

//global/global



Kandidatutkielma

Tietojenkäsittelytieteen kandiohjelma

Järjestelmäintegraatioiden tyylit ja suunnittelumallit

Joona Halonen

10.5.2024

MATEMAATTIS-LUONNONTIETEELLINEN TIEDEKUNTA
HELSINGIN YLIOPISTO

Yhteystiedot

PL 68 (Pietari Kalmin katu 5)
00014 Helsingin yliopisto

Sähköpostiosoite: info@cs.helsinki.fi
URL: <http://www.cs.helsinki.fi/>

Sisältö

1	Johdanto	1
2	Integraatioiden lähestymistavat	2
2.1	Integraatiotasot	2
2.2	Tekniset lähestymistavat	2
3	Sanomapohjaiset suunnittelumallit	8
3.1	Sanomajärjestelmät	9
4	Suunnittelumallien kehitystä	17
4.1	Kirjailijoiden omaa modernisointia	17
4.2	Tilalliset protokollat	18
4.3	Virtausprotokollat	19
4.4	Virheenhallinta	20
4.5	Formalisointi	20
5	Yhteenveto	21
	Lähteet	23

1 Johdanto

Järjestelmäintegraatiot (*Enterprise Application Integration, EAI*) tarkoittaa ohjelmistoja, jotka yhdistävät eli integroivat organisaation olemassa olevia erillisiä järjestelmiä toimimaan yhdessä.

Tarve EAI:lle kehittyi kun organisaation jo olemassa olevien järjestelmien hajautunut tieto piti yhdistää tiedonkulun optimoimiseksi ja automatisoimiseksi. Esimerkiksi organisaation asiakkuudenhallinnasta vastaavan ohjelmiston pitäisi pystyä keskustelemaan myynninhallinnointiohjelmiston kanssa pitääkseen asiakastilastot ajan tasalla. Järjestelmät jäivät helposti erillisiksi saarekkeiksi eivätkä jakaneet tietoa keskenään. Syntyi niin sanottuja "savupiippujärjestelmiä", joissa järjestelmä oltiin suunniteltu pitämään data esimerkiksi firman yksittäisen osaston sisällä, eikä järjestelmän suunnitteluvaiheessa oltu huomioitu datan jakamista; järjestelmästä puuttui esimerkiksi ulospäin suuntavat ohjelmointirajapinnat.

Järjestelmäintegraatio mahdollistaa näiden eri järjestelmien keskinäisen keskustelun toisilleen. Näin voidaan rakentaa yhteinen järjestelmä jo olemassa olevista palasista [21, sivu 15]. Useissa tapauksissa organisaation sisäisiä järjestelmiä yhdistetään myös ulkoisiin järjestelmiin tai muiden organisaatioiden kanssa [20]. Organisaation olemassa olevat järjestelmät voi olla kehitetty eri käyttöjärjestelmille, hyödyntävät eri ohjelmointikieliä tai tietokantaratkaisuita. Väliin tarvitaan integraatio, joka hoitaa kommunikoinnin ohjelmistojen välillä

Vuonna 2020 EAI-markkinan arvioitiin olevan 9,26 miljardin dollarin (USD) arvoinen [22] ja koostuu ohjelmistojättien (Microsoft, Oracle, Salesforce IBM) ja pienempien yksittäisten yritysten (Workato, SnapLogic, Tibico) tarjoamista järjestelmistä.

2 Integraatioiden lähestymistavat

Esittelen tässä luvussa suosituimpia jäsentely ja lähestymistapoja järjestelmäintegraatioille. Eri lähestymistavat perustuvat viitatuimpaan järjestelmäintegraatioarkkitehtuurin kirjallisuuteen. Luvun tarkoituksena on antaa kattokategorioita erilaisille teknisille lähestymistavoille ja antaa historiallista kontekstia eri arkkitehtuurisuuntauksille.

2.1 Integraatiotasot

Teoksessaan [21] luokittelee integraatioiden lähestymistavat neljään eri tasoon

1. Datataso: Dataa liikutellaan eri tietovarastojen välillä ja mahdollinen datan prosessointi ja muokkaus toteutetaan siirron yhteydessä. Kyseisellä menetelmällä on myös paljon yhtäläisyyksiä tyypillisen datavaraston (*data warehouse*) toteutuksen kanssa.
2. Rajapintataso: Tasolla pyritään hyödyntämään tarkoitusta varten tilattujen tai paketoitujen sovellusten, esimerkiksi SAP, PeopleSoft, tarjoamia ohjelmointirajapintoja.
3. Metoditaso: Tällä tasolla pyritään hyödyntämään jaettua sovelluslogiikka. Esimerkiksi hyödyntämällä sovelluspalvelimia (*application server*) metodienjakamiseen tai käyttämällä hajautettuja objekteja ja/tai etäkutsuja (*Remote Procedure Call, RPC*) hyödyntäviä teknologioita.
4. Käyttöliittymätaso: Integraatioiden yhdistävä tekijä on olemassa olevan sovelluksen käyttöliittymä kun sovellus ei tarjoa muita keinoja datan jakamiseen. Tämä toteutetaan hyödyntymällä ruudun tiedonharavointi tekniikoita (screen scraping).

2.2 Tekniset lähestymistavat

Järjestelmäintegraatioiden arkkitehtuurin määrää pitkälti liiketoiminnan tarpeet ja niiden asettamat vaatimukset. Integraatioiden tekniset lähestymistavat voi luokitella neljään erilaiseen yläkategoriaan [16, sivu 64].

1. Tiedostojen siirto
2. Jaettu tietokanta
3. Etäproseduuriherätys
4. Sanomat

Tiedostojen siirto: Tiedostojen siirrosta integroitavat sovellukset voivat toimia itsenäisesti ja yhden sovelluksen muutokset eivät vaikuta toisen sovelluksen toimintaan kunhan sovellukset toimivat sovitulla tiedostotyyppillä ja tiedoston nimeämisstrategioilla. Integroijan vastuulla on taata, että tiedostot muutetaan toisen sovelluksen ymmärtämään muotoon. Lähestymistapana tiedostojen siirto on yksinkertainen eikä vaadi lisätyökaluja, koska yleisten tiedostotyyppien (JSON, XML, CSV) tuki löytyy käytännössä jokaisesta ohjelmointikielestä tai integraatiotyökalusta. Data-analytiikassa ja datavarastojen saralla on myös muita yleisiä tiedostomuotoja kuten Apache Parquet, jolle löytyy myös tuki useista ohjelmointikielistä ohjelmistokirjaston muodossa [3]. Sovitusta tiedostomuodosta tulee käytännössä integroitavien sovellusten välinen rajapinta.

Tiedostojen siirron heikkouksina on tiedostojärjestelmäoperaatioiden hallinnointi ja datan synkronointi. Integraatiokehittäjien vastuulle jää siis tiedostojen lukitseminen kirjoitusoperaatioiden ajaksi tai kirjoitusten ajoittaminen jotta ne eivät mene päällekkäin lukemisen kanssa, tiedostojen nimeämiskäytännöt, tiedostojen arkistointi ja poistaminen. Jos integroitavilla sovelluksilla ei ole pääsyä samoille levyille niin kehittäjien ratkaistavaksi jää myös tiedoston siirtäminen oikealle laitteelle. Datan synkronointi järjestelmien välillä tuo oman haasteensa, koska tiedostojen siirtoa tapahtuu yleensä harvakseltaan. Esimerkiksi jos asiakkuudenhallintajärjestelmä tuottaa tiedostoja datan synkronointia varten vain kerran päivässä ja laskutusjärjestelmä lähettää laskut aikaisemmin samana päivänä, niin osa laskuista on jo saattanut lähteä vanhaan osoitteeseen, jos asiakas on päivittänyt osoitetietojaan aikaisemmin saman päivän aikana.

