Vikasietoisuus ja kuormanjako J2EE™ sovelluspalvelimissa

Tuomas Nurmela

Helsinki 18.marraskuuta 2002

Hajautetut sovellukset -seminaari

HELSINGIN YLIOPISTO

Tietojenkäsittelytieteen laitos

Vikasietoisuus ja kuormanjako J2EE ™ sovelluspalvelimissa

Tuomas Nurmela

Hajautetut sovellukset seminaari

Tietojenkäsittelytieteen laitos

Helsingin Yliopisto

Helsinki 18.marraskuuta 2002, 24 sivua

Seminaaritutkielma käsittelee J2EE sovelluspalvelimen klusteroinnin toteuttamista sovelluspalvelimen ja sovelluskehittäjän näkökulmasta. Tutkielmassa tarkastellaan aluksi hajautusasteeseen vaikuttavia tekijöitä, joita seuraa sovelluspalvelimen klusteroinnin toteutusvaihtoehtojen arviointi Web-säiliöön, EJB-säiliöön sekä JNDI-rajapintaan. Samassa yhteydessä tarkastellaan klusterin ryhmäkommunikointi-mekanismia sekä tämän toteuttamisvaihtoehtoja. Tutkielman lopuksi tarkastellaan open source sovelluspalvelimista Tomcat 4.0 J2EE web-säiliön sekä JBoss 4.0 EJB-säiliön toteutuksia.

Avainsanat: J2EE, load balancing, fault tolerance, clustering, Tomcat, JBoss

SISÄLLYSLUETTELO

1 JOHDANTO..1

2 J2EE JÄRJESTELMÄN HAJAUTUSASTE...2

3 J2EE SOVELLUSPALVELINKLUSTERIT...4

3.1 Ryhmäkommunikointimekanismi...5
3.2 Web-säiliö...7
3.3 JNDI-nimipalvelu 9
3.4 EJB-säiliö...10

4 TOMCAT JA JBOSS KLUSTEROINTI...14

4.1 JavaGroups..14
4.2 Tomcat 4.0 klusterointi 16
4.3 JBoss 3.0 klusterointi 17

5 YHTEENVETO 22

LÄHTEET 23

1 JOHDANTO

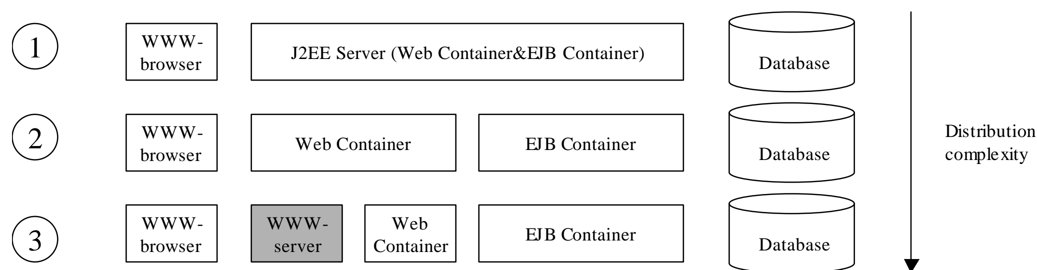
Internetissä toimivan palvelun vikasietoisuuden ja kuormanjaon rakentaminen lähtökohtana toimii *saapumismekanismi* (initial access logic), jonka myötä asiakkaiden pyynnöt saadaan hajautettua useammille palvelimille. Esimerkkejä tällaisista ovat DNS nimipalvelussa toteutettu kuormanjako, WWW-palvelinproxy tai kuormanjakokytkin. Kuormanjaon ja vikasietoisuuden vieminen läpi järjestelmän edellyttää kuitenkin kokonaisvaltaista suunnittelua, mukaanlukien J2EE sovelluspalvelimen toiminnan varmistaminen. J2EE standardi ottaa kuitenkin hyvin vähän kantaa näihin kysymyksiin, tarjoten ainoastaan arkkitehtuurin hajautuksen kautta skaalautumiselle tietynasteista tukea. J2EE sovelluspalvelintoimittajat ovat lähestyneet ongelmaa *sovelluspalvelinlusteroinnilla*, joissa ryhmäkommunikointimekanismia hyödyntäen mahdollistetaan usean sovelluspalvelimen löyhään sidokseen perustuva rinnakkaistoiminta.

Sovelluspalvelimen klusteroinnissa HTTP-istunnon tilan replikointi mahdollistaa J2EE *web-säiliölle* (web container) *läpinäkyvän vikatilanteesta toipumisen* (transparent failover) web-säiliön kaatuessa, parantaen saapumismekanismin luomaa perustaa. Edelleen *ejb-säiliön* (ejb container) instantoitujen objektien vikasietoisuutta ja kuormanjakoa voidaan myös tukea rajoitetusti ejb-papujen sekä J2EE palvelujen osalta.

Tutkielma jakautuu seuraaviin osiin: Luvussa 2 tarkastellaan J2EE hajautusasteen valintaan vaikuttavia tekijöitä. Luvussa 3 käsitellään sovelluspalvelimen klusteroinnin toteutusvaihtoehtoja. Luvussa 4 tarkastellaan Tomcat 4.0 web-säiliön ja JBoss 3.0:n ejb-säiliön klusterointitoteutuksia sekä näiden toteuttamiseksi vaadittavia mekanismeja. Luvussa 5 on yhteenveto.

2 J2EE JÄRJESTELMÄN HAJAUTUSASTE

J2EE järjestelmän hajautusasteen kasvaminen merkitsee monimutkaisuuden järjestelmän kasvua. Hajautusasteen tyypillisesti väitetään vaikuttavan (positiivisesti) suoraan järjestelmän skaalautuvuuteen ja vikasietoisuuteen sekä parantavan turvallisuutta. Perusteena on kyky partitioida järjestelmä fyysisesti useammalle koneelle. Käytännössä tilanne ei ole aivan näin yksiselitteinen. Kuva 1 määrittää hajautusastetet J2EE arkkitehtuurille, huomioiden lisäksi mahdollisuuden käyttää erillistä WWW-palvelinta staattisen sisällön tarjoamiseen.



Kuva 1: J2EE järjestelmän hajautusaste [muokattu RAJ02, s.414]

Hajautusasteiden tarkastelu eri näkökulmasti selkeyttää sen vaikutusta [RAJ02, s.415-417]:

- Suorituskyvyn näkökulmasta hajautusaste 1 sisältää vähiten tietoliikennettä (prosessien välistä tai palvelinten välistä) ja käyttää todennäköisimmin tehokkaimmin resursseja.
- Vikasietoisuuden näkökulmasta hajautusaste 1 sisältää vähiten mahdollisesti virheitä aiheuttavia osatekijöitä. Toisaalta J2EE palvelimen kaatuminen estää palvelun saatavuuden. Web-säiliön ja EJB-säiliön erottaminen voi johtaa siihen, että vain EJB-säiliö kaatuu, jolloin staattinen sisältö sekä Web-säiliöllä paikallisesti generoitu sisältö ovat vielä saatavilla.
- Ylläpidettävyyden näkökulmasta hajautusaste 1 sisältää identtisen sisällön, jonka myötä ylläpito on yksinkertaisinta

