Arkkitehtuurityylejä

Luento 5

Oppimistavoitteet

- Arkkitehtuurityylejä
  - Microservices, viestinvälitysarkkitehtuurit, vertaisverkkoarkkitehtuuri , map-reduce, (cloud), kerrosarkkitehtuurit

LISÄÄ ARKKITEHTUURITYYLEJÄ

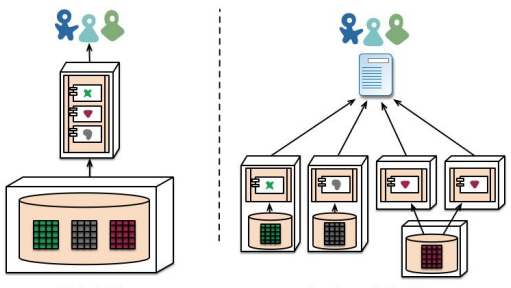
Microservices

- Nousussa oleva, vielä hieman täsmentymätön tyyli yritysjärjestelmien palvelinpuolen arkkitehtuuriksi
  - Vastaveto monoliittisiksi koetuille perinteisille kolmikerrosarkkitehtuureille (N-tier)
  - Tukee Continuous X –käytäntöjä (continuous integration, deployment jne.)
- ~ Unixin *pipes and filters* ajattelutapa sovellettuna palvelujen toteuttamiseen

Microservices

- Perusidea: palvelins*ovellus*
  - Koostuu kokoelmasta pieniä, *erillisiä palveluita*
  - Jokaista palvelua ajetaan omassa prosessissaan
  - Palvelut kommunikoivat kevyitä mekanismeja käyttäen (verrattuna ESB-väyliin), esimerkiksi HTTP resurssi-API:eja
- Palvelu
  - Osituksen perusteena liiketoimintaprosessit ja –toiminnot (yksi per palvelu, *single responsibility principle*)
  - Voidaan ottaa käyttöön (deployment) *itsenäisesti* ja *automaattisesti*
- Minimimäärä pavelujen keskitettyä hallinnointia
  - Palvelujen toteutus mahdollista eri ohjelmointikielillä ja kirjastoilla
  - Palvelut voivat käyttää omia tiedonhallintaratkaisujaan
  - Hyvä skaalautuvuus (+ ja -) tavoitteena
- Transaktioiden sijaan tarvitaan toisenlaisia menetelemiä tiedon eheyden takaamiseksi
  - Orkestrointi mikropalveluja *käyttävien* komponenttien vastuulla

Microservices



http://martinfowler.com/articles/microservices/images/decentralised-data.png

Microservices

- Martin Fowlerin sivut
  - http://martinfowler.com/microservices/
  - http://martinfowler.com/articles/microservices.html
- Katso esimerkiksi Fowlerin key-note esitys aiheesta GOTO Berlin 2014 –konferenssissa youtubesta
  - Linkki videoihin löytyy microservices – resurssisivulta (1. linkki yllä)

VIESTINVÄLITYSARKKITEHTUURIT

Viestinvälitysarkkitehtuurit

- Ongelma: miten toteutetaan asiakkaiden ja palvelujen välinen kommunikointi hajautetussa, heterogeenisessä palveluympäristössä?
  - Monia eri ohjelmointikieliä ja toteutusteknologioita käytössä
  - Halutaan välttää suoritusaikaisten komponenttien välisen suoran kommunikaation aiheuttamia suorituskyvyn pullonkauloja
  - Halutaan skaalautuvuutta, jossa voidaan dynaamisesti lisätä (ja vähentää) palvelevia komponentteja
  - Halutaan löyhät kytkennät komponenttien välille piilottamalla niiden toteutusteknologioiden yksityiskohdat (esim. ohjelmointikieli)
  - Halutaan joustavuutta ja ketteryyttä arkkitehtuuriin
  - Halutaan lisätä asynkronisuutta, jotta asiakkaan ei tarvitse jäädä odottamaan, kun palvelin käsittelee pyyntöä, vaan se voi jatkaa muita toimintojaan ja käsitellä palvelimen lähettämän vastauksen sen valmistuttua