Jaettu tietokanta: Jaetun tietokannan etuna tiedostojen siirtoon on muutosten nopea propagoituminen eri sovelluksille. Samassa tietokannassa uusin tieto on saatavilla eri sovelluksille lähes välittömästi. Nopea tiedon liikkuminen tekee virheiden havaitsemisesta ja korjaamisesta helpompaa. Etuna on myös, että tietokantojen datamallit takaavat yhtenevän datan esitysmuodon verrattuna tiedostojen siirtoon. Datan synkronoinnista ja kirjoitus- ja lukuvuoroista vastaa tietokannan hallintajärjestelmä (*DBMS, Database Management System*) ja transaktioiden hallinnointijärjestelmä antaa hyvät työkalut datan eheyden takaamiselle.

Gregor Hohpen ja Bobby Woolfin teos [16, sivu 69] korostaa myös SQL-pohjaisten relaatiotietokantojen yleisyyttä. Integraatiokehittäjien ei tarvitse opetella uutta teknologiaa tai taistella uuden tiedostoformaatin kanssa vaan kehittäjät voivat työskennellä laajalti tunnettujen relaatiotietokantojen parissa. Valtaosa ohjelmointikielistä ja kehitystyökaluista tukee SQL:n kanssa työskentelyä joten jaetun tietokannan kanssa työskentely on suoraviivaista ja adoptointi helppoa.

Saman tietokannan käyttö estää datan tulkitsemiseen liittyvien ongelmien, kuten semanttisen dissonanssin (*semantic dissonance*) pitkittymistä, missä samaa dataa voidaan tulkita ristiriitaisilla tavoilla. Koska integroitavat sovellukset käyttävät samaa datalähdettä, niin nämä tulkintakysymykset on kohdattava varhaisessa vaiheessa integraatiokehitystä, eikä vasta tuotannossa jossa data voi olla jo yhteensopimatonta tulkintaeroista johtuen.

Jaetun tietokannan suunnitteluhaasteisiin sisältyy yhtenäisen skeemaan suunnittelu, jota useampi eri sovellus pystyy tehokkaasti hyödyntämään. Usein useamman sovelluksen tuomat vaatimukset johtavat monimutkaiseen tietokantaskeemaan jonka käytön kehittäjät kokevat haastavaksi. Yhtenäisen skeeman suunnittelua voi myös vaikeuttaa "poliittiset" haasteet, koska tietokannan suunnittelu voi johtaa aikataulujen venymiseen ja kommunikaatiohaasteisiin tietokantaa hyödyntävien eri yksiköiden välillä. Lisää suunnitteluhaasteita tuo ulkoiset ohjelmistot. Lähes poikkeuksetta kaupalliset ohjelmistot tukevat vaan omaa ohjelmiston mukana tulevaa tietokantaformaattia eivätkä taivu siitä poikkeavaan tietokantaskeemaan. Vastaavia haasteita tuo sovellukset jotka on peritty toiselta organisaatiolta esimerkiksi yrityskaupan yhteydessä. Sovellusten jälkeinpäin tehtävä jatkokehittäminen jaettua tietokantaa hyödyntäväksi on yleensä työlästä ja kallista. Kun jaettuun tietokantaan yhdistettyjen sovellusten määrä lisääntyy niin ratkaisu voi aiheuttaa suorituskyyhaasteita, varsinkin jos luku- ja kirjoitusoperaatiot kohdistuvat vaan muutamaaan tietokantatauluun. Jos sovellukset on hajautettu useammalle laitteelle ja tietokanta niiden kanssa, jotta sovelluksilla on lokaali pääsy kantaan, niin hajauttaminen tuo omat haasteensa. Pääosin datan hajauttamistaktiikkojen muodossa ja lisää näin ratkaisun kompleksisuutta.

Etäproseduuriherätys: (*Remote Procedure Invocation*) Edelliset lähestymistavat keskittyivät pääosin datan jakamiseen, mutta näissä lähestymistavoissa pienet datanmuutokset voivat johtaa eri toimintoihin useiden sovellusten taholta. Osoitteen vaihto voi olla yksinkertainen kentän muutos tai laukaista useita rekisteröinti- ja lakiprosesseja useassa eri sovelluksessa. Jaettu tietokanta ei mahdollista minkäänlaista datan kapselointia ja tämä yksi iso datalähde tekee datamuutosten havaitsemisesta ja muutosten vaatimien prosessien

aktivoimisesta haastavaa. Tiedostojen siirto tarjoaa suoraviivaisen tavan reagoida datan muutoksen, mutta tämä tapahtuu yleensä viiveellä johtuen tiedostojen synkronoimisen haasteista. Jaetun tietokannan kapseloimattomuus tarkoittaa myös integraatioiden ylläpidon joustamattomuutta. Muutokset yhdessäkään integroidussa sovelluksessa vaikuttavat jaettuun tietokantaan ja tietokantamuutokset voivat aiheuttaa kauas kantautuvia muutoksia tietokantaa käyttävien sovellusten kesken.

Etäproseduuriherätys mahdollistaa mekanismin jossa sovellus voi kutsua toisen sovelluksen funktiota, jakaa vain tarvittavan datan ja kutsua funktiota joka kertoo datan vastaanottajalle miten toimia jaetun datan kanssa. Jos sovellus tarvitsee toisen sovelluksen dataa se voi kysyä sitä siltä suoraan. Vastaavasti jos sovelluksen tarvitsee muokata toisen sovelluksen dataa niin se voi tehdä funktiokutsun. Jokainen sovellus vastaa oman datansa eheydestä ja jokainen sovellus voi tehdä muutoksia omaan dataansa, vaikuttamatta muiden sovellusten tilaan.

Etäproseduuriherätyksen mahdollistavat teknologiat ovat myös yleisiä ja tuttuja kehittäjille. Etäproseduurikutsu (*Remote Procedure Call, RPC*) teknologiat ja kirjastot ovat tunnettuja ja yleisesti käytettyjä. Teoksessa [16, sivu 71] Martin Fowler listaa CORBA, COM, .NET Remoting ja Java RMI esimerkkeinä ja mainitsee, että web-palveluiden yleistyessä http-yhteyksiä hyödyntävät lähestymistavat kuten SOAP ja XML ovat tulleet kehittäjien suosikeiksi. Varsinkin kun http-yhteyksien kanssa on helppo työskennellä, koska useimpien yritysten palomuurit sallivat http-liikenteen. Teoksen julkaisun jälkeen REST ja JSON ovat pitkälti korvanneet SOAP:in ja XML:än web-palveluiden suosituimpana lähestymistapana.

Etäproseduuriherätyksellä on mahdollisuus vähentää semanttista dissonanssia, koska sovellukset voivat tarjota usean erillaisen rajapinnan samalle datalle. Eri asiakasohjelmille voidaan tarjota erilainen datanesitysmalli riippuen siitä, mikä asiakasohjelma on kyseessä. Tämä antaa enemmän mahdollisuuksia esittää datan useammalla eri tavalla verrattuna pelkkään relationaaliseen malliin. Useammat eri rajapinnat tarkoittavat lisää työtä integraatiokehittäjille datan muokkaamisen parissa ja integroitaviensovellusten täytyykin neuvotella mitä rajapintoja ne tulevat toisiltaan käyttämään.

Etäproseduuriherätyksen helppous kehittäjille voi myös olla sen haittapuoli jos integraatiokehittäjät eivät tiedosta etäkutsujen suorituskyky- ja luotettavuuseroja verrattuna paikallisiin kutsuihin. Useat etäkutsut voivat kasaannuttaa näitä ongelmia ja johtaa hitaaseen ja epäluotettavaan järjestelmään.

