Service design

TIE-23600 Palvelupohjaiset järjestelmät

Service-oriented architecture (SOA)

- Service-oriented architecture (SOA) is an architectural paradigm to create software based on the interaction of loosely coupled services
- A service is a piece of software that
  - is autonomous
  - performs a distinct business function
  - is defined by an implementation-independent interface, i.e. service contract

2

Tässä kertauksena SOA ja palvelu.

Eri lähteet esittävät erilaisia vaatimuksia SOA-järjestelmän osasille eli palveluille. Yleisimpiä ja tärkeimpiä ovat autonomisuus, löyhä sidonta, toteutusriippumaton rajapinta ja korkea abstraktiotaso. Tässä kalvosetissä tutustutaan tarkemmin näihin ja muihin palveluiden ominaisuuksiin.

Service design principles

- In his SOA book series, Thomas Erl defines eight principles of service design
  - Standardize service contract
  - Service loose coupling
  - Service abstraction
  - Service reusability
  - Service autonomity
  - Service statelessness
  - Service discoverability
  - Service composability
- Other sources have similar service design principles

Thomas Erl on kirjoittanut SOA:sta useita kirjoja, joissa muun muassa määritellään kahdeksan palveluiden suunnitteluperiaatetta.

Suunnitteluperiaatteita ei ehkä kannata noudattaa aina sataprosenttisesti, vaan hieman tilanteen mukaan. Ne ovat myös jonkin verran epämääräisiä; ei ole mitään tarkkaa mittaria esim. sille, milloin palvelu on löyhästi sidottu tahi uudelleenkäytettävä. Lisäksi ne ovat jossain määrin ristiriidassa toistensa kanssa; joudutaan tekemään kompromisseja sen suhteen kumpaa periaatetta painotetaan.

Joka tapauksessa ne ovat hyväksi havaittuja nyrkkisääntöjä suurten hajautettujen järjestelmien suunnittelussa. Käydään niitä tarkemmin läpi seuraavilla kalvoilla.

Standardize service contract

- "Services within the same service inventory are in compliance with the same contract design standards."
  - Service inventory = a collection of complementary services
- Harmonization of how service contracts are defined
  - How to express functionality
  - How to define data types
  - How to deal with policies
  - etc.
- Improves interoperability

Standardoitu palvelusopimus -suunnitteluperiaatteen tavoitteena on parantaa palveluiden yhteentoimivuutta

Tarkempi määritelmä palveluinventoriolle (service inventory):
"An independently standardized and governed collection of complementary services within a boundary that represents an enterprise or a meaningful segment of an enterprise."

Loose coupling

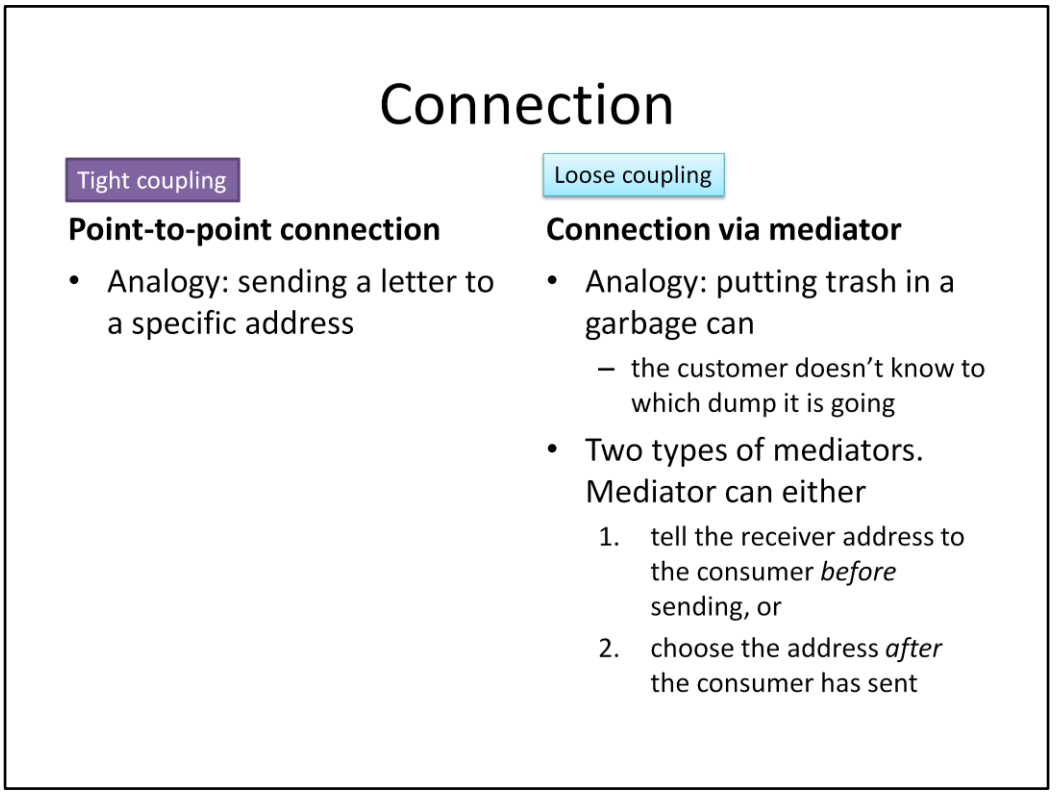
- “Service contracts impose low consumer coupling requirements and are themselves decoupled from their surrounding environment.”
- The aim of loose coupling is to minimize dependencies
- Loose coupling allows parts of the system to be developed independently of each other
- Loose coupling also has its drawbacks
  - Performance
  - (Complexity)