Viestinvälitysarkkitehtuurit

- Ratkaisu: komponentit kommunikoivat lähettämällä toisilleen *viestejä*
  - Viesti *abstrahoi* ja *kapseloi* palvelupyynnön (ja vastauksen), siihen liittyvät attribuutit ja datan (payload) sekä protokollan
  - Viestin hyötykuorma voi olla iso tai pieni
  - Viestintä voi olla
    - synkronista (lähettäjä jää odottamaan) tai
    - asynkronista (lähettäjä jatkaa toimintaansa)
  - Viesti voidaan lähettää (1) tietylle kohteelle tai (2) yleislähetyksenä (broadcast) kaikille tunnetuille/kiinnostuneille kohteille

Message (Java Message Service 2.0)

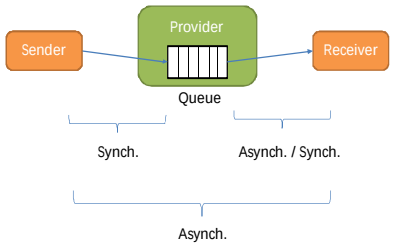
”Messages, as described here, are asynchronous requests, reports or events that are consumed by enterprise applications, not humans. They contain vital information needed to coordinate these systems. They contain precisely formatted data that describe specific business actions. Through the exchange of these messages each application tracks the progress of the enterprise.”

Viestinvälityksen peruskäsitteitä (Esimerkkinä JMS 2.0)

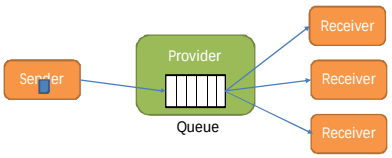
- Point-to-Point vs. Publish-and-Subscribe
  - Queue vs. Topic
  - ”Involved” vs. Fire-and-Forget

Point-to-Point

Jono (queue) on erillinen nimetty resurssi, johon lähettäjä ja vastaanottaja muodostavat yhteyden (eivät siis suoraan toisiinsa)
- Kummankaan ei tarvitse tuntea toista osapuolta (osoitetta tms.)

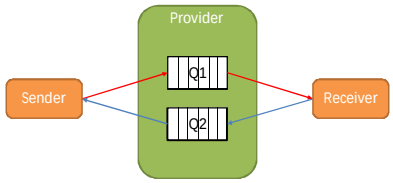


Point-to-Point
monta vaihtoehtoista vastaanottajaa



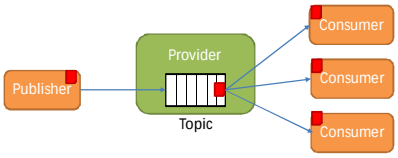
- Vain *yksi* vastaanottaja saa viestin (point-to-point)!
- JMS spesifikaatio ei määrää, miten saaja valitaan

Point-to-Point



Request-Reply – tarvitaan *kaksi* jonoa
("Involved" relationship)

Publish-and-Subscribe



Fire-and-Forget – julkaisijaa ei kiinnosta, ketkä viestin saavat

Viestinvälitysarkkitehtuurit

- Tyypillinen tilanne (roolit)
  - Julkaisija – Tilaaja (Producer/Publisher – Subscriber)
  - Tuottaja – Kuluttaja (Producer/Sender – Receiver/Consumer)
- Yksinkertaisimmillaan *julkaisija* pitää kirjaa vastaanottajista ja tietosisällön muuttuessa välittää tiedon tilaajille proseduurikutsun välityksellä.
- *Tilaaja* rekisteröityy vastaanottajaksi ja toimittaa rekisteröinnin yhteydessä vastaanottorajapinnan (call-back)
- Yllä oleva toimii ohjelman (prosessin) sisäisesti
- Verkkoratkaisussa tarvitaan *kommunikointiprotokolla* ja *välittäjä* (*broker, JMS 2.0 Provider*) huolehtimaan viestinvälityksestä

