On myös mahdollista, että käyttöjärjestelmä ei ole pitänyt jaetussa muistissa olevaa dataa keskusmuistissa, vaan paketti on sijainnut osittain käyttöjärjestelmän virtuaalimuistissa massamuistilla. Jaetun muistin olisi voinut toteuttaa siten, että käyttöjärjestelmä olisi pakotettu pitämään paketti koko ajan keskusmuistissa, mutta näin ei simulaattoria toteutettaessa huomattu tehdä.

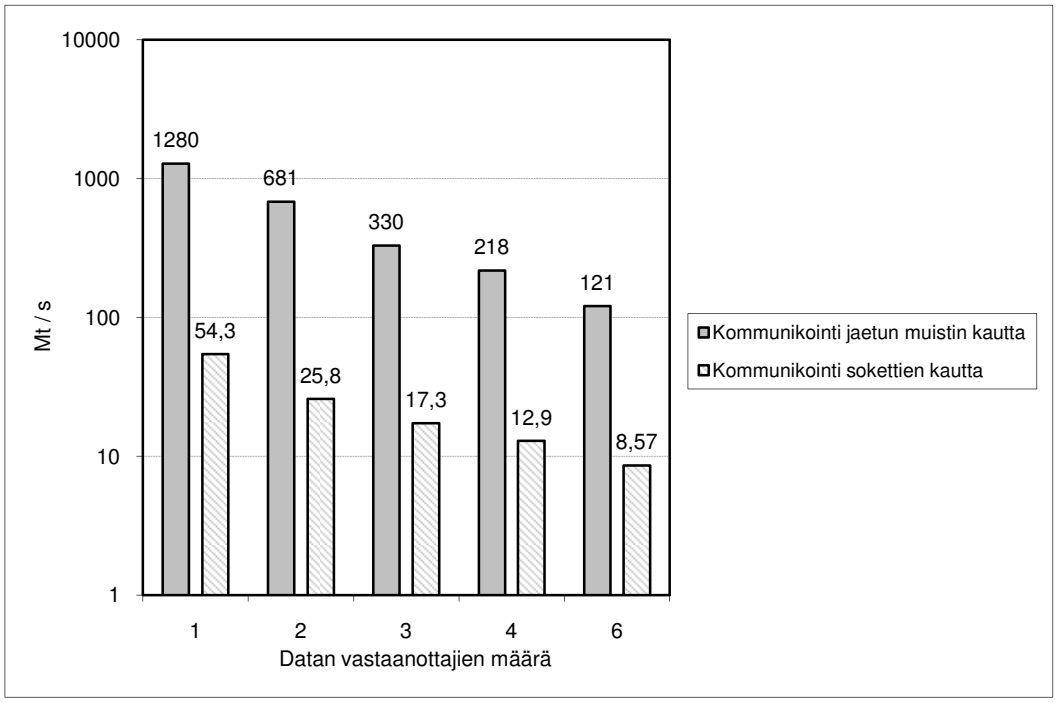
Soketin kautta tapahtuvassa kommunikoinnissa ei tapahtunut suurta eroa siirtonopeuksissa. Siirtonopeus pysyi optimaalisena, kun käytettiin pientä paketin kokoa. Tämä voi johtua TCP / IP:n luonteesta, joka on tarkoitettu verkkokommunikoimiseen. Siirtonopeus alkoi pudota reilummin siirryttäessä 256 kilotavun paketista 512 kilotavun pakettiin. Kontention osuus ei ollut millään pakettikoolla huolestuttavalla tasolla.

4.5.4 Vastaanottajien lukumäärän vaikutus siirtonopeuteen

Tässä testissä tutkittiin kuinka vastaanottajien määrä vaikutti siirtonopeuteen. Testi tehtiin käyttämällä 512 kilotavun kokoista pakettia.

Datan vastaanottajien määrä	Kommunikointi jaetun muistin kautta		Kommunikointi soketin kautta	
	Siirtonopeus (Mt / s)	Kontentio (%)	Siirtonopeus (Mt / s)	Kontentio (%)
1	1280	39,8	54,3	0,14
2	681	35,9	25,8	0,11
3	330	46,5	17,3	0,11
4	218	65,0	12,9	0,09
6	121	80,5	8,57	0,07

Taulukko 7. Siirtonopeus suhteessa datan vastaanottajien määrään



Kaavio 2. Datan vastaanottajien määrän vaikutus siirtonopeuteen

Testin tuloksista havaitaan, että siirtonopeus selvästikin putoaa datan vastaanottajien lisääntyessä. Tuloksista näkee selvästi, että jaetun muistin kautta tapahtuvassa kommunikoinnissa on sitä enemmän jätekuormaa (ks. luku 2.4), mitä useampi vastaanottaja simuloinnissa oli mukana. Kontention osuus jaettua muistia käyttävässä menetelmässä oli kuudella vastaanottajalla jo 80,5 prosenttia, mikä tarkoittaa, että suurin osa ajasta meni vastaanottajien odotteluun.