Vaikka etäproseduuriherätyksen mahdollistama datan kapselointi vähentää sovelluksen

kytköksiä karsimalla suuren yhteisen datalähteen, niin se voi silti aiheuttaa solmukoh-tia, erityisesti kun kyse on jaksossa - tietyssä järjestyksessä tehtävistä- operaatioista. Inte-graatiojärjestelmistä näistä muodostuu helposti ongelma, koska vastaavat kytkökset eivät välttämättä aiheuttaisi ongelmia yksittäisessä sovelluksessa, mutta useamman sovelluksen integraatiossa lisäkytkökset tarkoittavat lisäviivettä ja ylimääräisiä verkkokutsuja.

Sanomat: (*Messaging*) Aikaisemmat integraatioiden lähestymistavat keskittyivät joko da-tan tai toiminnallisuuden jakamiseen. Gregor Hohpen ja Bobby Woolfin mukaan yleinen integraationkehityksen haaste on saada eri järjestelmät toimimaan yhdessä mahdollisim-man viipettä ilman, että järjestelmien välillä on kytköksiä, jotka tekevät järjestelmäs-tä epäluotettavan joko sovelluksen suorittamisen tai kehittämisen kannalta [16, sivu 72]. Tiedostojen siirrossa datan siirtyminen ei ole tarpeeksi viipeetöntä ja sovellusten välinen toiminta tarpeeksi sujuvaa vaikka lähestymistapa estääkin rajoittavien kyskösten muo-dostamisen. Jaetussa tietokannassa data on jaettu ja datan muutokset ovat responsiivisia, mutta kaikki sovellukset ovat kytköksissä samaan tietokantaan ja lähestymistapa ei mah-dollista sovellusten yhteistä toimintaa. Etäproseduuriherätyksen heikkoudet olivat yleiset hajautettujenjärjestelmien sudenkuopat, kuten verkkoviiveet ja verkon luetattavuus, var-sinkin jos eäkutsuja käytetään samallalailla kuin paikallisia kutsuja ja lähestymistavassa sovellusten pitää jakaa tietoa toistensa rajapinnoista, mikä lisää kehitystä vaikeuttavien kytkösten määrää.

Sanomien käyttö erityisesti asynkroniseen viestintään on aikaisemmin esiteltyjen lähesty-mistapojen parhaiden puolien yhdistelmä [16, sivu 73]. Sanomien käyttö mahdollistaa pie-nien tiheästi kulkevien datapakettien lähetyksen ja tallentamisen ja tiedosto-operaatioiden yksityiskohtien abstraktoinnin. Tämä mahdollistaa nopeat skeeman muutokset vastaten yrityksen tarpeisiin. Sovellukset pystyvät jakaa toiminnallisuuksian lähettämällä sanoman toisilleen joka herättää esimerkiksi datanmuokkausproseduurin. Asynkroninen viestintä ei taas vaadi vastaanottajan olemaan saatavilla lähestyhetkellä ja asynkronine viestintä ohjaa kehittäjiä ymmärtämään, että etäyhteyksien käyttäminen on hitaampaa ja suun-nittelemaan korkeamman koheesion komponentteja, jolloin etänä tehtävien operaatioiden käyttö on harkitumpaa. Sanomapohjaiset järjestelmät myös mahdollistavat tiedostojen siirron kaltaisten löyhien kytkösten käytön. Sanomia voidaan muokata kesken lähetyksen ilman, että lähettäjä- tai vastaanottajasovelluksen tarvitsee olla tietoinen muokkausope-raation yksityiskohdista. Tämä mahdollistaa integroijien yleislähettävän (*broadcast*) sa-nomia useammalle vastaanottajalle, valitsevan yhden vastaanottajan useamman joukosta tai valita useasta muunlaisesta topologiasta jotka sallivat integraation irrottamisen sovel-

luksen kehitysprosessista. Sanomien tiheä lähettäminen mahdollistaa säännöllisen datan jakamisen lisäksi toiminnallisuutta. Käsittelyprosessi voidaan käynnistää heti kun yksittäinen sovellus saapuu ja asynkronisten kutsujen avulla lähettävän sovelluksen suoritus ei keskeydy odottamaan vastausta.

Sanomien lähetyksen tiheä datanvaihto ei kuitenkaan estä semanttisen dissonanssin syntymistä, varsinkin kuin datan esitysmuoto voi vaihtua useasti sanomia muokatessa. Sanomien lähetyksen tiheys ei myöskään täysin poista samoja datan synkronointihaasteita, joita tiedostojen siirrossa ilmeni. Sanomien siirrossta on jonkin verran viiveitä ja niiden ajoituksella tulee edelleen olemaan merkitystä. Asynkronisuus tuo myös lisä haasteensa integraation kehitys vaiheessa. Testaus ja sovellusten virheenpaikkannus tulee olemaan monimutkaisempaa sanomien lähetyksen rinnakkaisuuden takia ja vaati integraatiokehittäjiltä jonkin verran lisätututtelua. Sanomien käytön löyhät kytkennät lisäävät integroitavien sovellusten koheesioita, mutta tarkoittavat kuitenkin vaikeammin ylläpidettävän "liimakoodin" tarvetta jotta integraatiot saadaan toimimaan yhdessä.

Edellä mainitut haasteet tarkoittavat sanomapohjaisille järjestelmille suunniteltuja lähestymistapoja ja arkkitehtuureja mitkä toistuvat järjestelmissä niiden yksittäisistä eroista huolimatta.

3 Sanomapohjaiset suunnittelumallit

Edellisessä luvussa esitellyistä neljästä kategoriasta Hohpe ja Woolf suosivat sanomien käyttöä integraatoratkaisuissa ja suurin osa teoksesta keskittyy sanomapohjaisen suunnittelumallien esittelyyn [16, sivu 76]. Nämä järjestelmäintegraatioiden suunnittelumalleina (*enterprise integration patterns, EIP*) tunnetut kehitysohjeet ovat vaikuttaneet useamman eri integraatioalustan arkkitehtuuriin.

Vuonna 2018 viimeisintä tekniikka edustavista 48:sta integraatioalustasta yksitoista tutki EIP-malleja [7]. Kaksistoista vuotta kirjan julkaisun jälkeen pidetyssä haastattelussa kirjailijat sanovat, että useimmat avoimen lähdekoodin liikepalveluväylät (*enterprise service bus, ESB*) ovat omaksuneet teoksessa esitellyn mallikielen (*pattern language*) [27]. Samassa haastattelussa Hohpe ja Woolf toteavatkin, että kirjan pysyminen relevanttina yli 12 vuotta julkaisun jälkeen on harvinaista tietokoneaiheisille kirjoille [27]. Woolf selittää kirjan pitkän vaikutuksen johtuvan kirjan keskittymisestä suunnittelumalleihin eikä spesifisiin teknologioihin [27].

Teos sisältää 67 sanomapohjaista suunnittelumallia ja nämä mallit on jaettu seitsemään eri kategoriaan [16]:

1. Sanomajärjestelmät (*messaging systems*)
2. Sanomakanavat (*messaging channels*)
3. Sanomien rakentaminen (*message construction*)
4. Sanomien reititys (*message routing*)
5. Sanomien muokkaus (*message transformation*)
6. Sanomien päätepisteet (*messaging endpoints*)
7. Järjestelmän hallinta (*systems management*)

Tässä tutkielmassa perehdymme vain sanomajärjestelmiin, koska muissa kategorioissa esitetyt EIP:et sisältyvät sanomajärjestlemissä esiteltyihin malleihin.

3.1 Sanomajärjestelmät

Järjestelmät kategoria toimii kattokategoriana muille kategorioille ja tarjoaa konseptit ja termit muille suunnittelumalleille. Kategoria esittelee lähtökohtaiset pohjasuunnittelumallit minkä päälle muut suunnittelumallit rakentavat.

Sanomajärjestelmät (*messaging systems*) on kattokategorian nimi EIP:lle, mutta viittaa myös järjestelmään joka jakelee sanomia. Vaihtoehtoinen termi sanomajärjestelmille on sanomapohjainen väliohjelmisto (*message-oriented middleware*). Hohpe ja Woof käyttävät termiä vastaavasti teoksessaan [16]. Jos tekstissä viitataan sanomajärjestelmiin kattokategoriana se tarkennetaan erikseen, muussa tapauksessa sanomajärjestelmä viittaa edellä mainittuun ohjelmistojärjestelmään joka kuljettaa sanomia.