Some forms of loose coupling

	Tight coupling	Loose coupling
Connection	Point-to-point	Via mediator
Communication style	Synchronous	Asynchronous
Data model	Common complex types	Simple common types only
Interaction pattern	Navigate through complex object trees	Data-centric, self-contained messages
Control of process logic	Central control	Distributed control
Binding	Static	Dynamic
Platform	Strong platform dependencies	Platform independent
Transactionality	2PC (two-phase commit)	Compensation
Deployment	Simultaneous	At different times
Versioning	Explicity upgrades	Implicit upgrades

Nicolai M. Josuttis: SOA in Practice

Tässä joitain osa-alueita, joilla voidaan tehdä valintoja löyhän ja tiukan sidonnan välillä. Taulukko kirjasta Nicolai M. Josuttis: SOA in Practice.

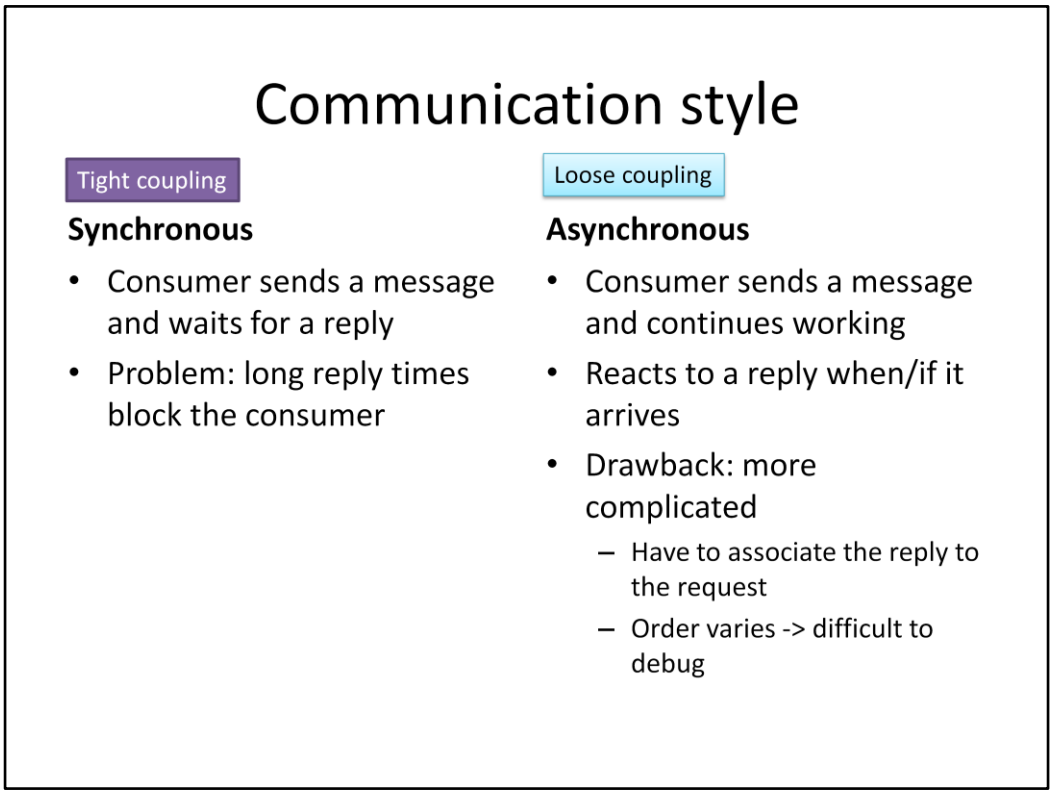


Erilaisilla yhteydenottotavoilla voidaan luoda eriasteista riippuvuutta lähettäjän ja vastaanottajan välillä.