Soketteja käyttävässä menetelmässä puolestaan ei tapahtunut vastaavaa kontention kasvua vastaanottajien lisääntyessä.

5 Pohdinta

Teoriaosassa käsiteltiin yleisimpiä prosessien välisiä kommunikointimenetelmiä, niihin liittyviä mekanismeja, sekä mekanismeihin liittyviä ongelmia. Empiirisessä osassa vertailtiin kahta prosessien välistä kommunikointimenetelmää ja havaittiin käytännössä muutamia menetelmiin liittyviä asioita, joita on otettava huomioon menetelmiä toteutettaessa. Näitä asioita olivat esimerkiksi kontentio, säikeiden synkronointi käytettäessä jaettua resurssia ja jaetun muistin liittäminen virtuaaliosoiteavaruuteen. Empiirisessä osassa havaittiin, että jaettua muistia käyttävän menetelmän tehokkuus oli aivan eri luokkaa verrattuna sokettipohjaiseen kommunikointiin.

Tässä luvussa pohditaan empiirisen osan testien luotettavuutta, sekä yleisesti koko pro gradu -tutkielman aihealuetta.

5.1 Kommunikointimenetelmät

Empiiriseen osaan valittiin kaksi kommunikointimenetelmää, joihin olin jo aikaisemmin tutustunut työelämässä. Arvelin jo ennen empiirisen osan tekemistä, että jaetun muistin kautta tapahtuva kommunikointi on tehokkaampaa kuin sokettien kautta tapahtuva kommunikointi. Menetelmien tehokkuuden välinen ero kuitenkin yllätti.

Sokettipohjaisen kommunikoinnin tehokkuus on parhaimmillaan silloin, kun tietoa voidaan lähettää jatkuvasti. Esimerkiksi tilanteessa, jossa lähetetään videokuvaa verkon yli toiseen koneeseen. Tämän tutkielman empiirisessä testissä sokettiliikenne oli hyvin hetkittäistä, sillä jokaisen lähetetyn paketin jälkeen odotettiin kuittaussanoman vastaanottoa. TCP / IP:n kautta kommunikointi ei sovellu tämänkaltaiseen menetelmään kovin hyvin.

Huomasin teoriaosaa kirjoitettaessa, että kommunikointimenetelmiä on suuri määrä. Joihinkin niistä en ollut törmännyt aikaisemmin, kuten postilaatikkoon tai transaktiomuisteihin. Useimmat niistä olivat kuitenkin ennestään tuttuja.

Empiirisen osan testit olisi kannattanut toistaa Linux-käyttöjärjestelmässä, jotta olisi nähty onko Windowsin ja Linuxin verkkokommunikoinnissa ja muistinhallinnassa eroja. Kiin-

nostavaa olisi ollut myös nähdä, mitä testit olisivat paljastaneet moniprosessoritietokoneessa ja 64-bittisessä järjestelmässä. Tähän ei ollut kuitenkaan empiiristä osaa tehdessä mahdollisuutta, sillä testien tekoon olisi kulunut reilusti enemmän aikaa.

Jälkeenpäin ajateltuna olisi ollut järkevämpää valita sokettien tilalle transaktiomuisteja käyttävä menetelmä tai sitten perinteisempi nimettyjä putkia käyttävä toteutus. Tällöin valittujen menetelmien suorituskykyjen välillä ei ehkä olisi ollut niin suurta eroa.

5.2 Mittaustulokset

Datanhajautussimulaattorin tekeminen onnistui melko kivuttomasti ja sain toteutettua kaikki ne piirteet, joita ennen toteutusta hahmottelin. Mittaustulokset olivat tarkkoja, sillä ACPI-laitteistoajastimen avulla päästiin suhteellisien aikojen mittauksessa mikrosekuntitasolle. Jos mittaukset olisi pitänyt tehdä Windowsin järjestelmäkellon avulla, olisi mittauksiin tullut reilusti enemmän virhettä johtuen huonosta kellon resoluutiosta.

