

Joensuun yliopisto
Tietojenkäsittelytiede

Hajautetut ja samanaikaiset järjestelmät

Syksy 2009

Simo.Juvaste@joensuu.fi

Istumajärjestys ensimmäisillä luennoilla:

- 1) Istu jonkun vieressä/edessä/takana
- 2) Koko luokan on oltava yhtenäinen

Luentomoniste: 5 euroa

Kurssin tiedot

Kts. <http://cs.joensuu.fi/pages/sjuva/hs.html>

Kurssin sisältö

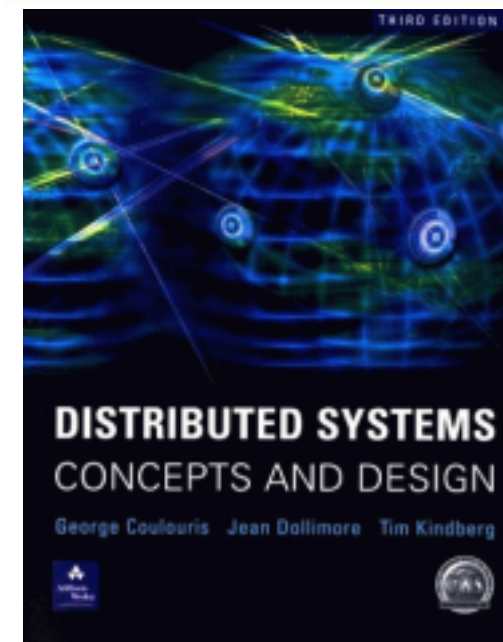
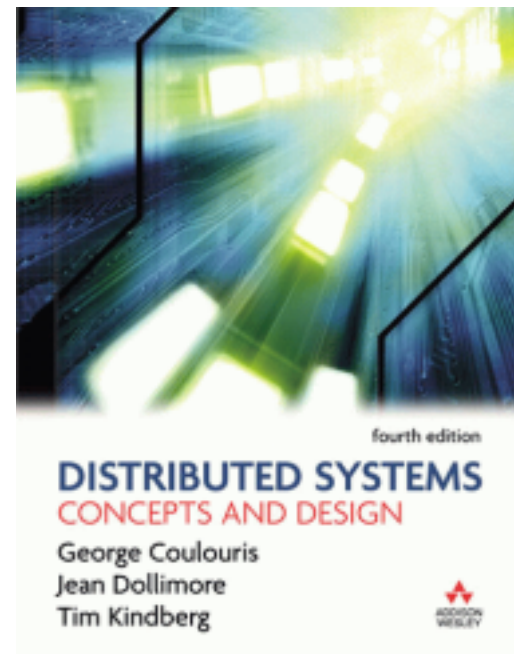
- Luku 1: Johdatus aiheeseen (sivu 8)
 - Mitä?, Miksi?, Miten?
 - Esimerkkejä
 - Hyödyt, haasteet, suunnittelun **lähtökohdat**
- Luku 2: Samanaikaisohjelmointi (sivu 64)
 - Säikeet, prosessit, lukot, jne.
- Luku 3: Hajautettujen järjestelmien mallit (sivu 101)
 - Arkkitehtuurit, vuorovaikutus, vikaantumisen, turvallisuus; perus"algoritmit"
- Luku 4: Kommunikaatio hajautetuissa järjestelmissä (sivu 189)
 - IP, TCP, UDP, socket, http, JavaRMI, jne, protokollasuunnittelua,
 - Verkonhallinnan alkeita.
- Luku 5: Rinnakkaislaskenta (sivu 269)

- **Hajautetut järjestelmät / Distributed systems**
 - Vanha laudatur-kurssi, kattaa pääosan tästä (ja enemmän)
- **Rinnakkaislaskenta / Parallel computing**
 - Tällä kurssilla johdanto.
- **Tietoliikenne(tekniikka)**
 - Hajautetut järjestelmät ovat kommunikaatiota, eli tietoliikennettä
 - Pitäydymme pääosin korkeammalla tasolla
- **Käyttöjärjestelmät, tietokonejärjestelmät**
 - Tarvitsemme/opimme käyttämään prosesseja/säikeitä, soketteja, jne.
- **Tietokannanhallintajärjestelmät/tiedonhallinta**
 - Sivuamme tällä kurssilla tietokannan etäkäyttöä
- **Järjestelmäsuunnittelu/-kehitys**
 - Ei juuri suunnittelumenetelmiä tällä kurssilla, tämä on käytännönläheisempi.