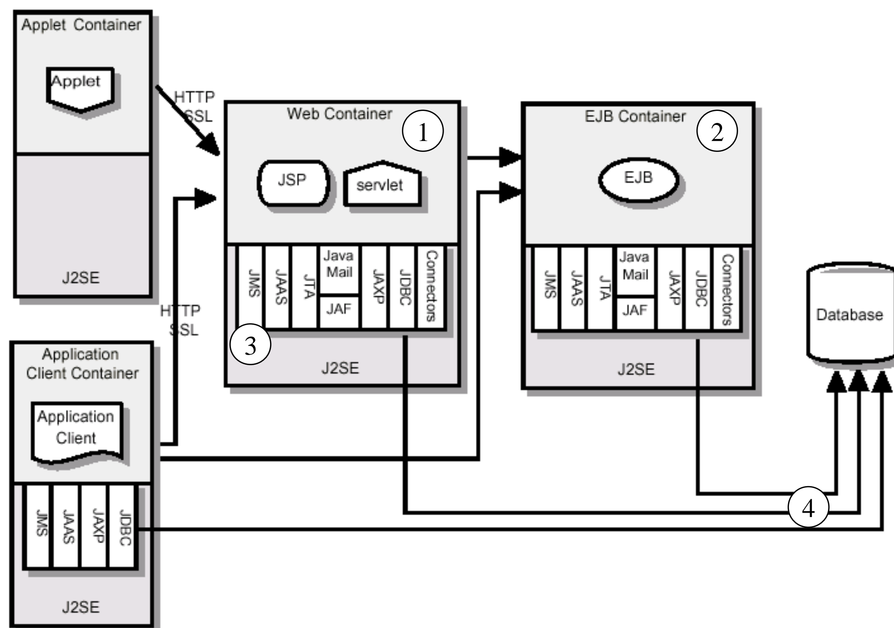
- Infrastruktuurisuunnittelun näkökulmasta hajautusaste 1 on helpoin suunnitella, koska fyysisen hajautuksen osalta ei tarvitse harkita mitkä palvelinten määrälliset suhteet tulevat olemaan toisiinsa. Toisaalta, mikäli palvelimia ei ole dedikoitu J2EE järjestelmän käyttöön, tulee jokatapauksessa harkittavaksi, sisällyttääkö resurssimanagerointiohjelmisto palvelimelle virheellisesti käyttäytyvän sovelluksen vaikutusalueen rajaamiseksi.
- Kuormanjaon näkökulmasta ratkaisut ovat tasavertoisia, sillä saapumismekanismiin liittyvä kuormanjako edeltää palvelimia.

Monimutkaisempia arkkitehtuureja saattavat toisaalta puoltaa ei-tekniset syyt [RAJ02, s.415-417]:

- Tietoturva näkökulmasta hajautusaste 2 ja 3 tarjoavat mahdollisuuden erottaa palomuurilla WWW-palveluja tuottavat palvelimet (joilla julkinen sisältö) EJB-säiliön liiketoiminnallisuutta tarjoavista sovelluspalvelimista, joissa sisältö on luottamuksellista.
- Suurissa järjestelmissä sovelluskehitys voi olla jakautunut erikseen WWW sisällön sekä liiketoimintasovelluksia toteuttavan EJB-kehitystiimin kesken, jolloin monimutkaisemmat hajautusasteet tarjoavat luonnolliset rajat. Toisaalta www-palvelimien tarjoamat virtuaalipalvelinmahdollisuudet saattavat myös olla syy erillisen www-palvelimen käyttämiseen.

3 J2EE SOVELLUSPALVELINKLUSTERIT

J2EE sovelluspalvelissa kuormanjaon ja vikasietoisuuden tukeminen edellyttää ryhmäkommunikointimekanismin (3.1) muutoksen JNDI-toteutukseenn (3.2) muutoksia itse sovelluspalvelimen palveluihin. Kuva 2 määrittää alueet, joissa sovelluspalvelintoimittajat tyypillisesti toteuttavat näitä piirteitä.



Kuva 2: J2EE 1.3 arkkitehttuuri [Sun01a, s. 4]

Muutokset sovelluspalvelimissa voidaan kiteyttää seuraavasti:

1. Web-säiliön muokkaaminen *HTTP-istunnon tilaa* (HTTP Session State) manipuloimalla
2. EJB-säiliön muokkaaminen home ja remote stubia manipuloimalla, JNDI APIn muutoksilla sekä paputyyppikohtaisilla klusterointimäärityksillä
3. Java Messaging System (JMS) muokkaaminen mahdollistamalla rinnakkaisten JMS topicien ja jonojen määritys samalle kohteelle
4. Java Database Connection (JDBC)-tietokantayhteyksien muokkaaminen rinnakkaisilla *yhteysaltailla* (connection pool) tai *tietolähteillä* (data source)

Sovelluspalvelinmuutosten osalta tutkielma rajautuu web-säiliön (3.3) ja EJB-säiliön (3.4) tarkasteluun.

3.1 Ryhmäkommunikointimekanismi

Ryhmäkommunikointimekanismi tyypillisesti koostuu ryhmän koordinoinnista vastaavasta ryhmäjäsenyyspalvelusta sekä joko deduktiivisesta tai probabilitistesta sanomien perillemenosta vastaavasta kommunikointimekanismista. Skaalautuvissa ratkaisuissa tulee varmistaa myös mekanismin tietoturvapiirteet. Tällä hetkellä tunnetuimmissa sovelluspalvelimissa (BEA WebLogic, IBM WebSphere, JBoss Group JBoss) sivuutetaan ryhmäkommunikoinnin tietoturva, sillä skaalautuvuus, ainakin implisiittisesti, rajautuu paikallisverkkototeutuksiin, joissa mekanismi rakennetaan palvelimen tuotantoyhteydestä erilliseksi.

Ryhmäjäsenyyspalvelu (Group Membership Service, GMS) vastaa ryhmän tilan seurannasta ja tilatiedon päivityksestä ryhmän jäsenelle.

GMS:n neljä ydintehtävää ovat [CDK01 s.559]:

1. sovellusrajapinnan tuottaminen, mahdollistaen prosessiryhmän luonnin ja tuhoamisen sekä jäsenen lisäämisen ja poistamisen
2. vian havaitseminen niin jäsenen kaatumisen kuin virheellisen toiminnan osalta
3. ryhmämuutoksesta (uuden jäsenen lisäyksestä tai poistumisesta) ilmoittaminen ryhmän jäsenille
4. yksikäsitteisen, ryhmän määrittävän ryhmätunnisteen sisällyttäminen jäsenten ryhmille lähettämiin sanomiin

Deduktiivinen ryhmäkommunikointi rakentuu ryhmän sanomien järjestyksen tarkkailuun sekä ryhmäkommunikoinnin koordinointiin. Vikatilanteen tapauksessa johdetaan tila aiemmista tilamuutoksista. Lähtökohdan myötä luotettava vikasietoisuus suunnitellaan näkökulmasta, jossa verkon oletetaan tuottavan virheitä, jotka vaarantavat tilatiedon ylläpitämisen.

Sanomien järjestysaste voidaan pelkistää seuraaviin luokkiin [CDK01, s.558]:

- järjestyksetön: järjestelmä ei takaa sanomien järjestystä
- FIFO: järjestelmä takaa kunkin yksittäisen jäsenen lähettämien sanomien käsittelyjärjestyksen lähetysjärjestyksessä yksittäisen vastaanottajan osalta

- kausaalinen järjestys: järjestelmä takaa kaikkien lähettäjien sanomalähetyksen mukaisen käsittelyjärjestyksen yhden vastaanottajan näkökulmasta
- täydellinen järjestys: mikäli yksi vastaanottaja käsittelee sanomat oikein kausaalisen järjestyksen mukaisesti, kaikki vastaanottajat käsittelevät sanomat samassa järjestyksessä

Paikallisverkkoon sijoittuvalle sovelluspalvelinklusteri ei tyypillisesti tarvitse täydellistä järjestystä takaavaa ryhmäkommunikointimekanismia.