- Sanomakanava (*Message Channel*): Kanavia käytetään viestin organisointiin ja jaoteluun sanomajärjestelmän sisällä. Kanava toimii eräänlaisena virtuaalisena putkena lähettäjän ja vastaanottajan välillä. Kehittäjän vastuulle jää kanavien luonti ja organisointi. Sovelluksen lähettäessä dataa, dataa ei lähetetä satunnaisesti mille tahansa kanavalle vaan lähetettävä sovellus tietää juuri oikean kanavan jolle data on tarkoitettu. Vastaavasti sama dynamiikka kuuluu myös vastaanottavalle sovellukselle; sovellus tietää juuri oikean kanavan mistä odottaa oikeanlaatuista dataa. Kanavat toimivat loogisina osoitteina sanomajärjestelmälle. Kanavan toteutusyksityiskohdat vaihtelevat eri sanomajärjestelmien välillä [18] [16]. Kanavien käyttö on laskentakapasiteetin osalta halpaa, mutta ei täysin ilmaista; jokainen kanava tarvitsee muistia sanomien datan esittämiseen ja pysyvien sanomien tapauksessa – sanomat jotka säilyvät levyllä esimerkiksi virhetilanteessa – myös levytilaa. Kanavia kannattaa siis luoda useita, mutta lukumäärän kasvaessa tuhansiin, alkaa sanomajärjestelmän skalautuvuudesta tulla haaste [16].
- Sanoma (*Message*): Jos kanavaa voi ajatella putkena niin sanomaa voi ajatella putken vetenä, paitsi virran sijaan putkessa data kulkee yksittäisinä datayksiköinä. Kuten monen muunkin protokollan kohdalla, niin myös sanomakin sisältää otsakkeen (*header*) ja hyötykuorman (*payload*). Sanomajärjestelmät eivät ota kantaa hyötykuorman sisältöön, mutta vastaanottava sovellus voi reagoida eritavoilla hyötykuorman sisältöön. Saapuva sanoma voi aktivoida proseduurin, välittää dataa, ilmoittaa sovellusta tilamuutoksesta tai vaatia sovelluksesta vastausta [16]. Sanomat voi myös lähettää sarjana jos lähetettävä data ei mahdu yhteen sanoman hyötykuormaan. Jos datalla

on rajallinen voimassaoloaika, voi sanoma tarkoittaa lähetetyn datan voimassa olo ajan [16]. Hohpen ja Woofin teos sisältää suunnittelumallit näille erillisille sanoman käyttötavoile.

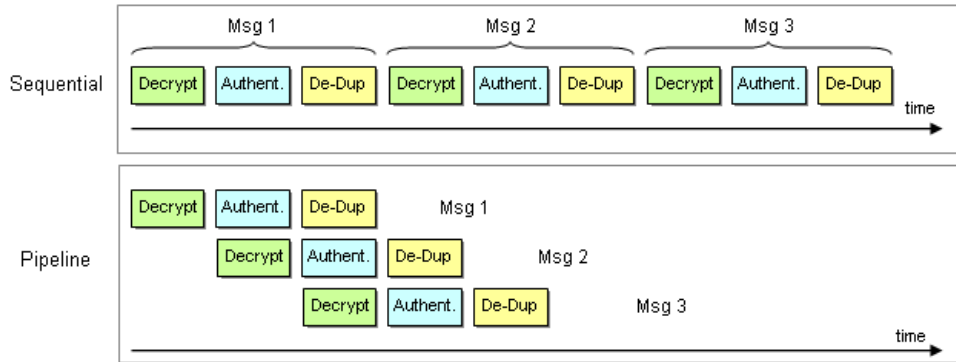
- Putket ja suodattimet (*Pipes and filters*): Putket ja suodattimet suunnittelumallissa putkena toimivat edellä mainitut kanavat ja suodattimet ovat pieniä itsenäisiä prosessointiaskeleita joilla ei ole kytköksiä muihin prosessointiaskeleihin. Kanavat yhdistävät nämä suodattimet toisiinsa ja suodattimet voivat toimia yksin tai olla yhdistettynä toisiinsa. Vaihtoehtoisesti putkena voi toimia muistissa oleva jono. Suodattimien rajapinta pitää olla tarpeeksi yksinkertainen, että putkien implementaatio on vaihdettavissa ja kiinteitä kytköksiä muihin suodattimiin tai putkiin ei synny. Asynkroninen viestintä komponenttien välillä mahdollistaa suodattimien samanlaisen käytön. Suodattimien löyhät kytkökset helpottavat testaamista kun suodatimia voi testata itsenäisinä komponentteina jonka lopputuloksen oikeudellisuutta voi helposti verrata siihen sisälle syötettyyn dataan.

Putket ja suodattimet lähestymistapana on myös saanut suosiota integraatioalustoiden arkkitehtuureissa; neljästäkymmenestäkahdeksasta modernista integraatioalustasta kymmenen hyödyntää putket ja suodattimet arkkitehtuurityyliä [7]. Putket ja suodattimet arkkitehtuurille löytyy myös implementaatioohjeita monelle alustalle ja teknologialle. Internet haulla putket ja suodattimet suunnittelumallille löytyy ohjeet seitsemän eri teknologian virallisilta sivuilta, Amazon Web Services, Red Hat Fuse, Spring Integration, Microsoft Azure, Mulesoft Runtime ja Apache Camel [4] [2] [1] [24] [6] [23].

Suodattamien käyttö ei ole täysin ilmaista. Suodattamien skaalautuvuuden haasteena on se, että jokainen suodatin tarvitsee oman putkensa datan kuljettamiseen. Jos putken toteutetaan sanomakanavana niin näiden käyttö eivät myöskään ole täysin ilmaisia kuten aikasemmassa suunnittelumallissa on mainittu; vastaavasti muistissa olevat jonot kuluttavat luonnollisesti muistiresursseja. Myös sanomien kulku putkissa ja suodattimista tuo omat haasteensa, koska viestit pitää käsitellä kanavan formaatista sovellukseen käyttämään formaattiin ja sama prosessi pitää tehdä takaperin sanoman lähtiessä, syöden operaatiosyklejä. Pitkät suodatinketjut tarjoavat arkkitehtuurista joustavuutta löyhien kytköstensä ansioista, mutta maksavat potentiaalisen suoritustehon laskemisen muodossa, johtuen toistuvista datan esitystavan muutoksista.

Asynkronisten sanomakanavien hyödyntäminen mahdollistaa jokaisen suodatinyksikön käsitellä sanoman itsenäisesti omassa säikeessään tai prosessissaan. Tämä johtaa huomattavasti suurenpaan suoritustehoon kun yksikään suodatinyksikkö ei ole odottamassa sanoman käsittelyn päättymistä vaan seuraavaa sanomaa voidaan al-

kaa käsittelemään enne kuin edellinen on valmistunut. Hohpe ja Woolf kutsuvat tätä prosessointiputkeksi (*processing pipeline*) jota selventää kuva 3.1 [16] [18] Kuvassa 3.1 sanoman salausta puretaan (*decrypt*), autentikoidaan (*authent.*) ja lopuksi tarkistetaan duplikaateilta (*de-dup*).