Mittaustulokset siirrettiin tekstimuotoisesta tiedostosta Microsoft Excel -taulukkolaskentaohjelmaan, jossa lopulliset tulokset laskettiin. Jälkeenpäin ajateltuna olisi ollut viisaampaa toteuttaa automaattinen menetelmä, jolla tulokset olisi laskettu. Silloin raaka käsityö olisi jäänyt pois, kuten myös mahdollisuus virheisiin tuloksia laskettaessa.

Empiirisen osan toiseksi viimeisessä testissä, jossa mitattiin paketin koon vaikutusta siirtomenetelmien nopeuteen, huomattiin, että optimaalisin paketin koko on 256 kilotavua. Aikaisemmat testit olisi pitänyt tehdä myös tällä tehokkaimmalla paketin koolla, mutta suuren työmäärän vuoksi testejä ei uusittu. Menetelmien välinen ero siirtonopeudessa olisi ollut hieman kapeampi, sillä 512 kilotavun pakettikoko pudotti enemmän sokettipohjaisen siirtomenetelmän suorituskykyä verrattuna jaetun muistin kautta tapahtuneeseen menetelmään. Eron kaventuminen ei kuitenkaan olisi ollut merkittävää luokkaa.

Testien tulokset eivät sellaisenaan takaa, että testien mukaan parempi menetelmä olisi myös parempi menetelmä valmiissa tuotteessa. Empiirisessä osassa tehdyissä testeissä käytettiin kaikki tietokoneen laskentakapasiteetti pelkästään datan siirtoon. Monesti ohjelmissa

koneen laskentakapasiteettia käytetään ohjelman varsinaiseen tehtävään ja prosessien välinen kommunikointi suoritetaan jäljelle jääneellä kapasiteetilla. Tästä syystä prosessien välisen kommunikoinnin hyötysuhteen pitää olla hyvä.

5.3 Optimaalisen menetelmän valinta

Prosessien väliseen kommunikointiin on olemassa reilu valikoima erilaisia menetelmiä. Näille jokaiselle on olemassa hyvät ja huonot puolet, joten järjestelmän suunnittelijan on osattava valita järjestelmää varten oikea menetelmä.

Suuren järjestelmän jakamista pienempiin prosesseihin pelätään suorituskyvyn heikkenemisellä, mutta monesti pullonkaulat ovat todellisuudessa aivan toisaalla. Jako prosesseihin selkeyttää laajojen järjestelmien monimutkaisuutta ja järjestelmän tehtävien jakoa. Järjestelmän vikasietoisuus kasvaa, sillä jos jokin järjestelmän osa ei toimi kunnolla, voivat muut osat jatkaa toimintaansa. Ihanteellisessa tapauksessa viallisen osan myötä järjestelmästä putoaa vain yksi ominaisuus pois. Järjestelmän laajennettavuus helpottuu, sillä muutoksia on helpompi tehdä pienempiin ja yksinkertaisenpiin prosesseihin.

Kommunikointimenetelmää valittaessa on päätettävä ajetaanko prosesseja saman tietokoneen sisässä vai eri tietokoneissa. Mietittävä on myös aiotaanko myöhemmin hajauttaa samassa koneessa toimivat prosessit eri tietokoneisiin. On järkevää miettiä tarvitaanko ylipäätään tehokasta menetelmää, vai riittääkö järjestelmään esimerkiksi soketteihin perustuva menetelmä. Viisainta olisi toteuttaa prosessien välinen kommunikointi modulaarisesti, jotta myöhemmin voidaan vaihtaa kommunikointimenetelmää suhteellisen pienellä työmäärällä.

5.4 Tutkielman tavoitteet

Tämän pro gradu -tutkielman tarkoituksena oli perehtyä otsikon mukaisesti prosessien välisten siirtomenetelmien toimintaan ja niiden tehokkuuteen. Teoriaosassa pyrkimyksenä oli kertoa järjestelmällisesti kaikki tekniikat ja asiat, joiden päälle jaetun muistin kautta tapahtuva kommunikointi rakentuu. Empiirisen osan tarkoitus oli selventää teoriaosan asioita

yksinkertaisella sovellusesimerkillä, samalla antaen suorituskykyarvoja kahdella valitulla kommunikointimenetelmällä.