Todennäköisyyspohjainen ryhmäkommunikointi (probabilistic broadcast, pbcast) ei perusmuodossaan oleta yhtä voimakasta sanomajärjestyksen tarvetta kuin deduktiivinen ryhmäkommunikointi. Tämän myötä laajojen järjestelmien toteutuksissa skaalautuvuus on vähintään helpommin saavutettavissa. Pbcastin toiminta perustuu osaltaan oletukseen verkon ja sovelluksen toimintakyvystä, painottaen palvelun saatavuuden varmistamiseen. Pbcast toteutuksissa ydintehtäviksi tulee taata *konvergenssiaika* (eventual convergence; aika, johon mennessä sanoma on välitetty kaikille jäsenille tai ei yhdellekään jäsenelle), välttää verkon kuormitusta turhilla sanomilla sekä mahdollistaa GMS:n luotettava mukautuminen ryhmäjäsenyysmuutoksiin (tyypillisesti erillisellä vian havaitsemismekanismilla). [Bir96, s.472-581]

Toteutusvaihtoehdot ryhmäkommunikointimekanismin jakautuvat *wrappereihin* (wrapper) sekä irrallisiin *toolkit-API:n* (toolkit) [Bir96, s.351-370].
Wrapperit perustuvat jossakin rajapinnassa tapahtuvaan, sovellukselle läpinäkyvään *kaappaukseen* (interception) ja uudelleenkäsittelyyn tai läpinäkyvään *väliintuloon* (interpositioning), jossa osapuolten väliin sijoitetaan erillinen prosessi, joka suorittaa uudelleenkäsittelyn. Vaihtoehtoisia rajapintoja sieppaamiselle ovat:

- systeemikutsu-tason kaappaus esimerkiksi UNIX:n /proc-pseudo-tiedostojärjestelmän kautta, jolloin tietyt systeemikutsut voidaan esimerkiksi ohjataan uudelleenkäsiteltäväksi toiselle prosessille
- jaetun kirjaston funktion kaappaus ja osittainen uudelleenkirjoitus sovelluksen tarjoamien rajapintojen kautta (esimerkiksi käyttöjärjestelmissä esiintyvät hookit)
- olioluokan periyttämisen kautta metodien korvaus tai laajentaminen

Rajapinnan määrittämisessä muutoksen vaikutuksen arviointi on keskeisessä asemassa: esimerkiksi socket()-funktion toteutuksen muuttaminen jaetussa kirjastossa ainoastaan vaikuttaa funktio-kutsuihin siinä missä vastaava systeemikutsu-tason sieppaus edellyttäisi open-systeemikutsun uudelleenkäsittelyä, vaikuttaen myös tiedostonavauksessa (fopen()-funktio) tapahtuvaan käsittelyyn. Toisaalta, mikäli esimerkiksi sovelluskoodi ei ole saatavilla, ei jaettuun kirjastoon perustuva toteutus ole mahdollinen. Olio-ohjelmoinnin lähtökohdista olioluokan periyttäminen on varmasti tyypillisesti vaivattomimmin menettely wrapper-vaihtoehdoista.

Toolkit-API:t koostuvat sovellustason mikroprotokollista, jotka kukin toteuttavat yhden tarvittavan toiminnallisen osan, kerroksittain rakentaen lisää piirteitä sanomankäsittelyyn. API:n metodikutsuissa määritetään syöteparametreilla, mitkä mikroprotokollat kuuluvat käytettävään ryhmäkommunikaatiomekanismiin ja mitä kuljetuskerroksen protokollaa käytetään toteutuksessa. Parametrointia muuttamalla saadaan joko yksinkertaistettua tai monimutkaistettua ryhmäkommunikointi-mekanismin toimintaa. Vaikkakin toteutus edellyttää muutoksia sovelluspalvelimeen, vältetään tämän myötä wrappereiden ennaltamääritetyn ympäristön rajoite.
ajoite.

3.2 Web-säiliö

HTTP-istunnon vikasietoisuus toteutetaan HTTP-istunnon tilatiedon tallentamisella ja jakamisella. Sovelluspalvelimen toteutusvaihtoehdot ovat [GPD02]:

- *tiedostoon tallennettu tila* (file-stored state): Kaikki web-säiliöt tallentavat jaettuun tiedostoon tilatietoja, mahdollistaen tilatiedon haun, mikäli web-säiliö kaatuu. Tiedostoon tallennuksen ongelmana ovat jaetun levyalueen vaatimus, operaation hitaus ja tämän myötä skaalautuvuuden rajoittaminen.
- *tietokantaan tallennettu tila* (database-stored state): Kaikki web-säiliö tallentavat tilatiedon tietokantaan, mahdollistaen tilatiedon haun. Mekanismi on hidas ja rajoittaa skaalautuvuutta.
- *keskitetty tilapalvelin* (centralized state server): Kaikki web-säiliö lähettävät tilapalvelimelle tilamuutostiedot. Mikäli web-säiliö kaatuu, hakee toinen web-säiliö tilatiedot muistista. Tilapalvelimella itsellään on varapalvelin, joka ottaa hoidettavakseen sen tehtävät, mikäli tilapalvelin kaatuu.

- *replikointi muistissa* (in-memory state replication): tilaa säilytetään ainoastaan muistissa. Tilan replikointi voi tapahtua tietylle toiselle klusterin jäsenelle, osalle klusterin jäsenistä tai kaikille klusterin jäsenille. Mikäli replikointi ei tapahdu koko ryhmälle, on muiden klusterijäsenten kyettävä edelleenohjaamaan käsittely toipumisesta vastaaville klusterijäsenille, mikäli vikatilanteen seurauksena pyyntö asiakkaalta ohjautuu ensin heille.

Suorituskyvyn näkökulmasta keskitetty tilapalvelin voi pienellä klusterijäsenten määrällä olla erittäin kilpailukykyinen vaihtoehto. Klusterin jäsenmäärän kasvaessa tilapalvelimesta voi kuitenkin tulla pullonkaula. Muistissa tapahtuva istunnon tilan replikointi on nykyisille sovelluspalvelimille (esim. BEA WebLogic, IBM WebSphere) tyypillisin. Tutkielman aikana ei löytynyt J2EE sovelluspalvelintoteutusta, joka olisi toteuttanut aktiivista replikointia.

HTTP-istunnon kuormanjako perustuu saapumismekanismin tarjoamiin vaihtoehtoihin. Sovelluspalvelinklusterin sisäisen kuormanjaon toteutus perustuu uusien istunnonperustuspyyntöjen siirtämiseen toiselle klusterijäsenelle. Tällöin, mikäli web-säiliölle kertyy jonotusta, voi se edelleensiirtää pyynnöt edellyttäen että saapumismekanismi tukee menettelyä (mikäli esim. paluusanoma tulee toiselta palvelimelta).

Koska HTTP-istunnon kopiointi edellyttää web-säiliöltä kykyä siirtää tilatieto jollekin toiselle kohteelle, edellyttää tämä poikkeuksetta HTTP istunnon tilatiedon serialisointia. Täten, vaikka JSP 1.2 / Servlet 2.3 standardit [Sun01c] eivät tätä edellytäkään, tulee tilaan tallentaa vain serialisoitavaa dataa.

Web-säiliön klusteroinnin osalta seuraavat asiat tulisi ohjeistaa sovelluskehityksessä ja järjestelmän ylläpidossa [Han01]:

- Suurten objektien luonti voi jäädyttä toiminnan kopioinnin ajaksi. Vaikutus on riippuvainen klusterin kommunikointimekanismin toimintatavasta.

- Istunnon aikana klusteroinnin ulkopuoleisiin kohteisiin tallentaminen (väliaikaistiedostoihin, ei-sarjallistettaviin luokkamuuttujiin) tekee vikasietoisuuden toteuttamisen mahdottomaksi
- Klusterointi käytännössä edellyttää klusterijäsenten identtistä konfigurointia ja ympäristön homogeenisuutta (kaikilla klusterijäsenillä on samat objektit) vaikka sovelluspalvelin ei tätä edellyttäisikään. Ylläpidettävyyden kannalta tämä merkittävästi selkeyttää kokonaisuuden hallintaa.
- Suuret klusteriryhmäkoot eivät kaikissa sovelluspalvelimissa järkevin ratkaisu, sillä ratkaisu voi lisätä kommunikoinnin määrää ja huonontaa skaalautuvuutta (esim. tuhlaamalla muistia replikoidessa kaikille klusterijäsenille istunnon tila), mikäli tähän ei ole varauduttu klusteroinnin suunnittelussa. Vaihtoehtona on jakaa klusteri aliryhmiin (mikäli tuettu) tai erillisiin klustereihin, joskin jälkimmäinen vaikeuttaa hallinnointia.