- Ohjelmointi/C/C++/Java/...
 - Perusohjelmointitaito oletetaan, opimme lisää säikeistä, prosesseista ja prosessienvälisestä kommunikaatiosta.
- WWW-ohjelmointi (käyttäen php:tä tms)
 - Ennemminkin näytetään **miten http-palvelin toimii** kuin miten sitä käytetään...
- Asema uudessa **UEF-kandissa**:
 - **Ohjelmointi 2** sisältänee samanaikaisuuden ja ehkä jotain socket-ohjelmoinnista.
 - Suunnitellut valinnaiset kurssit **Tietoliikennetekniikka** ja **Verkko-ohjelmointi** sivuavat tätä kurssia.
 - Syventäviin on suunniteltu **Hajautetut järjestelmät ja tapahtumankäsittely** tms. kurssia joka jatkaa tätä kurssia.

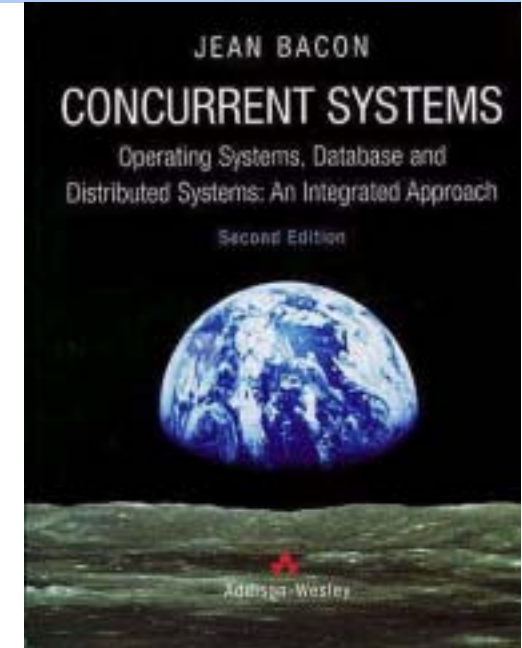
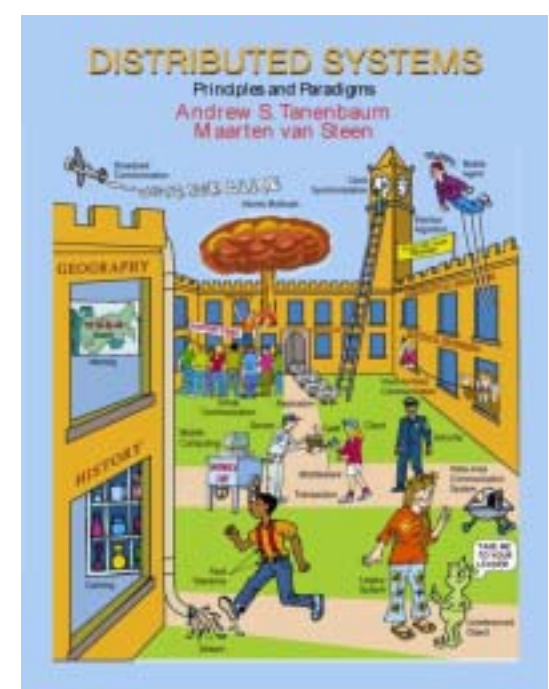
Kirjallisuus

- Luentomateriaali, muuta lähdemateriaalia.
 - Monistuskulut 5e.
- Coulouris, Dollimore, Kindberg: Distributed Systems, Concepts & Design, 3/4th Ed.
 - <http://www.cdk3.net/>
 - <http://www.cdk4.net/>
 - Pääosa kuvista on tästä kirjasta.