Tavoitteisiin päästiin mielestäni kohtuullisen hyvin. Prosessien välisistä kommunikointimenetelmistä käytiin kaikki tärkeimmät läpi, joita voisi ajatella nykyaikaisiin käyttöjärjestelmiin suunnitelluissa sovelluksissa käytettävän. Ehkä tutkielmassa olisi voinut olla enemmän tietoa DCOM:ista ja CORBA:sta, sillä näitä menetelmiä käytetään laajalti teollisuudessa. Transaktiomuistit olisivat olleet myös kiinnostava kohde tutkia tarkemmin, koska niissä säikeiden välinen synkronointi olisi toteutettu lukkovapaasti. Kommunikointimenetelmiin liittyvät tekniikat, kuten molemminpuolinen poissulkeminen, semafori, kriittinen alue sekä rajoitteet, kuten kontentio, lukon jonotusongelma ja molemminpuolisesta poissulkemisesta johtuva jätekuorma, tulivat selkeästi käytyä teoriaosassa läpi.

Empiirinen osa selvensi teoriaosaa, sillä siten asioita joutui ainakin kirjoittajan näkökulmasta miettimään hieman syvällisemmin. Simulaattorin toteuttaminen oli hyvä tapa esitellä prosessien välisiin kommunikointimenetelmiin liittyviä ongelmakohtia käytännössä. Mittaustuloksien avulla saatiin kommunikointimenetelmien toiminnasta runsaasti tietoa, jonka avulla saatiin selvitettyä hyvin tarkasti, mitkä kommunikointimenetelmän osa-alueet olivat menetelmän ongelmakohtia.

6 Yhteenveto

Tutkielmassa tutkittiin prosessien välisiä kommunikointimenetelmiä, menetelmiin liittyviä tekniikoita ja menetelmien suorituskykyä. Sisällytettyjä kommunikointimenetelmiä olivat mm. soketit, DCOM, CORBA ja nimetyt putket. Tutkielman pääpaino oli jaetun muistin kautta tapahtuvassa kommunikoinnissa.

Prosessien väliseen kommunikointiin sisältyy erilaisia mekanismeja ja tekniikoita, joiden päälle kommunikointimenetelmät rakentuvat. Monissa menetelmissä tarvitaan synkronointioperaatioita, joiden avulla kommunikoijat saavat vuonhallinnan toteutettua. Tutkielmassa käytiin läpi semaforin, kriittisen alueen ja molemminpuoliseen poissuljentaan liittyviä asioita. Tutkielmassa syvennyttiin prosessien synkronisoinnista johtuvaan kontentioon, joka etenkin jaettua muistin kautta tapahtuvassa kommunikoinnissa tulee menetelmän tehokkuuden rajoitteeksi.

Tässä tutkielmassa jaetun muistin kautta tapahtuva kommunikointi on rakennettu käyttöjärjestelmän tarjoaman muistiarkkitehtuurin päälle. Nykyaikaisissa käyttöjärjestelmissä jokaisen prosessin muistiavaruus on suojattu, jolloin jaettu muisti toteutetaan käyttöjärjestelmän ehdoilla. Tutkielmassa käytiin läpi kaksi tapaa kuinka Windows NT -käyttöjärjestelmät tarjoavat muistin näkymisen muille prosesseille. Jaettu muisti voidaan jakaa prosessien kesken absoluuttisesti samoihin muistiosoitteisiin tai sitten eri osoitteisiin. Tutkielmassa havainnollistettiin tästä aiheutuvia ongelmia.

Tutkielman empiirisessä osassa havainnollistettiin simulaattorin avulla kahden prosessien välisen kommunikointimenetelmän toimintaa. Menetelmiksi valittiin sokettien ja jaetun muistin kautta tapahtuva kommunikointi. Jaetun muistin kautta tapahtuva kommunikointi toteutettiin käyttäen teoriaosassa esiteltyjä tekniikoita. Simulaattorilla saaduista mittaustuloksista nähtiin, että samassa koneessa tapahtuva prosessien välinen kommunikointi oli merkittävästi nopeampaa käytettäessä jaettua muistia.

Tutkielma havainnollisti jaetun muistin kautta tapahtuvan kommunikoinnin olevan potentiaalinen kommunikointimenetelmä, kun kommunikoivat prosessit ovat samassa tietokoneessa.