3.3 JNDI-nimipalvelu

JNDI nimipalvelu on ensimmäinen kosketuspinta asiakassäikeiden hakiessa istunto tai entiteettipapujen home tai remote stubia. Täten JNDI nimipalvelimen pitää olla tietoinen papujen sijainnista eri instansseissa. JNDI nimipalvelimen toteutukselle on seuraavat vaihtoehdot [Kan01b]:

- *itsenäiset JNDI nimipalvelut* (independent JNDI naming servers): kukin klusterin jäsen omaa itsenäisen JNDI-hakemiston, eikä vikasietoisuutta näin ollen tueta. Eduksi voidaan korkeintaan laskea vähentynyt verkkoliikenteen määrä verrattuna muihin ratkaisuvaihtoehtoihin.
- *proxy-palvelin* (EJB proxy service): JNDI-kutsut välitetään erilliselle proxy-palvelimelle, joka huolehtii kuormanjaosta ja edelleenreitityksestä. Mekanismi kasvattaa vasteaikoja, koska jokainen JNDI-kutsu tulee reitittää ensin proxy-palvelimelle ja sitten vasta kohteeseensa.
- *keskitetty JNDI nimipalvelu* (centralized JNDI naming server): klusterijäsenet käyttävät JNDI-rajapinnan kautta yhteistä hakemistoa
- *jaettu globaali nimipalvelu* (shared global JNDI naming server): klusterijäsenet mainostavat ryhmäkommunikointimekanismin välityksellä muille ryhmän jäsenille JNDI-hakemistonsa, jonka kukin ryhmän jäsen liittää.

Lisäksi kullakin ryhmän jäsenellä on oma paikallinen, yksityinen nimipalvelu, jossa jäsenen omistamat objektit sijaitsevat. Yksityinen nimipalvelu nopeuttaa paikallisten objektien käsittelyä. Sovelluspalvelintoimittajan tulee toteuttaa synkronointimekanismi eri JNDI nimipalvelimien hakemistopuiden ylläpitämiseksi (mikäli esimekiksi yksi ryhmän jäsen liittyy ryhmään tai poistuu ryhmästä) tai toteuttaa läpinäkyvästi JNDI-metodikutsuissa kutsun välitys eri JNDI nimipalvelimien kesken.

3.4 EJB-säiliö

Jaetuissa JNDI-nimipalveluissa klusteroituihin objekteihin olevat viittaukset ovat ”älykkäisiin” home ja remote stubeihin, jotka toteuttavat objektin vikasietoisuuden ja kuormanjaon sekä omaavat tiedon metodikohtaisesti järjestelmän tilan muuttumattomuudesta. [RAJ02, s.420-427]

3.4.1 Home, remote home ja remote stubien toimintaperiaate

Home ja remote home stubin osalta lisätoiminnallisuudella mahdollistetaan sekä instantioinnin kuormanjako että vikasietoisuus. Kutsuttaessa pavun Create()-metodia osoittuu käsky jollekin klusterin jäsenistä. Mikäli instantiointi epäonnistuu, tehdään läpinäkyvästi uusi instanstiointipyyntö toiselle klusterijäsenelle.

Remote stubin osalta käsittelylogiikka on riippuvaista pavun tyypistä.

Home, remote home ja remote stubiin tulevan sovelluspalvelintoimittajan älykkyyden toteuttaminen voi tapahtua joko toimittajan erillisellä kääntäjällä tai vaihtoehtoisesti, esim. JDK 1.3 Proxy-luokkaa käyttäen, jolloin sovelluspalvelintoimittaja voi ajonaikaisesti generoida stubin mukaanlukien vaadittavan älykkyyden.

3.4.2 Järjestelmän tilan muuttumattomuus

Etäkutsussa epäonnistuminen voi tapahtua pyynnön jälkeen, ennen metodin suorittamisen aloittamista, metodin suorittamisen alettua tai metodin suorittamisen jälkeen, ennenkuin asiakas saa vastauksen käsiteltäväksi.

Ainoastaan ennen metodin suorittamisen alkua tapahtuneita virhettä voitaisiin käsitellä riippumatta metodin sisällöstä. Tämä kuitenkin edellyttäisi, että etäasiakas

pystyisi erottamaan vaiheen kahdesta muusta, sillä näissä tapauksissa asiakkaalla ei ole tietoa, toteutettiinko metodi ja vaikuttaako metodin uudelleen toteuttaminen järjestelmän tilaan eri tavalla.

Jotta ongelma voidaan ratkaista sovelluspalvelimen osalta, edellytetään kaikkien papujen metodien osalta klusteroinnin toteuttamiseksi takuuta *järjestelmän tilan muuttumattomuudesta* (idempotence), mikäli metodi voidaan suorittaa useaan otteeseen samoilla parametreilla. [RAJ02, s.420-421]

3.4.3 EJB papujen kuormanjako ja vikasietoisuus

Papujen toteutuksessa on huomioitava ettei *paikallisen liittymän* (Local Interface) käyttö ei ole mahdollista klusteroidun palvelun osalta: vikasietoisuus edellyttää tarvittaessa toiselle klusterijäsenelle tehtäviä kutsuja ja lisäksi paikalliset liittymät käyttävät viitteitä kutsuparametrien välityksessä, eivät serialisoinnin kautta arvoja.

Tilattomien istuntopapujen (Stateless Session Bean) home stubin tai remote home stubin home.Create() –metodi ei todellisuudessa instantoi objektia, sillä sovelluspalvelin rakentaa jo intanssin käynnistyessä *säiealtaan* (thread pool), johon objekteja jaetaan. Sovelluspalvelin vastaa instantoinnista säiealtaan minimi- ja maksimiparametrien rajoissa. Tämän vuoksi metodissa ei tulisi olla mitään sovelluslogiikkaa, taaten järjestelmän tilan muuttumattomuuden. Tämä takaa myös remote home stubin läpinäkyvän viasta toipumisen.

Remote stubit voivat toteuttaa kuormanjakoa kutsumalla eri klusterijäseniä kuormanjakoalgoritmin mukaisesti, jotta säiealtaan resursseja käytetään mahdollisimman tasavertoisesti, välttäen tarpeettoman jonotuksen. Sovelluspalvelimissa on mahdollista *kiinnittää* (pin) remote stubit niin, että ne välittävät joko kaikki metdodikutsut tai tietyt metodikutsut ensisijaisesti yhteen paikkaan. Remote stubit voivat myös toteuttaa affiniteettia, jolloin kuormanjakoa ei toteuteta sokeasti, vaan fyysisellä palvelimella sijaitsevalta instanssilta, kutsutaan näitä ensisijaisesti.

Remote stubin vikasietoisuus on riippuvainen muuttumattomuudesta: mikäli otteeseen toteutettavalle metodille vikasietoisuus voidaan automaattisesti toteuttaa kutsumalla

toista klusterijäsentä, muussa tapauksessa vikasietoisuus joudutaan toteuttamaan sovellustasolla.