Viestinvälitysarkkitehtuurit
Välittäjä (Broker)

- *Välittäjä* (broker) tietää vastaanottajat (rekisteröityminen tai konfigurointi)
  - Julkaisija toimittaa viestin välittäjälle
  - Välittäjä toimittaa viestin edelleen siitä *kiinnostuneille* tilaajille (filtteröinti)
  - Tilaaja käsittelee viestin
- Välittäjään (broker) ja asynkroniseen viestintään perustuvaa arkkitehtuuria voidaan käyttää yleisesti palveluarkkitehtuurin runkona "perinteisemmän" etäproseduurikutsuihin perustuvan Asiakas-Palvelin arkkitehtuurin sijaan
  - Palveluekosysteemit

Viestinvälitysarkkitehtuurit Välittäjä

- Arkkitehtuurissa määriteltävä seuraavat asiat:
  - Keskenään kommunikoivat komponentit
  - Viestit, joiden avulla kommunikointi tapahtuu ilman, että lähettäjä tuntee vastaanottajan (tai vastaanottaja lähettäjän) fyysistä sijaintia (protokolla, syntaksi)
  - Operaatiot, joilla komponentit reagoivat viesteihin (komponenttien ymmärtämä kieli, semantiikka)
  - Säännöt, joiden avulla komponentit ja viestit rekisteröidään järjestelmälle
  - Palvelutasolupaus (viestien elinaika, taattu toimitus, transaktiot jne.)

Viestinvälitysarkkitehtuurit Välittäjä

- Säännöt, joiden perusteella välittäjä tietää, mille komponentille viesti on lähetettävä
  - Viestin vastaanottajan selvittäminen:
    - yleisviesti (multiple dispatch; viesti kaikille, jotkut käsittelevät, toiset eivät)
      - etu: vastaanottajan ei tarvitse etukäteen kertoa, mitä viestejä se haluaa vastaanottaa
    - komponentit kertovat rekisteröinnin yhteydessä, mitä viestejä (viestin tyyppi tai muu tunniste) ne haluavat vastaanottaa (tai keneltä)

Viestinvälitysarkkitehtuurit

- Rinnakkaisuus: missä määrin välittäjä ja komponentit toimivat rinnakkain
  - Viestinvälittäjillä ja vastaanottajilla viestijonot (välittäjän laatikko, vastaanottajan laatikko)
  - Voiko viestijono olla jaettu eri vastaanottajien kesken
  - Puskurointi, palvelutasot

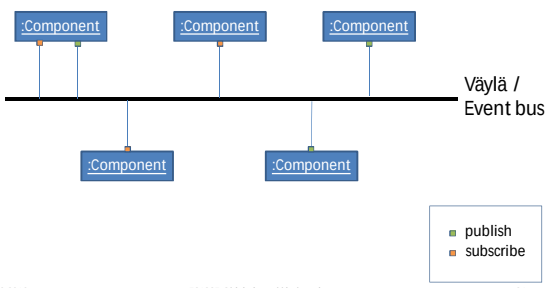
Esimerkkejä

- RabbitMQ on suosittu open-source viestinvälityspalvelu, joka tukee monia kieliä/ympäristöjä ja erilaisia viestinvälitys-protokollia http://www.rabbitmq.com
- JMS 2.0 – Java Message Service API https://jcp.org/en/jsr/detail?id=343