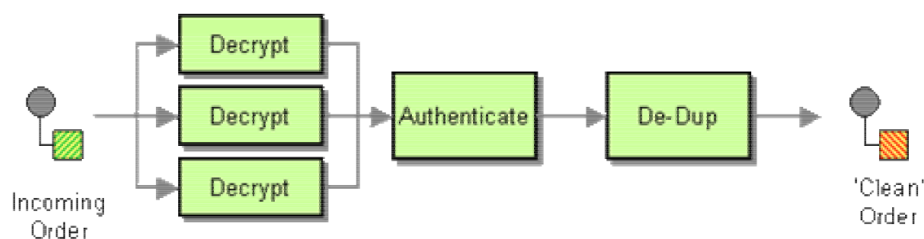


Kuva 3.1: Kuva sanomien peräkkäisen käsittelyn ja prosessointiputken eroista [16]

Peräkkäisessä käsittelyssä, eli ilman asynkronisuutta, sanomat joutuvat odottamaan toisten sanomien käsittelyä. Alemmassa prosessointiputkessa asynkronisuus mahdollistaa suodattimien operaatiot heti kun seuraava sanoma saapuu.

Suodatinketjun suorituskyykyä voidaan parantaa tästä entisestään. Edellisessä prosessointiputken esimerkissä heikkoutena on se, että kaikesta hitain suodatinprosessi määrää koko järjestelmän sanomien käsittelytahdin. Tässä tapauksessa hitain suodatin voidaan rinnakkaistaa. Haasteena on varmistaa, että jokaisen sanoman voi käsitellä vain yksi rinnakkaisista suodatinprosesseista. Toinen huomioon otettava aspekti on, että rinnakkaistaminen ei takaa sanomien suoritusjärjestyksen säilymistä. Jos sanomien käsittelyjärjestyksellä on väliä, suodatinprosesseja voi olla vain yksi instanssi. Kolmas haaste on suodattimien tilanhallinta. Tilanhallinta rinnakkaisissa operaatioissa lisää tunnettusti kompleksisuutta, niin suodattimen tilanpalautuminen lähtöpisteeseen operaation valmistuttua yksinkertaistaa huomattavasti suodattimien rinnakkaistamista.

Kuvan 3.2 esimerkissä Hohpe ja Woof näyttävät miten salauksen purkamisen (*decrypt*) käytetty suodatin voidaan rinnakkaistaa kolmeksi rinnakkaiseksi prosessiksi, koska purkamisoperaatio on työläin suodattimista [16]. Esimerkissä duplikaattien tarkastus (*de-dup*) on pidetty peräkkäisenä prosessina, koska duplikaattien havaitseminen tarvitsee sanomahistorian hyödyntämistä eli operaatio ei ole tilaton ja sen rinnakkaistaminen on haastavaa.



Kuva 3.2: Kuva sanomien rinnakkaisesta käsittelystä kun järjestyksellä on väliä [16]

- Sanomareititin (*Message Router*): Putkien ja suodattimien löyhät kytkennät mahdollistavat uuden prosessointiaskeleen lisäämisen olemassaolevien väliin. Sanomareitittimen tapauksessa reititin vastaa putkien ja suodattimien suodatinta siinä suhteessa, että se tekee suorittaa prosessointia, mutta suodattimien sijaan reititin on kytketty useampaan ulostulo kanavaan. Reititin ei myöskään muokkaa sanomaa vain on ainoastaan keskittynyt sanomien määränpään valitsemiseen. Reitittimen etuna on löyhä kytkentä muihin komponentteihin; ympärillä olevien suodattimien ei tarvitse olla tietoisia reitittimen olemassaolosta ja kaikki reitityspäätöksen logiikka elää yhdessä sijainnissa. Jos uusia sanomatyyppä tai reititysmuutoksia toteutetaan niin ainoastaan sanomareitittimen logiikka tarvitsee muokata. Koska saapuvan kanavan kaikki viestit siirtyvät reitittimen läpi yksitellen, on saapuvien viestien käsittelyjärjestys myös taattu. Löyhän kytkennän ylläpitohaasteet korostuvat sanomareitintä hyödyntäessä. Jos suurin osa järjestelmästä on löyhästi kytketty toiseen niin järjestelmän kokonaiskuvan muodostaminen käy hankalaksi. Hohpe ja Woofin mukaan näitä haasteita voi lieventää hyödyntämällä sanomien kulkuhistoriaa [16, sivu 93].

Reitittimen haasteina ovat erilaiset pullonkaulat. Reitittimestä voi syntyä ylläpiollinen pullonkaula, koska reitittimen pitää olla tietoinen kaikista vastaanottavista kanavista ja tästä voi syntyä ylläpiollinen ongelma varsinkin jos vastaanottokanavien lista vaihtuu tiheään. Toinen mahdollinen pullonkaula on suoritussy. Reititin luonteensa vuoksi lisää ylimääräisen prosessointiaskeleen ja monissa sanomajärjestelmissä sanoman siirtäminen toiselle kanavalle vaati sanomien uudelleen koodausta. Useamman reitittimen rinnakkainen käyttö lieventää ongelmaa, mutta jokataapauksessa reititin tulee vaikuttamaan negatiivisesti sanomien viiveeseen vaikka sanomien käsittelytahti pysyisikin samana [16, sivu 93].

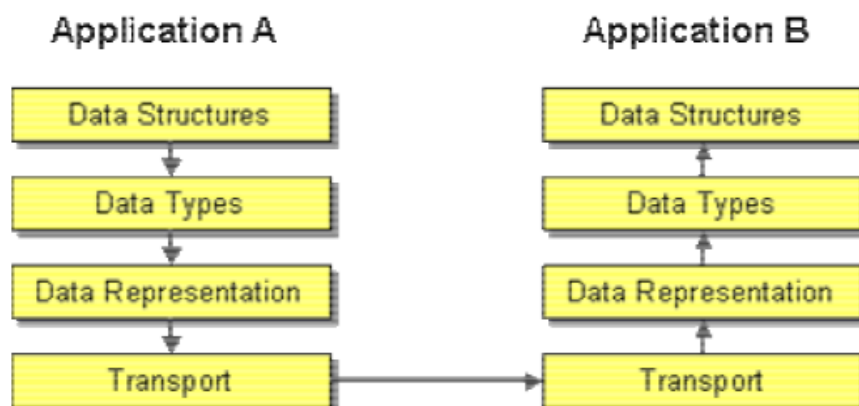
Reitittimen käyttö voidaan kategoroida useamaan eri tyyppiin. Yksinkertaisin rei-

titin on kiinteä jolloin sanoma kulkee yhdestä kanavasta yhteen vastaanottaja kanavaan. Kiinteiden reitittimen tarkoitus on irrottaa kiinteitä kytkentöjä erilaisista alijärjestelmistä tai välittää sanomia eri integraationratkaisujen välillä. Reititin voi olla sisältöpohjainen (*content-based*) tai ympäristöpohjainen (*context-based*). Sisältöpohjainen tekee reitityspäätöksiä sanoman sisällön perusteella. Ympäristöpohjainen ottaa huomioon ympäristön tilan ja tekee esimerkiksi kuormituksen tasaamista tai uudelleenreititystä ympäröivän systeemin virhetilanteessa. Reitittimet voi jakaa tilallisiin ja tilattomiin; tilalliset reitittimet ottavat huomioon aikasemmat sanomat ne voivat esimerkiksi estää duplikaattisanomien saapumisen tallentamalla jo saapuneet sanomat. Reititin voi olla myös dynaaminen *dynamic router* ja vastaanottaa konfiguraatiomuutoksia ohjausväylän kautta. Tällä tavalla reititin voi muuttaa reitityslogiikkaansa ilman koodimuutoksia [16, sivu 94].