Tilallisten istuntopapujen (Stateful Session Bean) perustoiminnassa sovelluspalvelimet pitävät yllä erillistä cachea tilallisia istuntopapujen objekteja varten. Objektien instantiointi tapahtuu aina vasta Create()-metodin yhteydessä. Tyypillisesti cachen täyttyessä tai tilallisen istuntopavun objektin timeoutin yhteydessä sovelluspalvelin passivoi pavun ja siirtää tilan levylle. Tässä yhteydessä tapahtuva serialisointi ei kuitenkaan tarjoa vikasietoisuuden osalta mitään takeita. Tämän vuoksi klusterointi tyypillisesti tehdään cachesta erillisellä mekanismilla. Tilallisten istuntopapujen objektin kuormanjako ja vikasietoisuus toteutetaan tilan replikoinnilla, joka tyypillisesti tapahtuu kahdessa vaiheessa [RAJ02, s.424]: jokaisen metodin lopussa sekä transaktion kommitoinnin yhteydessä. Toteutusvaihtoehdot ovat vastaavat kuin HTTP-istunnon tilan replikoinnin toteutukselle.

Remote home stubit voivat toteuttaa kuormanjakoa Create()-metodin osalta. Kuitenkin tämän jälkeen, metodikutsut tulisi ohjata tilallisen istuntopavun pääobjektille.

Tilallisten istuntopapujen osalta Create()-metodin ja set()-metodien vikasietoisuutta ei voida toteuttaa, koska näihin aina liittyy järjestelmän tilan muuttaminen. Muissa tapauksissa vikasietoisuuden toteuttaminen jää kuitenkin sovelluskehittäjän vastuulle.

Entiteettipapujen (Entity Bean) klusteroinnin osalta on huomioitava tyypillisesti käytettävä Session Façade –suunnittelumalli [Sun02], jolloin istuntopavut kutsuvat entiteettipapua paikallisen liittymän kautta, poistaen tarpeen tehdä kuormanjakoa entiteettipavuille.

Entiteettipapujen vikasietoisuus perustuu aina tietokannan käsitellyn transaktiohallintaan. Kuitenkin e*nteettipapujen objektien välimuisti* (entity bean object cache) tuottaa rajoitteita klusteroinnin käytölle. Sovelluspalvelin pyrkii vähentämään ejbLoad() ja ejbStore() –metodien aktiivisuutta, mutta samalla klusteroinnissa eri instanssien välimuistien synkronointi tulee hallita.

Välimuistit on jaettavissa kolmeen erilaiseen toteutukseen [RAJ02, s.426-429]:

- *lukuvälimuisti* (read-only caches): välimuisti tallentaa vain lukuoperaatioita suorittavia entiteettipapujen objekteja [Sun01b, s.258-260]. Entiteettipavun deklaratiivisena transaktioattribuuttina on oltava joko Never, NotSupported tai Supports [Sun01b, s.357-359], mikä estää entiteettipavun mahdollisuuden aloittaa transaktio. Lukuvälimuistit toteuttavat objektien mitätöinnin joko julkaisemalla klusterijäsenille tieto read-cachessa olleen objektin muuttumisesta, mahdollistamalla erityisen pavun mitätöinnin toteuttavan metodikutsun tai määrittämällä aikaintervallin.
- *hajautettu jaettu välimuisti* (distributed shared objects cache): sovellustoimittajan toteuttama, kaikki klusterin jäsenet kattava luku- ja kirjoitusdatan välimuisti, joka edellyttää niin klusterin jäsenten kesken synkronointia kuin synkronointia tietokannan kanssa.
- *pääosin lukuvälimuisti* (read-mostly cache): pääosin lukuvälimuistialgoritmi hyödyntää lukuvälimuistin kaltaista objektivälimuistia, mutta mitätöi välimuistin objekteja silloin, kun näihin kohdistuu päivityksiä. Tämän mekanismin kautta vältetään hajautetun jaetun objketivälimuistin monimutkaisuus. Mitätöinnin toteutus läpi klusterin edellyttää kuitenkin luotettavaa klusterin kommunikointimekanismia.

Välimuistien käyttämäninen rajoittaa lisäksi pavun ja sovelluspalvelimen *vahvistusmenettelykäytännöissä* (Commit option) [Sun01b, s.186-187] sekä isolaatiotasovaihtoehtoihin [RAJ02, s.339-347].

Viestilähtöisten papujen (Message-Driven Beans) toimintaperiaate poikkeaa merkittävästi muista pavuista, koska pavut ottavat vastaan viestejä eivätkä omaa home tai remote stubia. Klusterointi liittyy sen sijaan JMS:n klusterointiin, jossa sovelluspalvelimen on kyettävä yhdistämään yhteen jonoon useita sovelluspalvelininstansseja. Pavuille tämä toteutus puolestaan on näkymätön.

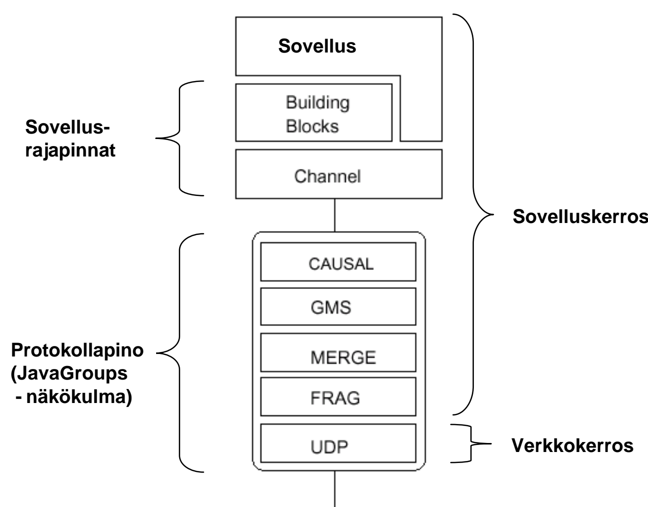
4 TOMCAT JA JBOSS KLUSTEROINTI

Tomcat 4.0 ja JBoss 3.0 klusteripalvelu rakentuvat JavaGroups ryhmäkommunikointimekanismin varaan. JBoss 3.0 ei mahdollista yksinkertaisimman hajautusasteen mukaista klusterointia, koska se ei sisällä web-säiliön vikasietoisuuspiirteitä. Nämä löytyvät osittain Tomcat 4.0:sta.

Kappaleessa esitellään JavaGroups (4.1), Tomcat 4.02 klusterointi (4.2) sekä JBoss 3.0 klusterointiratkaisu kerrosarkkitehtuuri (4.3)

4.1 JavaGroups

JavaGroups on dynaamista liittymistä ja poistumista tukeva, luotettavan ryhmäkommunikointimekanin mahdollistava toolkit-API. Toolkitin ensimmäinen versio julkaistiin 1998. Toolkittiä kehitetään edelleen yleisesti Javan ryhmäkommunikoinnin tukemiseksi, eikä se ole sidoksissa JBossin kehitykseen. Kuva 3 kuvaa JavaGroups 2.0:n mikroprotokolla-arkkitehtuurin sekä ohjelmointirajapinnat (API:t).



Kuva 3: JavaGroups protokollapino ja ohjelmointirajapinnat [Ban01, s.4]

JChannel-API mahdollistaa mikroprotokollilla monipuolisen kanavan märittelyn aina järjestyksettömän, luottamattoman UDP-kanavan määrityksestä totaalista järjestystä toteuttavaan TCP-kanavaan. Building Blocks API rakentaa kanavan päälle korkeamman tason, yksinkertaistetun ohjelmointirajapinnan. Raportti rajoittuu

klusteroinnissa käytettävälle UDP-JChannel kanavalle määriteltävien mikroprotokollien tarkasteluun.

Keskeisimmät mikroprotokollat vastaavat seuraavista toiminnallisuuksista [Ban01, s.33]:

- CAUSAL: vastaa kausaalisen järjestyksen toteuttamisesta
- GMS: vastaa jäsenten liittymisestä ja niiden eroamisesta. Kukin muutos ryhmän jäsenyyksissä johtaa uuden *näkymän* (view) muodostumiseen, jonka puitteissa viestijärjestystä voidaan hallinnoida. Tiedonlevitys voi tapahtua probabilistisella kommunikointimekanismilla (pbcast.GMS)
- MERGE2: Vastaa esimerkiksi verkkovian vuoksi jakautuneen ryhmän uudelleenyhdistymisen toteuttamisesta.
- FRAG: Vastaa suurten pakettien fragmentoinnista pienemmiksi sekä uudelleenkokoamisesta kohteessa.