Tapahtumapohjaiset Viestinvälitysarkkitehtuurit

- *Tapahtumapohjaiset* arkkitehtuurit (*event-based*, *event driven*)
- Komponentit kommunikoivat aiheuttamalla tapahtumia (event)
- Tapahtumat voivat välittyä kaikille muille komponenteille, joista jotkut käsittelevät ja toiset eivät noteeraa
  - Tapahtumien lähetys voidaan myös rajoittaa julkaisija – tilaaja rakenteen tapaan – tapahtuma lähetetään vain tapahtuman tyypistä kiinnostuneille
  - komponentteja ei ole kuitenkaan lokeroitu joko julkaisijoiksi tai tilaajiksi vaan komponentti voi toimia kumpanakin

Tapahtumapohjainen viestintä

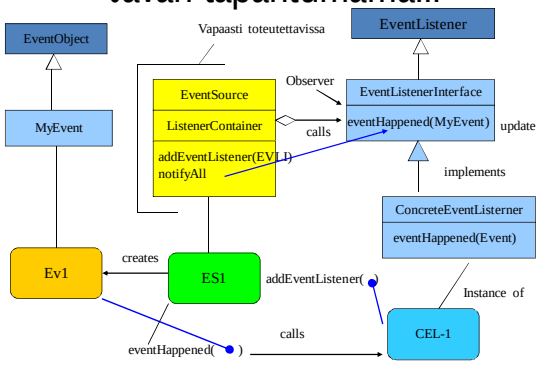


Tapahtumapohjaiset Viestinvälitysarkkitehtuurit

- Tapahtuma
  - Tilanne, joka voi sattua ohjelman suorituksen aikana
  - Edellyttää reagointia joiltain järjestelmän osilta (vrt. signaali)
  - Tapahtumaan liittyy usein pieni määrä dataa (ei välttämättä yhtään)
  - *Lähde* = tapahtuman synnyttävä komponentti
  - *Tarkkailija* = tapahtumaan reagoiva komponentti
- Lähde lähettää tapahtumailmoituksen sitä tarkkailemaan rekisteröityneelle tarkkailijalle
  - Esimerkiksi *Signals & Slots*[1] Qt-sovelluskehyksen olioarkkitehtuurissa
- Lähde tietää dynaamisesti tarkkailijoidensa olemassaolon, mutta ei niiden tarkkaa tyyppiä
  - Tapahtumaväylää käytettäessä lähteet eivät tiedä tarkkailijoistaan (vrt. MVC, Observer -mallit)
- Ilmoituksen lähetys voidaan hoitaa proseduurikutsuna tai sanomanvälityksenä (event bus, message bus)
  - proseduurikutsu synkroninen (esim Qt Signals & Slots)
  - Sanomajono asynkroninen/synkroninen

[1]http://en.wikipedia.org/wiki/Signals_and_slots,
http://qt-project.org/doc/qt-5.0/qtcore/signalsandslots.html

Javan tapahtumamalli



Viestinvälitysarkkitehtuurit

- Etuja
  - Joustavuus, muunneltavuus, modulaarisuus
    - Uusien tuottajien/julkaisijoiden ja kuluttajien/tilaajien dynaaminen lisääminen
    - Välittäjän/väylän tekemät automaattiset esitystapamuunnokset, erilaisia palvelutasoja toteutettavissa (varmistettu perillemeno vs. fire and forget)
    - Komponenttien omatahtinen evoluutio
  - Paljon käytettyjen välittäjä-/väyläratkaisujen hyväksi kehittynyt laatu
    - Luotettavuus, skaalautuvuus, suorituskykyoptimoinnit, jne.