Lähteet

[ABi84] Andrew D. Birrell, Bruce Jay Nelson, *Implementing Remote Procedure Calls*, ACM Transactions on Computer Systems (TOCS), Volume 2, Issue 1, 2 / 1984

[AMa86] A. J. Martin, *A New Generalization of Dekker's Algorithm for Mutual Exclusion*, IPL 23(6), 1986

[AMD64] *AMD64 Architecture Programmer's Manual Volume 2: System Programming*, http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/24593.pdf

[CDw97] Cynthia Dwork, Maurice Herlihy, Orli Waarts, *Contention in Shared Memory Algorithms*, Journal of the ACM (JACM), Volume 44, Issue 6, 11 / 1997

[DMo93] David Mosberger, *Memory Consistency Models*, ACM SIGOPS Operating Systems Review, Volume 27, Issue 1, 1 / 1993

[DRu75] David L. Russell, Thomas H. Bredt, *Error resynchronization in producer-consumer systems,* Proceedings of the fifth ACM symposium on Operating systems principles, 1975

[EDi65a] E. W. Dijkstra, *Solution of a Problem in Concurrent Programming Control*, Communications of the ACM, Volume 8, Issue 9, 9 / 1965

[EDi65b] E. W. Dijkstra, *Cooperating Sequential Processes*, (Technice Hoheschool, Eindoven, 1965), Reprinted in: F. Genuys (ed.), Programming Languages, Academic Press, 1968

[EDi68] E. W. Dijkstra, *The structure of the "THE"-multiprogramming system*, Communications of the ACM, Volume 11, Issue 5, 5 / 1968

[EDi71] E.W. Dijkstra, *Hierarchical Ordering of Sequential Processes*, Acta Informatica, Volume 1, 1971

[ESta82] E. W. Stark, *Semaphore Primitives and Starvation-Free Mutual Exclusion*, Journal of the ACM, Volume 29, Issue 4, 10 / 1982

[For95] Message-Passing Interface Forum, *MPI: A Message- InterfaceStandard*, University of Tennessee, 1995.

[GBr83] Gabriel Bracha, Sam Toyeg, *Resilient Consensus Protocols*, Proceedings of the second annual ACM symposium on Principles of distributed computing, 1983

[GPe81] G. L. Peterson, *Myths About the Mutual Exclusion Problem*, IPL 12(3), 1981

[IA64] *Intel® 64 and IA-32 Architectures Software Developer's Manul*, http://www.intel.com/design/processor/manuals/253665.pdf, 11 / 2006

[JHa01a] Juha Harakka, Lasse Lilja, Tommi Hytönen, Pekka Paadar, Olli-Pekka Saxell, Tuomas Tuurala, Mikko Vapa, *Comparison of Distributed Real-Time Operating Systems*, 1 / 2001

[JHa01b] Juha Haataja ja Kaj Mustikkamäki, *Rinnakkaisohjelmointi MPI:llä,* CSC - Tieteellinen laskenta Oy, 2. painos, 2001, http://www.csc.fi/oppaat/mpi/

[JQu85] John S. Quarterman, Abraham Silberschatz, James L. Peterson, *4.2BSD and 4.3BSD as examples of the UNIX System*, ACM Computing Surveys, Volume 17, Issue 4, 12 / 1985

[JRi96] Jeffrey Richter, *Advanced Windows*, Microsoft Press, A Division of Microsoft Corporation, ISBN 1-57231-548-2, 1996

[KAl03] K. Alagarsamy, *Some myths about famous mutual exclusion algorithms*, ACM SIGACT News, Volume 34 , Issue 3, 10 / 2003

[KGh90] K. Gharachorloo, D. Lenoski, J.Laudonm Phillip Gibbons, Anoop Gupta, John Hennesy, *Memory consistency and event ordering in scalable shared-memory multiprocessors*, Computer Architecture News, 6 / 1990

[KLi86] Kai Li, Paul Hudak, *Memory Coherence in Shared Virtual Memory Systems*, Proceedings of the fifth annual ACM symposium on Principles of distributed computing, ISBN: 0-89791-198-9, 1986

[LLa79] Leslie Lamport, *How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Prog*rams, IEEE Trans. Comput. C-28,9, p. 690-691, 9 / 1979

[MDu86] M. Dubois, C. Scheurich, F. A. Briggs, *Memory Access Buffering in Multi-processors*, Proceedings of the 12th International Conference on Distributed Computing Systems, p. 416 – 423, 6 / 1992

[MHe93] Maurice Herlihy, J. Eliot B. Moss, *Transactionaly Memory: architectural support for lock-free data structures*, Proceedings of the 20th annual international symposium on Computer Architecture p. 289 - 300, 1993