Muita, sovelluspalvelinklusterointiin liittyviä mikroprotokollia ovat:

- FD: Ryhmän jäsenen vianhavaitseminen ja vikaantumisen epäilystä ilmoittaminen ryhmälle. Perustuu heartbeat-mekanismiin (jäsenet lähettävät sanoman, joka ilmoittaa kunkin saatavuudesta)
- FLUSH: keskeneräisten sanomien uudelleenjärjestys ennen näkymän muutosta. Keskeneräiset sanomat olemassa olevassa näkymässä priorisoidaan ja käsitellään, jonka jälkeen ryhmä rakentaa uuden näkymän tavanomaisesti.
- NAKACK: Varmistaa vastaanottajan sanoman tiedustelun sekä ja tarvittaessa pyynnön uudelleenlähetyksen. Voi perustua probabilistiseen ryhmäkommunikointiin (pbcast.NAKACK)
- PING: vastaa IP-multicast jäsenten löytämisestä. Tämän myötä ryhmän koordinaattori voidaan määrittää ja lähettää tälle ryhmään liittymisviesti.
- QUEUE: Ylläpitää jonoa, jonka kautta kaikki sanomat kierrätetään, mahdollistaen tilan ylläpitämisen sanomia varastoimalla.
- STABLE: poistaa sanomia, jotka kukin ryhmän jäsen on nähnyt. Perustuu probabilistiseen ryhmäkommunikointiin (pbcast.stable)
- STATE_TRANSFER: Sovellus hakee JChannel kanava kautta kaikki jäsenen kanavan kautta saamat sanomat. Edellyttää QUEUE-mikroprotokollaa.

- UNICAST: unicast-pakettien NACKACKkia vastaava mekanismi UDP-sanomille. Vastaa luotettavuudesta sekä FIFO-järjestyksestä (vrt. TCP ilman vuonhallintaa)
- VERIFY_SUSPECT: tarkistusmekanismi vikaantumisesta epäillyn jäsenen tilan tarkistamiseksi

4.2 Tomcat 4.0 klusterointi

Filip Hanikin Tomcat 4.0 klusterointitoteutus tarjoaa tilan muistissa replikointiin perustuvatn web-säiliön klusterointitoiminnallisuuden. Toteutus koostuu neljästä luokasta, välttäen Tomcatin lähdekoodeihin tehtäviä suoria muutoksia [Han01]:

- org.apache.catalina.session.InMemoryReplicationManager extends org.apache.catalina.session.StandardManager – StandardManagerista periytetty luokka, joka vastaa JChannelin luonnista, http-istunnon perustamisesta sekä ryhmäjäsenyydestä
- org.apache.catalina.session.ReplicatedSession extends org.apache.catalina.session.StandardSession – korvaa metodit tilanhallinnan metodit (setAttribute, removeAttribute, expire ja access), laajentaen näitä huomioimaan muut ryhmän jäsenet JavaGroupsin ja InMemoryReplicationManagerin välityksellä.
- org.apache.catalina.session.SessionMessage - sanoma, jota käytetään tiedotettaessa jäsenille uuden http-istunnon luonnista, http-istunnon tilan muutoksesta sekä http-istunnon tilan umpeutumisesta.
- org.apache.catalina.session.SerializablePrincipal – sarjallistaa security principal-luokan, mahdollistaen sen välittämisen tilatietojen yhteydessä sanomassa.

Tomcatin konfigurointitiedostoon (server.xml) määritetään InMemoryReplicationManager-luokan edellyttämät parameterit JChannel-kanavan luomiseksi esimerkiksi alla olevan mukaisesti [Han01]:

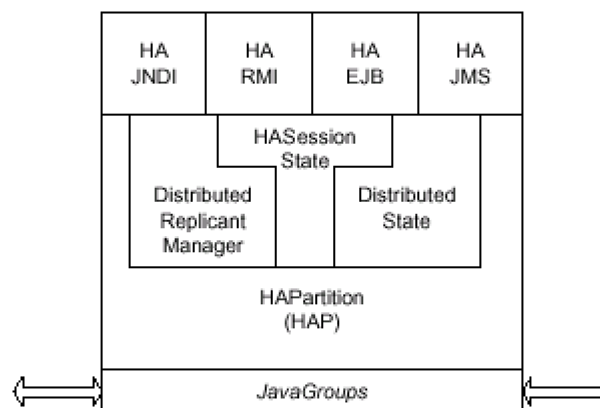
```
<Manager>
<ManagerclassName=
"org.apache.catalina.session.InMemoryReplicationManager"
protocolStack=
"UDP(mcast_addr=228.1.2.3;mcast_port=45566;ip_ttl=32):
PING(timeout=3000;num_initial_members=6):
FD(timeout=5000): VERIFY_SUSPECT(timeout=1500):
pbcast.STABLE(desired_avg_gossip=10000):
pbcast.NAKACK(gc_lag=10;retransmit_timeout=3000):
```

```
UNICAST(timeout=5000;min_wait_time=2000):MERGE2:FRAG:
pbcast.GMS(join_timeout=5000;join_retry_timeout=2000;
 shun=false;print_local_addr=false)">
```

Tomcatin suoraviivaisesti toteutettu replikointi ei ole skaalautuva, koska kukin jäsen omaa kaikkien jäsenten tilatiedot. Toteuttaessa suurempia järjestelmiä, jäsenmäärä tulisi jakaa esim. kahden tai kolmen palvelimen klustereihin.

4.3 JBoss 3.0 klusterointi

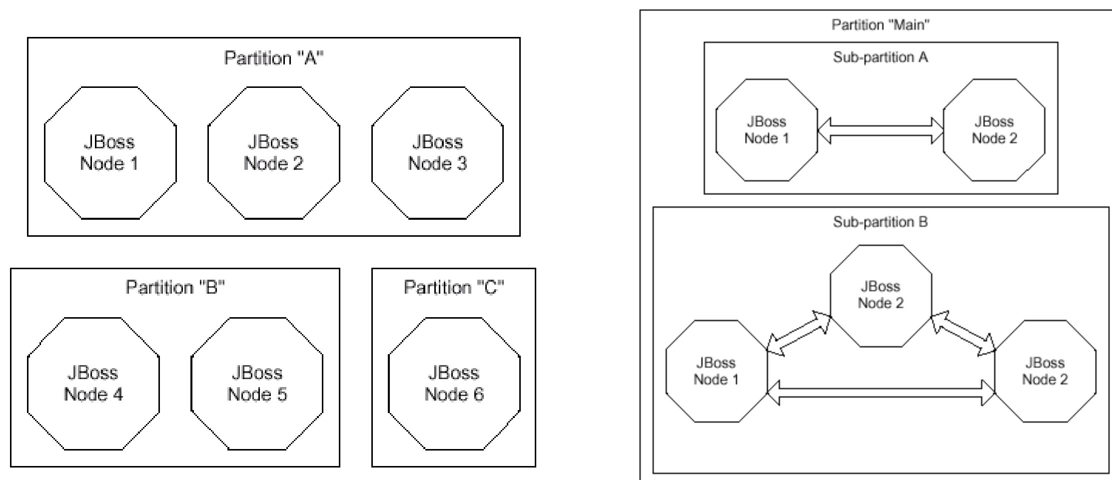
JBossin klusteroinnin kehitystyö alkoi maaliskuussa 2001 prototyypin rakentamisella Sacha Labourney toimesta. Bill Burke osallitui toimintaan elokuussa 2001. Klusterointi lisättiin JBoss 3.0:aan toukokuussa 2002.