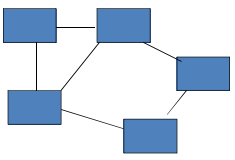
Viestinvälitysarkkitehtuurit

- Haittoja
  - Välittäjän tai tapahtumaväylän tuoma suoritusrasite (overhead) ja suorituskyvyn optimointimahdollisuuksien kaventuminen verrattuna suoriin (IPC-) yhteyksiin
  - Välittäjä tai väylä on *kriittinen komponentti* (single point of failure), jonka luotettavuus pitää saada paljon korkeammaksi kuin kommunikoivien komponenttien

MUITA HAJAUTETTUJA ARKKITEHTUUREJA

Vertaisverkkoarkkitehtuuri

- *peer-to-peer*
- Verkko, jonka solmut voivat toimia sekä *asiakkaina* että *palvelimina*
- Uusi solmu kytkeytyy verkkoon liittymällä suoraan johonkin tuntemaansa verkon solmuun
  - Solmut voivat dynaamisesti liittyä ja poistua



Puhtaassa vertaisverkossa kaikki solmut ovat tasavertaisia
Verkko on *symmetrinen* ja *ei-hierarkinen*

Vertaisverkkoarkkitehtuuri

- Palvelupyyntö etenee verkossa solmusta solmuun kunnes löytyy solmu, joka kykenee täyttämään pyynnön
  - Pyytäjän ja pyynnön täyttäjän välille voidaan muodostaa suora yhteys - tai sitten ei
- Rakenteettomassa verkossa pyynnön eteneminen sokeaa
  - Solmut tuntevat suorat naapurinsa, mutta eivät näiden tarjoamia palveluita
- Rakenteellisessa solmut jakavat rakennetietoa palveluista ja tätä käytetään ohjaamaan etenemistä
- Edellyttää verkkoprotokollaa

Vertaisverkkoarkkitehtuuri

- Etuja
  - Saatavuus paranee, jos palvelu/resurssi on hajautettu useina (osa-)kopioina verkon solmuihin (esimerkiksi jonkin mediatiedoston fragmentit)
  - Vikasietoisuus kasvaa, koska yksittäisen solmun vikaantuminen ei ole kriittistä (tiedon/palvelun toisteisuus, vaihtoehtoiset saantipolut)
  - Skaalautuvuus ja laajennettavuus ovat hyvät, ei yksittäistä kriittistä komponenttia (no single point of failure)

Vertaisverkkoarkkitehtuuri

- Haittoja
  - Verkkoon voi muodostua saaria tai klikkejä, jonka solmut eivät ole yhteydessä klikin ulkopuolisiin solmuihin
  - Yksittäiset haut saattavat aiheuttaa paljon turhaa liikennettä (resurssia etsitään turhaan, tai se löytyy monista solmuista)
  - Verkon rakenne ei ole vakaa (solmuja tulee ja lähtee)

Vertaisverkkoarkkitehtuuri

- Vertaissolmujen rinnalle onkin usein pakko luoda pysyviä *Super-* tai *Ultra-solmuja,* jotka
  - Liittävät uusia solmuja verkkoon
  - Kytkeytyvät suoraan moniin alempiin solmuihin ja toisiinsa hakujen optimoimiseksi
  - Katso esimerkiksi Gnutella, Skype

Map-Reduce -arkkitehtuuri

- Motivaatio
  - Hyvin suuren datamäärän hajautettu tallennus ja prosessointi
  - Data ja käsittelyoperaatiot suhteellisen yksinkertaisia, mutta niitä on todella paljon
- Perusperiaate
  - Koko datamassa jaetaan pienempiin samankaltaisiin osiin (*splits*)
  - Osat käsitellään rinnakkain suoritettavissa tehtävissä, jotka tuottavat paikallisen välituloksen (*map*-operaatio)
  - Välitulokset yhdistetään globaaliksi tulokseksi (*reduce*-operaatio)
- Katso hyvä esimerkki (YouTube – *Intro To MapReduce* by MapRAcademy)
  http://www.youtube.com/watch?v=HFpIUBeBhcM