- Sanomankääntäjä (*Message Translator*): Sanomankääntäjä on suodatintyyppi, joka mahdollistaa eri dataformaatteja käyttävät sovellukset keskustelemaan keskenään sanomien avulla. Hohpe ja Woof vertaavat sanomankääntäjä EIP:tä sanomajärjestelmille tarkoitetuksi versioksi tunnetusta sovitinsuunnittelumallista [16, sivu 97], joka sai alkunsa Gang of Four teoksesta [8]. Keratauksena aikaisempiin esiteltyihin EIP: hin verrattuna sanomakääntäjän tarkoituksena on vähentää tiukkoja riippuvuuksia arkkitehtuurissa; sanomakanava mahdollisti, että sovellusten ei tarvitse tietää toistensa sijaintia, sanomareititin mahdollisti löyhät kytkennät sovellusten viestien reitittämisessä, joten sanomakääntäjä estää järjestelmien tiukat riippuvuudet toistensa dataformaateista. Sanomakääntäjä ratkaisee jaetun tietokannan haasteita jota käsiteltiin luvussa 2.2. Erityisesti yhtenäisen skeemaan löytäminen ei muodostu ongelmaksi ja järjestelmät on kytketty löyhemmin toisiinsa kääntäjiä hyödyntämällä. Vaihtoehtoinen lähestymistapa sanomakääntäjän sijaan olisi käyttää yhteistä datatyyppiä koko sanomajärjestelmässä, joka jakaa myös yhtäläisyyksiä jaetun tietokannan kanssa 2.2, toteuttamalla sanoman datatyyppin muunnokset sanomien päätepisteisiin 3.1, mutta valmiissa kaupallisissa sanomajärjestelmiä tukevilla sovelluksissa päätepisteen koodi ei ole muokattavissa. Lisäksi sanomien kääntämislogiikan eläessä päätepisteessä, koodin uudelleenkäyttäminen on haastavaa.

Hohpe ja Woof [16] jakavat sanomakääntäjän operaatiot eri abstraktiotasoihin, ottaen löyhästi vaikutteita OSI-mallista [19], jakaen kääntämisoperaatiot kuljetus-, datan esitys-, datatyyppi- ja tietorakennekerrokseen (sovelluskerros). Kuljetuskerrokseen kuuluvat protokollat kuten http, mqtt, JMS. Datan esityskerros on kiin-

nostunut datan muodosta kuten JSON, XML ja enkoodauksesta kuten UTF-8. Datatyyppi viittaa soveluksen muistinsisäisiin esityistapoihin esim. ovatko postinumerot merkkijonoja vai kokonaisnumeroita ja ovatko esim. aikaleimat aina UTC-ajassa vai sisältävätkö aikavyöhykeinformaation. Tietorakennekerros (tai sovelluskerros) ottaa kantaa logiisiin kokonaisuuksiin, kuten eri dataentiteettien assosiaatioihin ja niiden kardianaliteetin esim. onko dataentiteettien välillä moni-moneen-yhteyksiä. Jakamalla kääntämisoperaatiot usempaan kerrokseen voidaan operaatioita käsitellä itsenäisesti omassa abstraktiotasossaan, kuten kuvassa 3.3, ilman riippuvuutta muihin kääntäjiin ja kääntämisoperaation koodin uudelleenkäyttäminen on suoraviivaista. Myös eri abstraktiotasojen operaatiot ovat vaihdettavissa toisiin. Esimerkiksi datan esityskerroksessa (*Data Representation*) voidaankin päättää, että data esitetäänkin JSON:ina CSV:een sijaan ilman, että vaihto aiheuttaa muutoksia muiden tasojen operaatioissa.



Kuva 3.3: Sanomakääntäjän operaatiot voivat kohdistua kaikkiin abstraktiotasoihin [16]

- Sanomien päätepiste (*Message Endpoint*): Verrattuna aikaisempiin esiteltyihin katokategorian EIP:in, sanoman päätepiste on puuttuva palanen, joka mahdollistaa integroitavan sovelluksen liittämisen osaksi sanomaviestintää hyödyntävää kokonaisuutta; sanomien päätepiste on siis asiakasohjelmisto, joka yhdistää sovelluksen sanomakanavaan ja mahdollistaa sanomien lähettämisen tai vastaanottamisen. Sanomajärjestelmät ovat palvelinohjelmia ja sanomien päätepiste toimii niiden asiakasohjelmienä. Sanomien päätepiste vastaanottaa sovelluksen komennon tai datan ja muuttaa sen sanomaksi ja lähettää sanoman sanomakanavalle ja päinvastoin sanomia

vastaanottavalle päätepisteelle. Hohpe ja Woof rajaavat yhden päätepisteinstanssin vain vastaanottamaan tai lähettämään, eikä toteuttamaan molempia eli sovellus hyödyntäisikin useampaa päätepidettä jos sen pitäisi kommunikoida useammalle sanomakanvalle [16, sivu 106]. Sovellus voi kuitenkin hyödyntää useampaa päätepidestanssia vaikka sanomat lähtisivätkin vai yhdelle kanavalle, mahdollistaakseen instanssien skaalatutumisen useammalle samanaikaiselle säikeelle.

Päätepidteen koodi on kustomoitu kyseiselle sovellukselle ja sanomajärjestelmän asiakasrajapinnalle sopivaksi. Päätepidte kapseloi sanomajärjestelmän sovellukselta joten sovellus ei ole tietoinen sanomajärjestelmästä. Esimerkiksi sanomajärjestelmän asiakasrajapinnan muutokset vaikuttavat vain päätepidteen koodiin, eikä lainkaan itse integroitavaan sovellukseen.

Päätepidte EIP:estä on muokattuja versioita riippuen siitä haluuako sanomia vastaanottava päätepidte käyttää erilaista vastaanottostrategiaa kuten kiertokyselyä tai tahtumapohjaista reagointia. Useammat vastaanottavat päätepidteet voivat kilpailla sanomien käsittelystä (*computing consumer*) tai jakaa viestin lähettämisen eri päätepidteelle sisäine logiikan mukaan (*message dispatcher*). Vastaanottava päätepidte voi vastaavasti olla idempotenssi niin toistuivat sanomat eivät aiheuta järjestelmässä virhetiloja [16, sivu 106].

4 Suunnittelumallien kehitystä

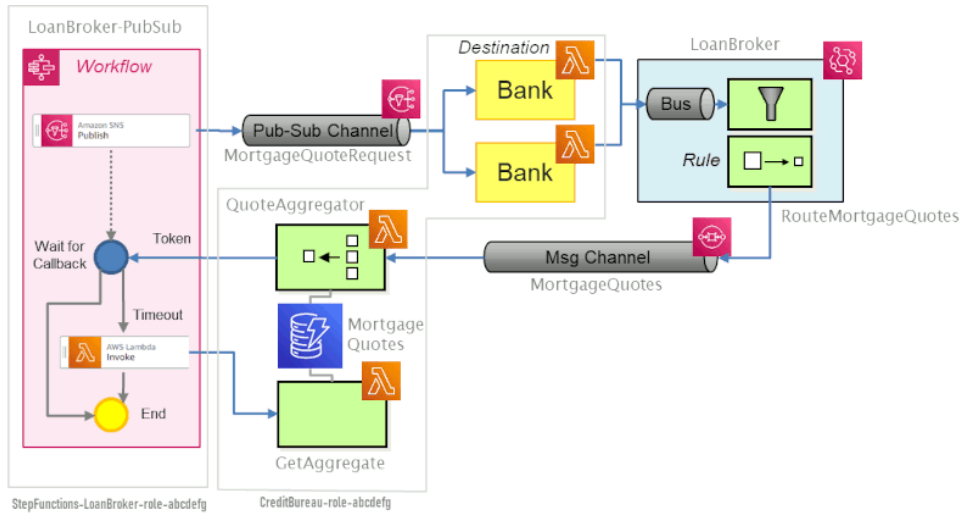
4.1 Kirjailijoiden omaa modernisointia

EIP-teoksen [16] julkaisun jälkeen kirjailijat (pääasiassa Gregor Hohpe) ovat pyrkineet pitämään sisällön relevanttina. Vaikka kirjan arkkitehtuurisisältö itsessään on pysynyt relevanttina ja vaikuttaa selkeästi nykypäivän järjestelmäintegraatoratkaisuissa ja asynkronisten sanomien käytössä, on kirjan koodiesimerkit ja teknologiamaininnat vanhentuneet; jota on myös tässä tutkielmassa pyritty korjaamaan tuoreemmilla esimerkeillä jotka ovat tämänpäivän opiskelijalle tuttuja.