Kuva 4: JBoss 3.0 klusteroinnin kerrosarkkitehtuuri [LB02, s.31]

4.3.1 HA-Partition

JBoss 3.0 klusterista käytetään termiä partitio. Partitiosta poistuminen ja siihen liittyminen tapahtuvat dynaamisesti. Klusterijäsenen on mahdollista saada sovellustasolle tieto jäsenen tilamuutoksesta. Tilan replikointi muistipohjaisesti tapahtuu kaikille jäsenille, rajoittaen ratkaisun skaalautuvuutta. Ratkaisua tullaan tulevaisuudessa laajentamaan alipartitioinnilla, jolloin alipartition jäsenet toteuttavat aliryhmäreplikointia.



Kuva 5: JBoss 3.0 partitiot vs. tulevat alipartitiomääritykset [LB02, s.6-8]

Kaikki partition jäsenet on oletusarvioisesti konfiguroitu identtisesti, joskaan tämä ei ole ehdoton vaatimus. Huolimatta klusterijäsenyydestä, klusteria ei voi hallinnoida kokonaisuutena.

HAPartition tuki edellyttää JBossin käynnistystä kattavimmassa all-server konfiguraatiossa. Mbean –määritys HAPartition käynnistämiseksi voidaan määrittää seuraavasti [LB02]:

```
<mbean code="org.jboss.ha.framework.server.ClusterPartition"
name="jboss:service=MySpecialNewPartition">
      <attribute name="PartitionName">MySpecialNewPartition</attribute>
      <attribute name="DeadlockDetection">true</attribute>
      <attribute name="PartitionProperties">
       UDP(mcast_addr=228.1.2.3;mcast_port=45566):PING:FD(timeout=5000):
       VERIFY_SUSPECT(timeout=1500):
       MERGE:NAKACK:UNICAST(timeout=5000;min_wait_time=2000):
       FRAG:FLUSH:GMS:STATE_TRANSFER:QUEUE
      </attribute>
</mbean>
```

Attribuutti DeadlockDetection määrittää, josko JavaGroups tarkistaa jokaisen sanoman välityksen yhteydessä mahdollista sanomajonojen välistä lukkiutumistilannetta

4.3.2 Distributed Replicant Manager

Distributed Replicant Manager (DRM) mahdollistaa RMI-palvelinten klusteroinnin ylläpitämällä kunkin ryhmänjäsenen sarjallistettua dataa. DRM synnyttää tapahtuman, mikäli ryhmän näkymässä tapahtuu muutos.

4.3.3 Distributed State -palvelu

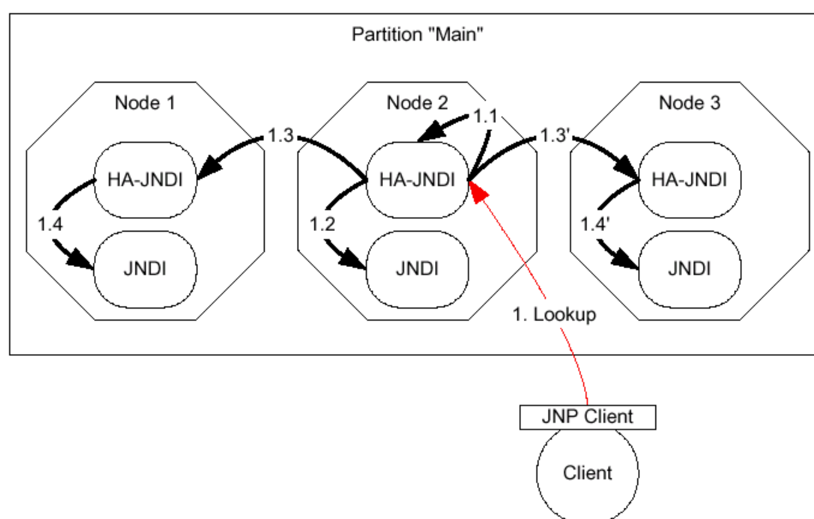
Distributed State –palvelu tarjoaa kaikki klusterin jäsenet kattavan hakemiston, johon kaikilla jäsenillä on mahdollisuus tallentaa kaikille muille jäsenillekin tarkoitettua tietoa.

Palvelu mahdollistaa myös mihin tahansa luokkaan kohdistuvien tapahtumien tilaamisen. Luokkaa tiedotetaan tällöin *tapahtumalla* (event), mikäli luokkaa käytetään jonkin jäsenen toimesta, mahdollistaen näin ollen esim. tilatiedon ulkopuolisen tiedon muutoksista saatavat ilmoitukset.

4.3.4 HA-JNDI

JNDI toteutus perustuu partition kattavaan (globaaliin) JNDI kontekstiin, jota kaikki partition jäsenet ensisijaisesti käyttävät. Globaalin JNDI:n konteksti replikoidaan kaikille partition jäsenille, jotta JNDI-hakemiston vikasietoituus ja kuormanjako voidaan toteuttaa. JNDI API sovelluksille pysyy muuttumattomana. Paikalliset JNDI-hakemistot ovat mukana edelleen toteutuksessa, mikäli replikointimekanismissa on jonkinlainen ongelma. Uuden toiminnallisuuden käyttöönotto ei edellytä sovelluskoodimuutoksia, toteutus perustuu InitialContext( ) –metodin uudelleentoteutukseen.

JBoss 3.0:ssa HA-JNDI:n käyttö edellyttää JNP-JNDI:tä, LDAP-pohjainen JNDI:n hakemisto ei ole tuettua.



Kuva 6: HA-JNDI ja Sovelluksen JNDI loopkup [LB02, s.19]

1: Sovellus suorittaa tavalliseen JNDI-kutsun

1.1: Sovelluksen lookup-haku ohjautuu globaaliin HA-JNDI-palveluun

1.2: Mikäli vastausta ei löyty globaalista hakemistosta, ohjautuu palvelu paikalliseen JNDI-hakemistoon

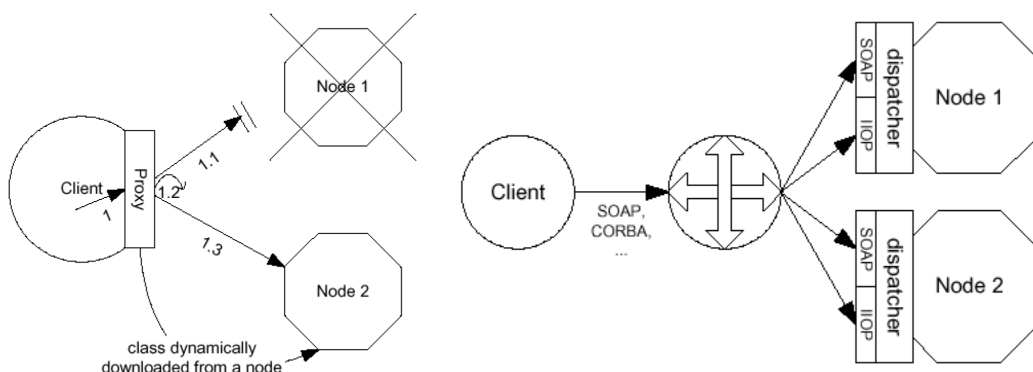
1.3 ja 1.3’: Mikäli paikallisesta JNDI-hakemistosta ei löydy, haetaan muiden jäsenten globaalista JNDI-instanssista

1.4 ja 1.4’: Toiset partition jäsenet hakevat paikallisista JNDI-hakemistoista, mikäli vastaus ei löydy omasta globaalista hakemistosta

HA-JNDI toteutus tukee täten läpinäkyvästi ympäristöjä, joissa klusterin jäsenille ei ole. Nimiavaruus on kuitenkin näkyvä läpi klusterin, joten kahdessa eri instanssissa sijaitsevat, sisällöltään eroavat, samannimiset luokat sekoittuvat keskenään eikä ole varmuutta siitä

4.3.5 HA-RMI

JBoss EJB klusterointi edellyttää vähintään JDK 1.3. Kuormanjaon ja vikasietoisuuden toteutus stubeissa perustuu ajonaikaisesti luotavaan älykkyyteen. Tämän seurauksena ei-RMI –asiakkailla toteutus poikkeaa: mm. SOAP ja IIOP tukeutuvat erilliseen, keskitettyyn dispatcher-prosessiin, joka välittää tämän edelleen ryhmäjäsenelle. Vikasietoisuuden toteuttaminen dispatcherin osalta edellyttää sovelluspalvelimen ulkopuolista ratkaisua.