Map-Reduce -arkkitehtuuri

- Etuja
  - Skaalautuvuus, saatavuus, suorituskyky, varmistukset
    - Rinnakkaisen käsittelyn massiivinen hyödyntäminen
  - Katso esim. Rackspace –esimerkki 1. luennolta
- Huomattavaa
  - Suoritusnopeus riippuu siitä, miten split-operaatio onnistuu jakamaan syötedatan ”yhtä vaativiin” ja keskenään samantyyppisiin, rinnakkain käsiteltäviin palasiin
  - Reduce-operaatioita voidaan ketjuttaa, mutta monimutkaisten operaatioiden koordinointi voi osoittautua hankalaksi

Cloud

- Erittäin perusteellinen kuvaus pilvessä tapahtuvasta tietojenkäsittelyn perusteista patternien muodossa
  - http://www.cloudcomputingpatterns.org/Cloud_Computing_Patterns
- Kirjan voi lukea SpringerLink –e-kirjastossa ja ladata sieltä pdf:nä (kuuluu yliopiston tilaukseen)
- Perusteet pilvilaskennasta olisi hyvä jokaisen ohjelmistoarkkitehdin tuntea

Kerrosarkkitehtuurit

- *Layered architectures*
- Kerrosarkkitehtuuri koostuu tasoista, jota on järjestetty jonkin abstrahointiperiaatteen mukaan nousevaan järjestykseen
  - Usein abstrahointi tapahtuu akselilla ihminen-laite
    - Sovellus – Palvelu - Resurssi
  - Usein kerrokset ovat luonteeltaan niin teknisiä, että eri abstraktiotasojen nimeäminen ja erottaminen käsitteellisesti toisistaan on hankalaa
    - Alemman tason palvelut ovat tyypillisesti *primitiivisempiä* ja *yleiskäyttöisempiä* kuin ylemmän tason palvelut, jotka ovat yleensä erikoistuneempia (tietty, rajatumpi tarkoitus)

Kerrosarkkitehtuurit

- Kerrokset ovat yleensä nähtävissä yhden suoritusympäristön (tietokoneen, solmun) sisällä suoritettavan ohjelmiston rakennetta ja sen käyttämiä palveluja tarkasteltaessa
  - Eri kerrokset *eivät* yleensä ole *hajautettuja* eri solmuihin suoritusympäristössä, toisin kuin kolmitasoarkkitehtuurissa
  - Kerroksisuus on nimenomaan ohjelmiston sisäinen organisointiperiaate

Kerrosarkkitehtuurit

- Virtuaalikoneet (virtual machines) kerrosarkkitehtuurityyppinä
- Perusajatus:
  - Taso toimii virtuaalikoneena, joka tarjoaa palveluita ylemmän tason korkeamman abstraktion virtuaalikoneelle. Ylemmän tason tarjoamat palvelut toteutetaan välittömästi alemman tason palveluita hyväksikäyttäen.
  - Ajatus toteutuu vain *puhtaassa (strict) kerrosarkkitehtuurissa*

Kerrosarkkitehtuurit

- Käytännössä puhtaasta kerrosarkkitehtuurista voi olla kahdenlaisia poikkeamia:
  - Palvelukutsuja tehdään alemmasta kerroksesta ylempään kerrokseen
  - Palvelukutsu ohittaa kerroksia kulkiessaan ylhäältä alaspäin (avoin kerrosarkkitehtuuri)
- Kerrosten ohittaminen on tarpeen useimmiten tehokkuussyistä (tai sitten välissä oleva kerros ei toteuta tiettyä palvelua, joka löytyy alemmalta kerrokselta)

Kerrosarkkitehtuurit

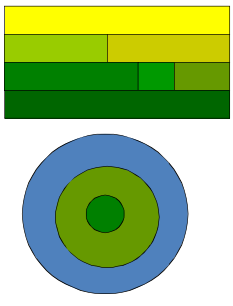
- Kerroksen ohittamisesta aiheutuva ongelma:
  - Tietty kerros tulee riippuvaiseksi myös muista kuin suoraan sen alapuolella olevasta kerroksesta
    - à Kerroksen vaihtaminen hankalaa
- Palvelukutsun tekeminen alemmasta kerroksesta ylempään päin voi olla merkki vakavasta ongelmasta arkkitehtuurissa
  - Alempi kerros saattaa olla riippuvainen ylemmästä
  - Syklinen riippuvuus