- Tanenbaum, van Steen: Distributed Systems, Principles and Paradigms.
<http://www.prenhall.com/tanenbaum>

- Bacon: Concurrent Systems
- java.sun.com (Java tutorial)
- Muuta, kts. viiteluettelo sivu 318
- WWW-linkkejä kurssin www-sivulla



Luku 1

Johdatus aiheeseen

Mitä?, Miksi?, Miten?

Siis: hajautettu/samanaikainen/rinnakkainen järjestelmä!

Edut, haitat

Esimerkkejä

Ihmiskoe

Haasteita, ratkaisutapoja

Mikä on *hajautettu järjestelmä* (Distributed System)

⇒ Hajautettu järjestelmä on kokoelma yhteen kytkettyjä **itsenäisiä tietokoneita** joka käyttäjälle näyttää **yhtenäiseltä järjestelmältä**.

- Koneet ovat itsenäisiä, ne voivat toimia (toimivat myös) itsenäisesti.
 - Erilliset tietokoneet **toimivat yhtä aikaa toisistaan riippumatta**.
 - Niillä ei ole yhteistä tarkkaa kelloa, ne voivat myös käynnistyä, toimia, kommunikoida, vikaantua ja palautua itsenäisesti.
- Käyttäjän näkemys: käyttäjä näkee yhden järjestelmän suorittavan hänen **antamaansa tehtävää** vaikka todellisuudessa tietokoneita olisi useita ja ne olisivat eri paikoissa.

⇒ Ylläolevan määritelmän mukaan Internet ei sinänsä ole hajautettu järjestelmä, vaan **alusta** (infrastruktuuri) erilaisille hajautetuille järjestelmille ja palveluille.

- kts. Lämmittelyveli sivulla 33.

Pelkällä hajautusta tukevalla järjestelmäohjelmistolla tietokoneet voivat:

- Tehdä yhteistyötä, jakaa resursseja (laitteita, ohjelmistoja).
- Esim. AD, NFS, Samba
- Ei oikein sovi hajautetun järjestelmän määritelmään, rajatapaus.

Miksi (teemme) hajautettuja järjestelmiä?

Joudumme:

- Yhdistämään (maantieteellisiä) etäisyyksiä.
- Yhdistämään organisaatioita.
 - Mahdollistaa (automaattista) yhteistyötä. *
- Yhdistämään eri alustoja (laite ja käyttöjärjestelmä).

Saamme myös hyötyjä:

- Parannamme suorituskykyä. *
- Vähittäinen järjestelmän kasvattaminen tarpeen mukaan.
- Parannamme saavutettavuutta.
- Parannamme vikasietoisuutta *
- Resurssien jakaminen
 - alhaisemmat kustannukset
 - tiedon yhtenäisyys

* = ehkä haastavimmat tavoitteet

Haasteet (erot paikallisiin/keskitettyihin järjestelmiin)

⇒ Kommunikaatio!

- **Monimuotoisuus** (heterogeneity)
 - Kaiken monimuotoisuus.
- **Verkkoviive** (latency)
 - Kaikki kommunikaatio kestää pidempään.
- **Yhteisen muistin (osoitteiston) puute**
 - Etämuistia ei voi käyttää kuten paikallista muistia.
 - Paikalliset viittaukset (osoittimet) eivät päde muualla.
- **Synkronisaatio**
 - Hajautuksesta seuraa samanaikaisuus (ja epädeterministisyys) ja samanaikaisten tapahtumien yhteensovittaminen.
- **Osittainen vikaantuminen**
 - Sovellusten olisi **siististi sopeuduttava** järjestelmän (minkä tahansa) **muiden osien** vikaantumiseen.

- **Turvallisuus**
 - "Vihamieliset" tietoverkot, enemmän uusia & vierailevia käyttäjiä.
- **Tuntumattomuus** (transparency)
 - **Kuinka piilottaa hajautus** käyttäjältä (ja ohjelmoijalta)?
 - Tavoite on saada hajautettu järjestelmä näyttämään yhdeltä (virtuaaliselta, keskitetyltä) järjestelmältä.
 - Voisi olla (lähes) yhtä helppo käyttää, suunnitella ja toteuttaa.
- kts. Lämmittelypeli sivulla 33.