[MSDNa] Useita kirjoittajia, *Microsoft Developer Network: Pipes*, http://msdn2.microsoft.com/en-us/library/aa365780.aspx, haettu 14.7.2007

[MSDNb] Useita kirjoittajia, *Microsoft Developer Network: Mailslots*, http://msdn2.microsoft.com/en-us/library/aa365576.aspx, haettu 14.7.2007

[MSDNc] Useita kirjoittajia, *Microsoft Developer Network: DCOM Architecture*, http://msdn2.microsoft.com/en-us/library/ms809311.aspx, haettu 14.7.2007

[MSDNd] Useita kirjoittajia, *Microsoft Developer Network: Using Event Objects*, http://msdn2.microsoft.com/en-us/library/ms686915.aspx, haettu 14.7.2007

[MSDNe] Useita kirjoittajia, *Microsoft Developer Network: Clipboard*, http://msdn2.microsoft.com/en-us/library/ms648709.aspx, haettu 14.7.2007

[NHa71] A. Nico Habermann, *Synchronization of Communicating Processes,* Proceedings of the third ACM symposium on Operating systems principles, 10 / 1971

[OMG03] Object Management Group, *Object Transaction Service*, http://www.omg.org/docs/formal/03-09-02.pdf, 10 / 2003

[OMG04] Object Management Group, *Common Object Request Broker Acrhitecture: Core Specification*, http://www.omg.org/technology/documents/corba_spec_catalog.htm, 3 / 2004

[RAl94] Rajeev Alur, Gadi Taubenfeld, *Contention — free complexity of shared memory algorithms*, Proceedings of the thirteenth annual ACM symposium on Principles of distributed computing, 1994

[RFC1831] *RPC: Remote Procedure Call Protocol Specification Version 2*, RFC 1831, ftp://ftp.funet.fi/pub/doc/rfc/rfc1831.txt, 8 / 1995

[RFC793] Information Sciences Institute, *Transmission Control Protocol*, RFC 793, ftp://ftp.funet.fi/pub/doc/rfc/rfc793.txt, 1981

[RRa02] Ravi Rajwar, James R. Goodman , *Transactional lock-free execution of lock-based programs*, Proceedings of the 10th international conference on Acrhitectural support for programming languages and operating systems, 2002

[THa83] Theo Haerder, Andreas Reuter, *Principles of Transaction-oriented database recovery*, ACM Computing Surveys, Volume 15, Issue 4, 12 / 1983

[WHDC] *Guidelines For Providing Multimedia Timer Support*, Windows Hardware Developer Central, http://www.microsoft.com/whdc/system/CEC/mm-timer.mspx, 10 / 2002

Termit

CORBA	Object Management Groupin määrittelemä standardi, joka sallii eri alustoilla toteutettujen prosessien välisen kommunikoinnin verkon yli (Common Object Request Broker Architecture)
DCOM	Microsoftin toteuttama teknologia, joka sallii prosessien välisen kommunikoinnin verkon yli (Distributed Component Object Model)
DSM	Hajautettu jaettu muisti (Distributed Shared Memory)
Kontekstin vaihto	Prosessorin tukema operaatio, jonka avulla saadaan simuloitua moniajoa
Kontentio	Viive, joka aiheutuu säikeitten odottamisesta pääsyä jaettuun resurssiin
Kriittinen alue	Alue ohjelmakoodissa, jonka suorittaminen yhdellä säikeellä kerrallaan on kriittistä ohjelman toimimiselle oikein
MPI	Viestinvälitysrajapinta (Message Passing Interface)
Multicore	Prosessori, jossa on useampi kuin yksi ydin
Multiprocessor	Järjestelmä jossa on useampi kuin yksi prosessoriyksikkö
Mutual exclusion	Molemminpuolinen poissuljenta, mekanismi rajoittaa rinnakkaisten säikeiden pääsyä kriittiselle alueelle
Nimetty putki	Prosessien välinen kommunikointimenetelmä, joka loogisesti ajateltuna on putki, joka pitkin data välittyy toiselle prosessille
POSIX	Standardiperhe, joka määrittää alustariippumattomat API-rajapinnat esim. tiedosto- ja säiekäsittelylle, jotta sovellus voisi

	toimia alustariippumattomasti (portable operating system interface for unix)
RPC	Etäkutsu (Remote Procedure Call)
Semafori	Mekanismi, jolla molemminpuolinen poissuljenta toteutetaan
Transaktio	Tapahtuma, joka suoritetaan vain jos sen sisältämät ACID-mallin ehdot ovat voimassa