Kerrosarkkitehtuurit

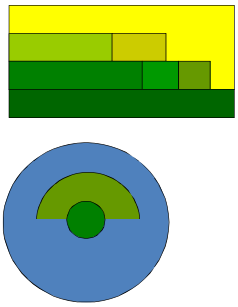
- Joskus alemman kerroksen on kuitenkin tarpeen kutsua ylemmän kerroksen koodia
  - Alemman kerroksen täytyy mukauttaa toimintaansa ylemmän kerroksen mukaan
  - Jotta alempi kerros ei tulisi (liian) riippuvaiseksi ylemmästä kerroksesta, voidaan hyödyntä *takaisinkutsuperiaatetta* (callback)
    - Alempi kerros tarjoaa rekisteröintioperaation, jonka avulla ylempi kerros rekisteröi takaisinkutsussa kutsuttavan koodin (funktion) alemman kerroksen käyttöön
    - Alempaa kerrosta ei kiinnosta, mitä ylemmän kerroksen takaisinkutsutfunktio tekee, se vain kutsuu sitä tietyssä tilanteessa

Kerrosarkkitehtuurit

Ei ohiteta kerroksia:



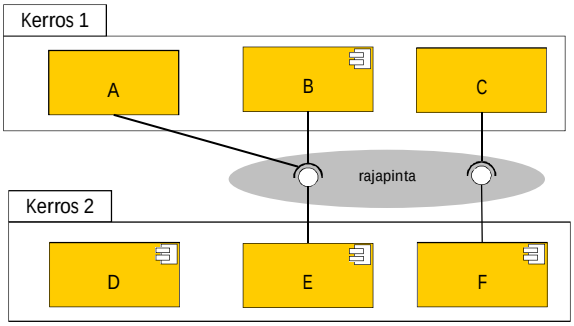
Ohitetaan kerroksia:



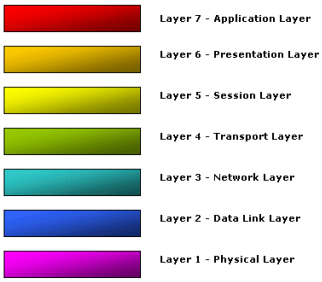
Kerrosarkkitehtuurit

- Kun kerrosten väliset rajapinnan on selkeästi määritelty, kerros voidaan vaihtaa toiseksi ilman, että se vaikuttaa muuhun järjestelmään
- rajapintojen/ kerrosten toimintojen *standardointi* lisää vaihdettavuutta

Kerrosarkkitehtuurit



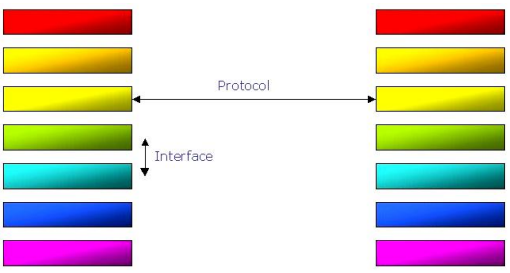
Kerrosarkkitehtuurit – OSI[1]