Kuva 7: JBoss stubit vs. ei-RMI asiakkaan toteutus [LB02, s.10-11]

4.3.6 HA-EJB

Tilattomilla istuntopavuilla ei ole mahdollisuutta määrittää kiinnitystä tai affiniteettia. Pavun klusterointi määritellään jboss.xml –tiedostoon pavun määrityksiin seuraavasti [LB02]:

```
<clustered>True</clustered>
<cluster-config>
     <partition-name>DefaultPartition</partition-name>
     <home-load-balance-policy>
          org.jboss.ha.framework.interfaces.RoundRobin
     </home-load-balance-policy>
     <bean-load-balance-policy>
          org.jboss.ha.framework.interfaces.RoundRobin
     </bean-load-balance-policy>
</cluster-config>
```

Tuetut kuormanjakovaihtoehdot ovat Round-Robin, ensimmäinen saatavilla oleva (FirstAvailable) tai jokin oma toteutus.

Tilallisten istuntopapujen tilanhallinta tapahtuu HA SessionState –palvelun välityksellä, jolle on mahdollista määrittää mm. tilatietojen säilytysaika. Pavun klusterointi määritellään jboss.xml –tiedostoon seuraavasti [LB02]:

```
<clustered>True</clustered>
<cluster-config>
     <partitionname>DefaultPartition</partition-name>
     <home-load-balance-policy>
          org.jboss.ha.framework.interfaces.RoundRobin
     </home-load-balance-policy>
     <bean-load-balance-policy>
          org.jboss.ha.framework.interfaces.FirstAvailable
     </bean-load-balance-policy>
     <session-state-manager-jndi-name>
          /HASessionState/Default
     </session-state-manager-jndi-name>
</cluster-config>
```

Entiteettipapujen klusterointi ei tue hajautetun välimuistin käyttöä (eli sovelluspalvelin käyttää commit optio C:tä [Sun01b, s.186]). Synkronointi edellyttää transaktioissa vahvinta isolaatiotasoa SERIALIZABLE [RAJ02, s.339-347].

Pavun klusterointi määritellään jboss.xml –tiedostoon seuraavasti [LB02]:

```
<clustered>True</clustered>
<cluster-config>
     <partition-name>DefaultPartition</partition-name>
     <home-load-balance-policy>
          org.jboss.ha.framework.interfaces.RoundRobin
     </home-load-balance-policy>
```

```
        <bean-load-balance-policy>
            org.jboss.ha.framework.interfaces.FirstAvailable
        </bean-load-balance-policy>
    </cluster-config>
```

4.3.7 HA-JMS

JBoss 3.0 ei tue JMS topicien tai jonojen vikasietoisuutta. Ominaisuus on työn alla.

5 YHTEENVETO

Sovelluspalvelinklustereiden markkinajohtavat sovelluspalvelimet ovat toteutukseltaan suhteellisen kypsiä. Sovelluspalvelimen konfiguroinnin mutkattomuus kuitenkin on eri asia kuin sen piirteiden käyttäminen tarkoitetulla tavalla.

Sovelluspalvelinklustereiden toteutus ja käyttöönotto on osittain riippuvainen hajautusasteesta, mutta keskeisemmin hajautusasteen valintaan vaikuttavat muut tekijät. Hajautusaste ei myöskään kuormanjaon kannalta ole keskeisin tekijä, mutta vaikuttaa vikasietoisuuteen sekä skaalautuvuuteen suurissa järjestelmissä.

Säiliöiden toteutuksissa toiminta perustuu passiiviseen replikointiin. Web-säiliön toteutuksessa kaikille klusterijäsenille tapahtuva tilan replikointi rajoittaa skaalautuvuutta, vaikkakin yksinkertaistaa järjestelmän ylläpitoa verrattuna esim. aliryhnmätasoiseen replikointiin tai klusterin useaan irralliseen klusteriin jakamiseen. EJB-säiliöän osalta paikallinen liittymä sekä sovelluspalvelimen välimuistin käyttö ovat tilan muuttumattomuuden ohella erityisesti huomioitavia asioita varsinkin ratkaisuissa, joissa näiden kahden tuomiin suorituskykyetuihin ollaan jo toteutuksessa tukeuduttu.

Klusteroinnin toteutuksessa valitut sovelluspalvelimen suunnitteluvaihtoehdot määrittävät järjestelmän skaalautuvuuden ja vikasietoisuuden, mutta niistä huolimatta J2EE sovelluspalvelinklusterit kattavat vain osan järjestelmän vikasietoisudeen tuottamisesta hajautetuissa ratkaisuissa. J2EE eivät myöskään tuota vikasietoisuutta tai skaalautuvuutta järjestelmille, joissa liiketoimintasovelluksen suunnittelussa ei ole huomioitu kyseisiä asioita tai joissa järjestelmän hallinnointikäytännöt tai ympäristössä toimivien osapuolen tiedot ovat puutteellisia.

LÄHTEET

Ban99 Bela Ban: "Design and Implementation of Reliable Group Communication", Cornell University, 1999 citeseer.nj.nec.com/ban98design.html [26.10.2002]

Ban01 Bela Ban: "JavaGroups 2.0 Users Guide", Fujitsu Network Communications, December 21st, 2001 www.javagroups.com/javagroupsnew/docs/ug.html [26.10.2002]

Bir96 Kenneth Birman, *Building Secure and Reliable Network Applications,* Manning Publishing Company, 1996

CDK01 G. Couloris, J. Dollimore, T. Kindberg, *Distributed Systems Concepts and Design (3rd edition),* Addison-Wesley, 2001

GPD02 Damian Guy, Allan Packer, Tom Daly: "High Availability for J2EE Platform-based Application", Sun Best Practices Developers Handbook, January 30, 2002 dcb.sun.com/practices/devnotebook/j2ee_applications.jsp [26.10.2002]

Han01 Filip Hanik: "In Memory Session Replication in Tomcat 4", The ServerSide April 2001, www.theserverside.com/resources/articles/Tomcat/article.html [26.10.2002]

Kan01a Abraham Kang: "J2EE Clustering, Part 1", JavaWorld, February 2001 www.javaworld.com/javaworld/jw-02-2001/jw-0223-extremescale.html [26.10.2002]

Kan01b Abraham Kang: "J2EE Clustering, Part 2", JavaWorld, August 2001 www.javaworld.com/javaworld/jw-08-2001/jw-0803-extremescale2.html [26.10.2002]

LB02 Sacha Labourey, Bill Burke, *JBoss 3.0 Clustering*, The JBoss Group, June 2002

RAJ02 Ed Roman, Scott W. Ambler, Tyler Jewell, *Mastering Enterprise JavaBeans, 2nd edition,* John Wiley and Sons, 2002

Sun01a Sun Microsystems, *J2EE Specification (Version 1.3),* Final Release, January 27, 2001 java.sun.com/j2ee/download.html#platformspec [26.10.2002]

Sun01b Sun Microsystems, *Enterprise JavaBeans Specification (Version 2.0),* Final Release 2, August 14, 2001
java.sun.com/products/ejb/docs.html [26.10.2002]

Sun02 Sun Microsystems, *Core J2EE Pattern Catalogue*
java.sun.com/blueprints/corej2eepatterns/Patterns/index.html
[02.11.2002]