Samanaikaisuus, rinnakkaisuus ja muut sukulaistermit

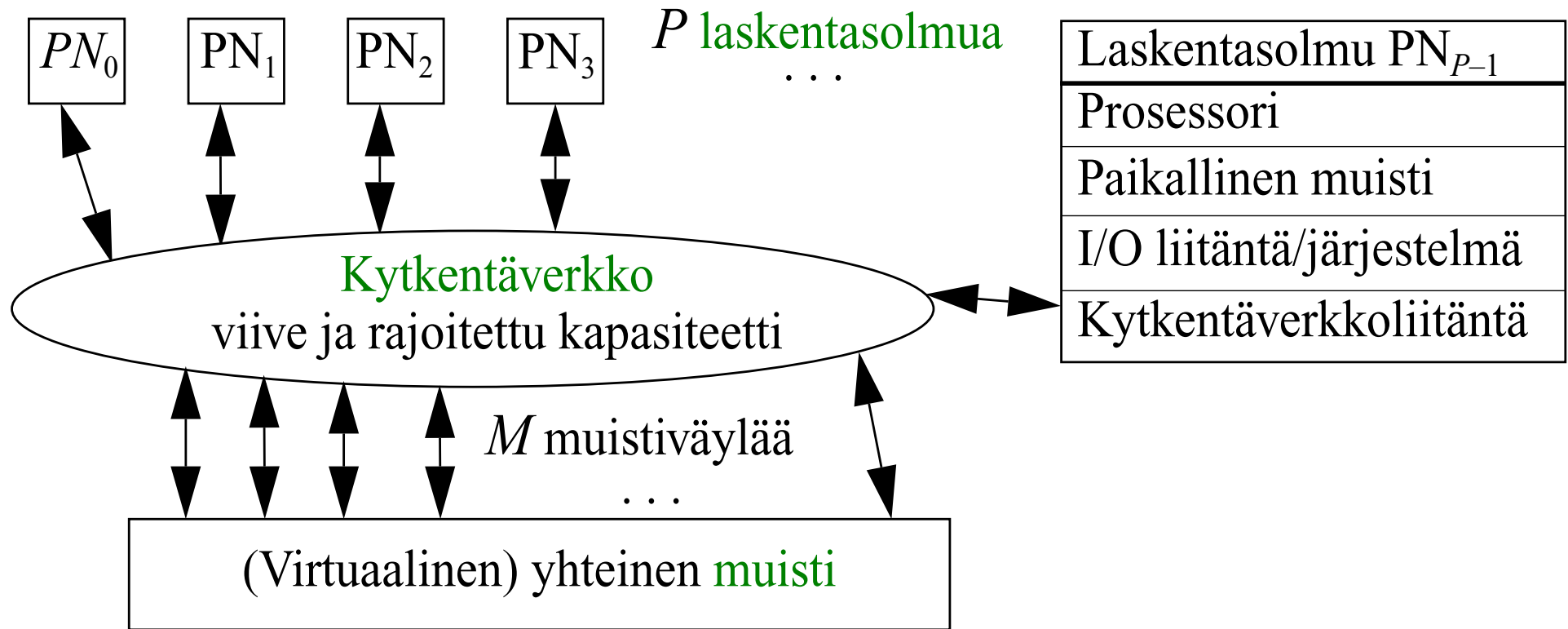
Rinnakkaislaskenta, -tietokone (parallel computation/computer)

- Käytetään useita prosessoreja/tietokoneita ratkaisemaan yhtä laskennallista tehtävää.
 - (Ainoa) tavoite on saada vaativa laskenta nopeammaksi.

⇒ Jopa P kertaa nopeammin P prosessorilla.

- Hyödyllistä (vain) jos meillä on kiire (simulaatio, ennustus, tosiaika-sovellus) (tai meillä on joutavia prosessoreita ja käyttäjä odottamassa).

- Rinnakkaistietokoneessa on yleensä kymmeniä..tuhansia samanlaisia prosessoreja **kytkettynä nopealla kommunikaatioverkolla** ja (yleensä) laitteistolla toteutettu virtuaalinen yhteinen muisti.