Hohpen henkilökohtainen blogi sisältää useamman päivitetyn esimerkin tunnetuista EIP-suunnittelumallista, mutta nyt implementoituina käyttäen pilviteknologioita:

- Sisältöpohjainen reititin, EIP:een kategoriassa sanomien reititys, modernisoitiin käyttäen Googlen pilvialustan (*GCP*) Google Cloud Functions palvelitonta arkkitehtuuria vuonna 2017 [12]. Alkuperäisen teoksen *C# Microsoft Message Queuing* pohjainen toteutus vaihtui Node.js pohjaiseen, Google Cloud Pub/sub kirjastoa hyödyntävään toteutukseen.
- EIP-teoksessa esitelty monimutkaisempi esimerkki lainanvälittäjästä (*Loan Broker*) [16, sivu 317] toteutettuun kolmella eri tavalla: Java / Apache Axis web-palvelimena, C# / Microsoft Message Queuing sanomajonona ja Tibco ActiveEnterprise julkaise ja tilaa kanavana. Vuonna 2021 Hohpe aloitti sarjan blogikirjoituksia joissa lainanvälittäjä esimerkki toteutetaan uudelleen käyttäen Amazon Web Services (*AWS*) pilviteknologioita [9]. Lainanvälittäjä esimerkki toteutettiin pilvituotteilla AWS Step Functions, AWS Lambda, AWS Simple Notification Service, AWS Simple Queuing Service ja AWS DynamoDB. Katso kuva 4.1.

Myöhemmissä sarjan osissa Hohpe keskittyy ohjelmiston julkaisemiseen AWS:ää hyödyntäen [13] [14]. Hohpe myös argumentoi, että EIP-suunnittelumallit ovat itse pilviarkkitehtuuriratkaisun lisäksi hyödyllisiä abstraktioita koodin julkaisua automatisoidessa [14]. Julkaisussa käytettävät teknologiat ovat AWS CloudFormation, AWS Simple Storage Service, AWS Serverless Application Model ja AWS Cloud Development Kit.



Kuva 4.1: Kuva EIP:een lainanvälittäjä esimerkistä AWS:än palveluita hyödyntämällä. [13]

- Hohpen esimerkkien modernisointi jatkuu samana vuonna, lainanvälittäjä esimerkin siirtämisenä AWS:ltä Google Cloudille [15]. Ohjelman siirto ja kuvan 4.1 AWS palvelut kuvaantuivat yksi yhteen GCP:een palveluille. Esimerkissä hyödynnettiin GCP Workflows, GCP Cloud Functions, GCP PubSub, GCP DataStore teknologioita. Julkaisun automaatiossa huomattavaa oli, että AWS CloudFormationille ei löytynyt suoraa vastaavuutta GCP:een tarjonnasta ja vastaava toiminnallisuus pitää löytää kolmannen osapuolen ratkaisuksista [15].

EIP-teoksessa esiteltyjen suunnittelumallien esimerkkejä on myös vähitellen uudistettu EIP:een kotisivulle [11]. Tällä hetkellä 67 suunnittelumallista 14 on modernisoitu. Modernisoidut on koottu yhteen paikkaan Hohpen blogiin [10]. Aiemmin mainittujen pilvialustaesimerkkien lisäksi EIP-suunnittelumalleja on kyseisissä esimerkeissä toteutettua Golangilla, Apache Kafkalla, Apache Camelilla, Mule ESB:llä, RabbitMQ:lla ja Microsoft Azurella.

Edellä mainitut esimerkit, mukaan lukien listatut pilvialustaesimerkit, löytyvät julkisesta githubissa isännöidystä tietovarastosta [17].

4.2 Tilalliset protokollat

EIP:een toinen sudenkuoppa on tilalliset (*stateful*) protokollat. Hohpe myöntää, että tilallisesta suunnittelumalleista voisi tulla EIP toinen voluumi ja tilallisten suunnittelumallien

sisällyttäminen oikeuttaisi "Enterprise Integrations Patterns"otsikon käyttöä [27]. Hohpe on alkanut keräämään tilallisia suunnittelumalleja sivulle [11], mutta toistaiseksi suunnittelumallit eivät ole poikeneet uutta kirjaa. Verrattuna EIP:een Hohpen tilalliset mallit käyttävät sanomien sijaan keskusteluja (*conversations*) jonka ympärille suunnittelumallien abstraktiot on rakennettu.

Integraatiotrendien ja kaupallisten sekä avoimen lähdekoodin alustojen analyysissä todettiin, että lähes kaikki alustat tarjoavat omat tilalliset "keskustelunsa" ja mahdollisuudet tallennustilan käyttöön [26]. Tosin tämä alustojen tarjoama tuki todetaan alkeelliseksi. Palvelukeskeisen arkkitehtuurin (*Service Oriented Architecture, SOA*) kirjallisuudesta löytyy kuitenkin keskustelusuunnittelumalleja mitä ei löydy integraatioalustojen toteutuksista [26]; julkaisussa [5] suunnittelumallit yhdestä-moneksi (*one-from-many*) ja yhdestä-moneen (*one-to-many*) ovat monenvälisiä keskustelusuunnittelumalleja jotka todetaan puuttuvan olemassaolevista toteutuksista [26].

Koska nykypäivän integraatiot keskittyvät enemmän pilveen, integraatiovaatimukset ovat alkaneet kallistua tilallisiin tallennustilaa vaativiin skenaarioihin mihin EIP ei vastaa ja kerätyt keskustelusuunnittelumallit ovat myös keskenräisiä vastatakseen [26]. Paremmiin muodostetuille keskustelusuunnittelumalleilla löytyisi tarvetta.

Huomioitavaa on, että keskustelusuunnittelumalleja ei voi kuitenkaan verrata palvelukeskeisen arkkitehtuurin koreografia- (*choreography*) tai vuorovaikutussuunnittelumalleille (*interaction patterns*), koska keskustelumallit kuvaavat monimutkaisempia tehtäviä kuin datan lähettämisen ja vastaanottamisen [26].

4.3 Virtausprotokollat

Synkronisia- ja virtausprotokolleja (*streaming protocol*) ei ole huomioitu järjestelmäintegraatioissa eikä niiden suunnittelumalleissa, minkä hohpe ja woof myöntävät haastattelussa [27]. Erityisesti virtausprotokollien puute aiheuttaa haasteita esimerkiksi big data järjestelmien integraatioissa [26]. Samassa haastattelussa hohpe ja woof toteavatkin, että virtausprotokollien suunnittelumallien dokumentointi parantaisi sanomien käytön ja virtauskäyttötapausten yhtäläisyyksien ymmärtämistä ja johtaisi eip:een kaltaisten suunnittelumallien löytämiseen [26].

Virtausdatan käsittelemistä eip-malleilla on kokeiltu laitteistokiihdyttämisen kanssa hyödyntämällä ohjelmoitavia porttimatriiseja [25]. Virtausdatan käsittelyä varten eip-malleja

laajennettiin kahdella uudella suunnittelumallilla: kuormituksen tasaaja (*load balancer*) ja liittäjä reititin (*join router*). Kuormituksen tasaaja lähettää sanomaan yhteen useasta kanavasta hyödyntämällä yleisiä kuormituksen tasaus mekanismeja. Liittäjä reititin liittää useasta eri kanavasta saapuvat sanomat yhdelle kanavalle ohjelmoidun logiikan mukaisesti.