Point-to-point –yhteydenottotavassa lähettäjä tietää etukäteen vastaanottajan osoitteen. Kuin kirjeen lähettäminen: lähettäjän täytyy kirjoittaa vastaanottajan osoite kuoreen.

Löyhempi sidos lähettäjän ja vastaanottajan välillä saadaan aikaan käyttämällä välittäjää (mediator).
Välittäjät voidaan jakaa kahteen perustyyppiin:

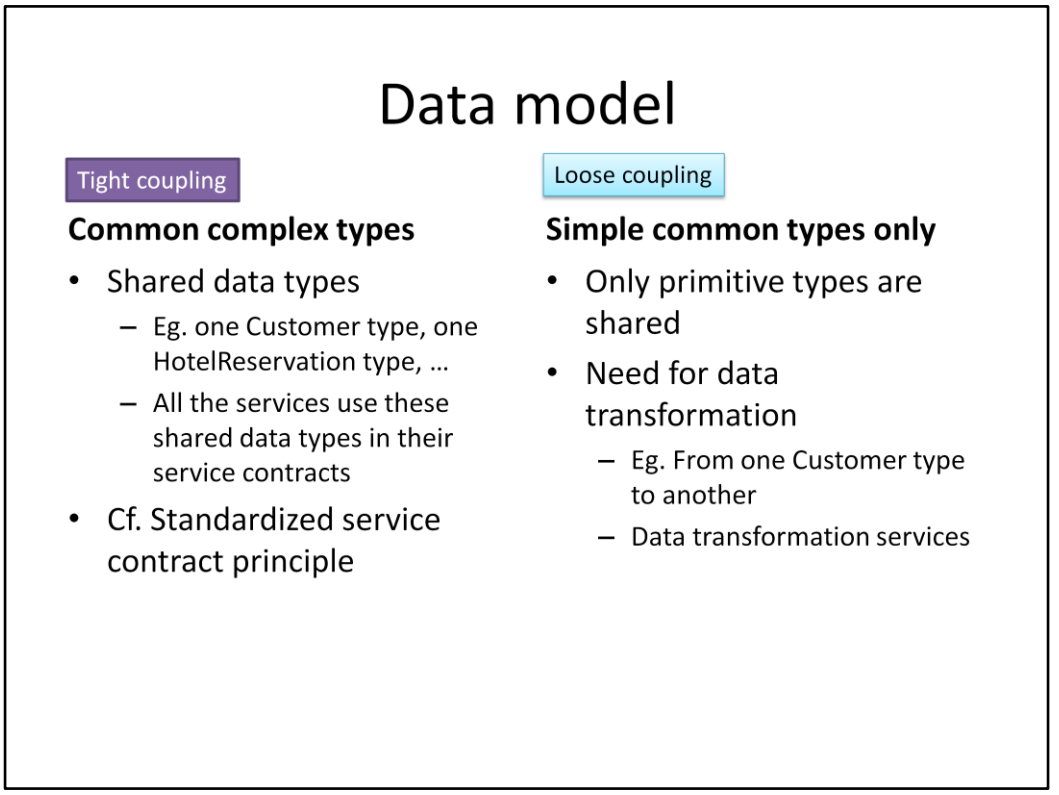
1) Välittäjä määrää osoitteen ennen lähetystä. Tässä tapauksessa lähettäjä kysyy välittäjältä vastaanottajan osoitetta. Välittäjä kertoo osoitteen, minkä jälkeen lähettäjä lähettää viestin. Tässäkin siis lähettäjä kyllä tietää vastaanottajan osoitteen, mutta saa sen selville vasta juuri ennen lähetystä. Esimerkiksi DNS – nimipalvelimet toimivat tähän tapaan.
2) Välittäjä määrää osoitteen lähetyksen jälkeen. Tässä asiakas antaa viestin välittäjälle liitettyään mukaan tarpeeksi tietoa, jotta viesti osataan ohjata perille.