Eräänlainen *referenssiarkkitehtuuri* kommunikaatiojärjestelmille

[1]http://en.wikipedia.org/wiki/Osi_model

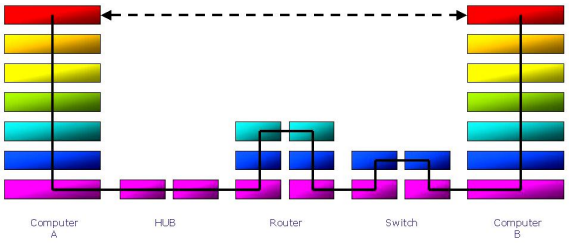
Kerrosarkkitehtuurit - OSI



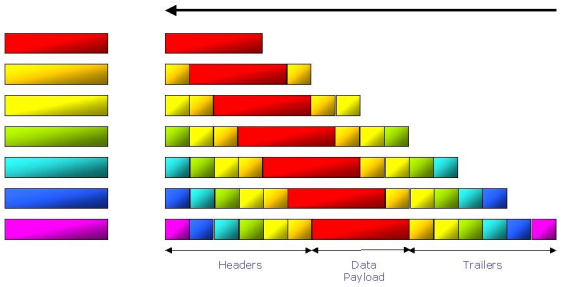
Vastinkerrokset kommunikoivat samalla abstraktiotasolla (yhteisellä protokollalla)

Kerrosarkkitehtuurit - OSI

Kokonaiskuva



Kerrosarkkitehtuurit - OSI



Jokainen kerros lisää oman palansa viestiin sen kulkiessa kerrosten läpi

Kerrosarkkitehtuurit – muita esimerkkejä

- Virtuaalikoneella rajapinta tai kieli (esim. Java bytecode), jonka avulla konetta käytetään
  - kätkee toteutuksen, mahdollistaa useita alustoja
- Sovellusrajapinnat palveluihin (API, application programming interface)
  - Esimerkiksi käyttöjärjestelmillä, luokkakirjastoilla à kätkee yksityiskohdat, parantaa siirrettävyyttä

Kerrosarkkitehtuurit

- Etuja:
  - Kerrosarkkitehtuuri on yleinen malli, jota voidaan soveltaa lähes kaikissa järjestelmissä pienessä tai isossa mittakaavassa
  - Jakaa järjestelmän korkealla tasolla osiin, joiden muodostamana kokonaisuutena järjestelmä on helppo ymmärtää
  - Intuitiivinen, helposti ymmärrettävä à voidaan käyttää kommunikointiin hyvin erilaisten sidosryhmien välillä
  - Ohjaa ohjelmiston riippuvuuksien minimointiin à muutosten paikallisuus à helppo ylläpidettävyys, muutettavuus, vaihdettavuus, *standardoitavuus*
  - Kerrosarkkitehtuuri tukee ohjelmiston uudelleenkäyttöä
    - Kukin kerros toteuttaa tietyn abstraktion à pohja työn jakamiselle à tasokohtaista erikoistumista

Kerrosarkkitehtuurit

- Ongelmia:
  - Tehokkuushäviö
    - Palvelukutsu ei välity suoraan palvelun toteuttavalle kerrokselle, vaan kulkee välissä olevien kerrosten kautta
    - Palvelukutsun parametrit mahdollisesti muutetaan joka kerroksen välillä uuteen esitysmuotoon
  - Toteutustaakka: periaatteessa jokainen palvelu on toteutettava joka tasolla (vaikkakin osa toteutuksesta delegoidaan alemmalle kerrokselle)
  - Oikean (toimivan) tasojaon keksiminen on hankalaa
    - Mihin kerrokseen tietty palvelu pitäisi liittää?

Kerrosarkkitehtuurit

- Ongelmia:
  - *Poikkeusten käsittely*
  - Kutsu ylemmältä kerrokselta à
    - poikkeus kerrosrakenteen pohjalla à
    - palataan kutsupinossa taaksepäin kunnes löytyy käsittelijä poikkeukselle à
    - poikkeuksen käsittely ylemmällä tasolla kuin missä poikkeus tapahtui à
    - käsittelijä ei välttämättä pysty korjaamaan tilannetta
  - Äärityspaus: poikkeus palaa käyttäjälle saakka antaen mystisen virheilmoituksen
  - Miksei sitten poikkeuksia käsitellä aina siellä missä ne syntyvät?