4.4 Virheenhallinta

Virheenhallinnasta: To handle erroneous situations during message processing, escalate them and make systems more fault-tolerant, error handling is seen as a major aspect [5], [45]. Hohpe et al. [3], [70] do only cover Dead Letter Channel as solution and sketch some ideas about the topic. Overall, in the literature, the topic is neither addressed from a pattern, formalization, nor modeling perspective. While [5] mentions missing patterns and formalization, Merkel et al. [45] lists Balancing and Distribution, as well as [69] mentions Fault-tolerance and Message Scheduling as missing aspects. Similarly, the insight into the current state of affairs, called monitoring, for services and cross-cloud are seen as important topics in [45], [61], [62][26]

4.5 Formalisointi

5 Yhteenveto

Lähteet

- [1] *3. Spring Integration Overview*. 2024. URL: <https://docs.spring.io/spring-integration/docs/4.2.5.RELEASE/reference/html/overview.html> (viitattu 02.02.2024).
- [2] *4.4. Pipes and Filters Red Hat JBoss Fuse 6.0 | Red Hat Customer Portal*. 2024. URL: https://access.redhat.com/documentation/en-us/red_hat_jboss_fuse/6.0/html/implementing_enterprise_integration_patterns/msgsys-pipes (viitattu 02.02.2024).
- [3] *Apache Parquet*. 2023. URL: <https://parquet.apache.org/> (viitattu 11.08.2023).
- [4] *Application integration patterns for microservices: Orchestration and coordination | AWS Compute Blog*. 2024. URL: <https://aws.amazon.com/blogs/compute/application-integration-patterns-for-microservices-orchestration-and-coordination/> (viitattu 02.02.2024).
- [5] A. Barros, M. Dumas ja A. H. T. Hofstede. "Service Interaction Patterns". *Lecture Notes in Computer Science* 3649 (2005), s. 302–318. ISSN: 1611-3349. DOI: [10.1007/11538394_20](https://doi.org/10.1007/11538394_20). URL: https://link.springer.com/chapter/10.1007/11538394_20.
- [6] *Enterprise Integration Patterns Using Mule | MuleSoft Documentation*. 2024. URL: <https://docs.mulesoft.com/mule-runtime/latest/understanding-enterprise-integration-patterns-using-mule> (viitattu 02.02.2024).
- [7] D. L. Freire, R. Z. Frantz, F. Roos-Frantz ja S. Sawicki. "Survey on the run-time systems of enterprise application integration platforms focusing on performance". *Software: Practice and Experience* 49 (3 maaliskuu 2019), s. 341–360. ISSN: 1097-024X. DOI: [10.1002/SPE.2670](https://doi.org/10.1002/SPE.2670). URL: <https://onlinelibrary.wiley.com/doi/full/10.1002/spe.2670%20https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.2670%20https://onlinelibrary.wiley.com/doi/10.1002/spe.2670>.
- [8] E. Gamma, R. Helm, R. Johnson ja J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1994, 139ff. ISBN: 0-201-63361-2. URL: <https://archive.org/details/designpatternsel00gamm/page/139>.

- [9] G. Hohpe. *Loan Broker Implementation with AWS Step Functions - Enterprise Integration Patterns*. 2021. URL: https://www.enterpriseintegrationpatterns.com/ramblings/loanbroker_stepfunctions.html (viitattu 03.05.2024).
- [10] G. Hohpe. *Modern Examples for Enterprise Integration Patterns - Enterprise Integration Patterns*. 2017. URL: https://www.enterpriseintegrationpatterns.com/ramblings/eip1_examples_updated.html (viitattu 03.05.2024).
- [11] G. Hohpe. *Overview - Enterprise Integration Patterns 2*. 2017. URL: <https://www.enterpriseintegrationpatterns.com/patterns/conversation/index.html> (viitattu 02.03.2024).
- [12] G. Hohpe. *Serverless Integration Patterns on Google Cloud Functions - Enterprise Integration Patterns*. 2017. URL: https://www.enterpriseintegrationpatterns.com/ramblings/google_cloud_functions.html (viitattu 03.05.2024).
- [13] G. Hohpe. *Serverless Loan Broker @ AWS, Part 4: Automation - Enterprise Integration Patterns*. 2021. URL: https://www.enterpriseintegrationpatterns.com/ramblings/loanbroker_automation.html (viitattu 03.05.2024).
- [14] G. Hohpe. *Serverless Loan Broker @ AWS, Part 5: Integration Patterns with CDK - Enterprise Integration Patterns*. 2022. URL: https://www.enterpriseintegrationpatterns.com/ramblings/loanbroker_cdk.html (viitattu 03.05.2024).
- [15] G. Hohpe. *Serverless Loan Broker @ GCP - Enterprise Integration Patterns*. 2022. URL: https://www.enterpriseintegrationpatterns.com/ramblings/loanbroker_gcp_workflows.html (viitattu 03.05.2024).
- [16] G. Hohpe ja B. Woolf. "Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions: Hohpe, Gregor, Woolf, Bobby: 9780321200686: Amazon.com: Books". *Addison-Wesley Professional; 1 edition* (2004), s. 736. URL: <https://books.google.com/cu/books?hl=es&lr=&id=bUlsAQAAQBAJ&oi=fnd&pg=PR7&dq=HOHPE,+G.+AND+WOOLF,+B.+Enterprise+Integration+Patterns:+Designing,+Building,+and+Deploying+Messaging+Solutions.+edited+by+I.+PEARSON+EDUCATION.+Edtion+ed.+Boston,+MA,+USA:+Addis.>
- [17] G. Hohpe ja B. Woolf. *GitHub - spac3lord/eip: Code for modern EIP examples*. 2023. URL: <https://github.com/spac3lord/eip> (viitattu 04.05.2024).
- [18] *Home - Enterprise Integration Patterns*. 2024. URL: <https://www.enterpriseintegrationpatterns.com/> (viitattu 26.01.2024).

- [19] *ISO/IEC 7498-1:1994 - Information technology — Open Systems Interconnection — Basic Reference Model: The Basic Model*. URL: <https://www.iso.org/standard/20269.html>.
- [20] P. Johannesson ja E. Perjons. ”Design principles for process modelling in enterprise application integration”. *Information Systems* 26 (3 toukokuu 2001). ISSN: 03064379. DOI: [10.1016/S0306-4379\(01\)00015-1](https://doi.org/10.1016/S0306-4379(01)00015-1).
- [21] D. S. Linthicum. *Enterprise application integration*. 2000, s. 21–23.
- [22] Mordorintelligence. *Enterprise Application Integration Market | Growth, Trends, Forecasts (2020 - 2025)*. 2020. URL: <https://www.mordorintelligence.com/industry-reports/enterprise-application-integration-market> (viitattu 11.05.2020).
- [23] *Pipeline :: Apache Camel*. 2024. URL: <https://camel.apache.org/components/4.0.x/eips/pipeline-eip.html> (viitattu 02.02.2024).
- [24] *Pipes and Filters pattern - Azure Architecture Center | Microsoft Learn*. 2024. URL: <https://learn.microsoft.com/en-us/azure/architecture/patterns/pipes-and-filters> (viitattu 02.02.2024).
- [25] D. Ritter, J. Dann, N. May ja S. Rinderle-Ma. ”Industry paper: Hardware accelerated application integration processing”. *DEBS 2017 - Proceedings of the 11th ACM International Conference on Distributed Event-Based Systems* 12 (kesäkuu 2017), s. 215–226. DOI: [10.1145/3093742.3093911](https://doi.org/10.1145/3093742.3093911). URL: <https://doi.org/http://dx.doi.org/10.1145/3093742.3093911>.
- [26] D. Ritter, N. May ja S. Rinderle-Ma. ”Patterns for emerging application integration scenarios: A survey”. *Information Systems* 67 (heinäkuu 2017), s. 36–57. ISSN: 0306-4379. DOI: [10.1016/J.IS.2017.03.003](https://doi.org/10.1016/J.IS.2017.03.003).
- [27] O. Zimmermann, C. Pautasso, G. Hohpe ja B. Woolf. ”A decade of enterprise integration patterns: A conversation with the authors”. *IEEE Software* 33 (1 2016), s. 13–19. ISSN: 07407459. DOI: [10.1109/MS.2016.11](https://doi.org/10.1109/MS.2016.11).