Kommunikointi asiakkaan ja palvelun välillä voi olla synkronista tai asynkronista.

Synkroninen tapa on yksinkertaisempi. Huonona puolena siinä on, että asiakas joutuu odottamaan vastausta tekemättä mitään. Ei sovellu hyvin tilanteisiin, joissa vastauksen saaminen voi kestä kauan.

Asynkronisessa kommunikaatiossa asiakas jatkaa toimintaansa viestin lähettämisen jälkeen ja reagoi sitten sopivalla tavalla saapuneeseen vastaukseen. Huonona puolena tässä on lisääntynyt monimutkaisuus. Asiakkaan täytyy tietää, mikä vastaus liittyy mihinkin pyyntöön. Vastaukset saapuvat mielivaltaisessa järjestyksessä, joten debuggaus on vaikeaa.



Palveluilla täytyy olla jonkinlainen yhteinen käsitys käytettävistä tietotyypeistä, vähintään primitiivistä tyypeistä kuten string ja int. Se kuinka suurelta osin tietotyypit harmonisoidaan eri palveluiden välillä on suunnitteluratkaisu, jossa on tehtävä tradeoffeja.

Käyttävätkö esim. kaikki palvelut samanlaista Asiakas-tietotyyppiä, vai saavatko kaikki määritellä omansa. Jos Asiakas-tietotyyppi on määritelty keskitetysti ja siihen lisätään myöhemmin esim. puhelinnumero-kenttä, on muutoksia tehtävä kaikkiin palveluihin, jotka käyttävät Asiakas:ta rajapinnassaan.

Toisaalta jos Asiakas-tietotyyppiä ei ole harmonisoitu järjestelmän laajuisesti joudutaan tekemään muunnoksia tyypistä toiseen kun käytetään eri palveluita.

Type system

Tight coupling

Strong type checking

- Detect errors early
- Data type correctness can be checked by the SOA infrastructure before reaching the service

Loose coupling

Weak type checking

- More flexible
- ”Duck typing”
- Errors are detected later
  - Debugging more difficult

Transactions

Tight coupling

Two-phase commit (2PC)

- First, the coordinator puts all the participating services into a ”pre-commit” state
- If all participants agree, the actual commits are made in each service

Loose coupling

Compensation

- Commits are done immediately
- In the event of failure, a compensation is made
- Compensation = an action that undoes a specific action
  - Eg. Reserve hotel room – cancel the reservation

Esimerkki: matkatoimisto tilaa lentolipun ja hotellihuoneen eri palveluista. Molempien varausten pitää onnistua tai sitten kumpaakaan ei tehdä.

On ainakin kaksi tapaa toimia: 2PC ja kompensaatio (on muitakin).

2PC: luodaan hajautettu transaktio, jossa ”koordinoija” asettaa kaikki osallistuvat palvelut ensin tilaan, jossa ne ovat valmiita committiin. Jos kaikki palvelut onnistuivat tässä, suoritetaan varsinainen commit kussakin palvelussa.

Kompensaatio: commit tehdään heti. Jos joku palveluista epäonnistui, suoritetaan kaikissa palveluissa ns. kompensoiva operaatio. Esim. hotellihuoneen varausta kompensoiva operaatio on varauksen peruminen. Jokaiselle palvelun operaatiolle on siis oltava myös kompensoiva operaatio.

2PC: http://en.wikipedia.org/wiki/Two-phase_commit_protocol
Compensation: http://www.oracle.com/technetwork/articles/soa/ind-soa-7-service-comp-2005463.html

Binding

Tight coupling

Static (early) binding

- The service to be used is decided at design time

Loose coupling

Dynamic (late) binding

- The service to be used is decided at run time

Service autonomity

- "Services exercise a high level of control over their underlying runtime execution environment."
- Design-time autonomity
  - Services control their execution logic
- Run-time autonomity
  - Services control their execution environment

Service abstraction

- “Service contracts only contain essential information and information about services is limited to what is published in service contracts.”
- Services abstract unessential information about technology, logic and function
- Everything that is not defined in the service contract is hidden

Jotkin muut palveluiden suunnitteluperiaatteet (reusability, composability, …) vaativat, että palvelun rajapinnassa on saatavilla paljon informaatiota. Abstraktio-periaate rajoittaa tätä. Sen mukaan palvelun on julkaistava palvelusopimuksessa vain olennaiset tiedot, mitä se sitten minkin palvelun tapauksessa tarkoittaa. Ja kaikki mitä palvelusopimuksessa ei sanota, piilotetaan ulkopuolisilta.

Palvelusopimuksen abstrahoinnissa joudutaan tekemään kompromisseja.

Roskaesimerkki: asiakkaan kannalta olisi yksinkertaisinta jos jäteyhtiön ”rajapinta” olisi vain yksi laatikko, jonne kaikki roskat voi heittää. Yleensä jäteyhtiöt kuitenkin tarjoavat monimutkaisemman rajapinnan, jossa on erilaisia laatikoita erilaisia roskia varten. Tällöin jäteyhtiön sisäinen jätteidenkäsittelyn toteutus siis hieman ”vuotaa” ulos. Tässä ”epäabstraktimmassa” ratkaisussa on toki hyviä puolia (roskat saadaan näin paremmin kierrätettyä, jätteiden lajittelu yhdestä isosta laatikosta olisi jälkikäteen kallista, …)

Samantapaista tradeoffia voidaan joutua tekemään ohjelmistopalveluidenkin suunnittelussa. Esim. olisi asiakkaan kannalta helppoa ja muutenkin kaunista, että lentolippupalvelulla on yksi rajapintakutsu, jolla saa kaiken tiettyyn tilaukseen liittyvän tiedon. Voi kuitenkin olla esim. sellainen tilanne, että palvelu joutuu hakemaan istumapaikan tiedot jostain hitaasta, kalliista ja huonosta alipalvelusta. Lisäksi asiakkaat tarvitsevat istumapaikkatietoa vain harvoin. Tällöin voi olla järkevää tehdä erillinen operaatio istumapaikan hakua varten, jota asiakkaat kutsuvat vain silloin kun sitä tarvitsevat.

Kuvat varastettu osoitteista
http://www.jokimaki.com/main.site?action=siteupdate/view&id=16
http://www.hornborg.fi/verkkokauppa/vaihtolavat-ym-kuljetukset/roskalavapaketti-2-vrk

Service reusability

- “Services contain and express agnostic logic and can be positioned as reusable enterprise resources.”
- Agnostic functional context
  - Separate multi-purpose logic to separate services (vs. single-purpose logic)
- Generic service logic
- Generic and extensible service contract
- Support for concurrent access

”Agnostic functional context” tarkoittaa, että palvelu ei välitä (eli on agnostinen sen suhteen) missä kontekstissa sitä suoritetaan. Palvelun pitäisi siis olla sellainen, että sitä voi käyttää mahdollisimman monessa tilanteessa. Tätä varten pitäisi pystyä eriyttämään moneen käyttötarkoitukseen soveltuva (multi-purpose logic) logiikka omaksi palvelukseen, ei niin että se olisi osa jotain yhteen tiettyyn tarkoitukseen suunniteltua palvelua.

Palvelun pitäisi olla sopivan geneerinen. Tässäkin joudutaan tekemään tradeoffia sen suhteen, että kuinka helppoa palvelua on käyttää (service discoverability principle) vs kuinka geneerinen se on.

Service discoverability

- “Services are supplemented with communicative metadata by which they can be effectively discovered and interpreted.”
- Service contracts are accompanied with metadata
  - For humans
  - For computer programs

Service statelessness

- "Services minimize resource consumption by deferring the management of state information when necessary."
- Each call to service is independent
  - Service doesn't store state information between calls
  - State information = information that grows as the number of concurrent customer grows
- Improves scalability

Palvelun tilattomuudella tarkoitetaan sitä, että jokainen sille tuleva kutsu sisältää kaiken kutsun käsittelemiseen vaaditun tiedon. Palvelun ei siis tarvitse tietää mitä sama asiakas on tehnyt aiemmin.

Service composability

- "Services are effective composition participants, regardless of the size and complexity of the composition."
- Closely related to reusability
  - The "reusers" are often other services
- Controller services
  - May remain stateful while the composition members do work