- Kytkentäverkko ja muisti(väylä) vievät helposti **puolet budjetista** jos halutaan tehokas yleiskäyttöinen tietokone.
 - Prossessorit massatavaraa**, kytkentäverkko kallista erikoisrautaa.
 - Verkon ja muistin **kustannus nousee solmujen määrän kasvaessa** jos halutaan pitää muistikaista vakiona.

- Vanhat peräkkäiset **ohjelmat** eivät nopeudu yhtään ellei niitä **rinnakkais-teta**.
- Rinnakkaisohjelmointia tehdään yleensä yhden ohjelman - monen prosessorin -mallilla (**Single Program Multiple Data**), erityisesti yhteisen muistin koneissa.
 - Joko `for i := 1 to N parallel do A[i] := ...`, tai
 - haarautuminen prosessori-id:n mukaan
 - **Viestinvälitysohjelmoinnissa** (message passing) ei käytetä yhteistä muistia, vaan prosessorilta (-ssilta) toiselle viestejä (send, receive).
 - Kirjasto-operaatiot yleensä sisältävät valmiiksi rinnakkaistetut vektori- ja matriisioperaatiot, yms.
- Välineet:
 - (Tietokone) valmistajan erikoiskääntäjät
 - Fortran -66, -77, -90, High Performance Fortran
 - MPI, PVM, OpenMP
 - SUN Java6.

- **Hinta-teho** -suhde ratkaisee

- 1000GHz PentiumXII ei ole nyt saatavissa vaikka olisi enemmänkin rahaa...
- Paras hinta-tehosuhde PC-proessoreissa on <100e hintaluokassa (Celeron, Sempron).
- Katso google.com, wikipedia.org, jne. laitehankintoja.

- **Moniprosessoritietokoneet** (multiprocessing)
 - Joko **pienimuotoinen rinnakkaistietokone**/laskenta (yleensä **jaetun muistin** moniprosessorikoneessa (SMP)), tai
 - moniajavan käyttöjärjestelmän (ja sovellusten) nopeuttamista kahdella tai useammalla prosessorilla.
 - Esim. cs: $2 \times 4 \times 2,4\text{GHz}$ US VII.
 - **Muistiväylä** usein pullonkaulana (cs: $16 \times \text{DDR2}$).
- **Moniydinprosessorit** (multicore)
 - Samalle lastulle on integroitu usea (lähes) täydellinen prosessori.
 - Kullakin omat toiminnalliset yksikkönsä ja 1-tason välimuisti, yleensä yhteinen 2-tason välimuisti ja ulkoiset liitännät.
 - Edullisempi kotelointi ja emolevy kuin erillisissä moniprosessoriratkaisuissa.
 - Muistiväylä yleensä pullonkaulana.
 - Core Duo/Quad, Pentium D, Athlon X4, Sun T2, PS3, XBox360, ...

- **Monisäie**proessorit (multithreading)

- Kukin fyysinen prosessoriydin suorittaa lomittain useaa prosessia.
- Hyödyntää prosessorin useita toiminnallisia yksiköitä (superskalaa-riproessorissa).
- Näkyvät käyttöjärjestelmälle useana prosessorina (ytimenä).
- Muistiväylä pullonkaulana.
- Sun T2 (8 ydintä, á 8 säiettä), Intel HyperThreading, Tera MTA.

Hajautettu laskenta (distributed computing)

- Yleensä tarkoitetaan usean **erillisen** (jopa maantieteellisesti) **tietokoneen** käyttöä yhden laskennallisen ongelman ratkaisuun rinnakkaisesti.
 - Viestinvälitys (ohjelmointi) mahdollistaa joissakin sovelluksissa hyvin **pitkät viiveet**, **pienen kaistanleveyden** ja **satunnaiset virheet**.
 - Esim. SETI@home.
 - Hajautetun ja rinnakkaisen laskennan raja on varsin häilyvä.
- **Ritilälaskenta (grid)**
 - Uusi nimi vanhalle asialle (hajautettu laskenta, käyttöjärjestelmä, etäkäyttö).
 - Nopeammat verkot tehostavat käyttöä.
 - **Vertaisverkkolaskenta**, **tarjotaan oma** joutava laskentakapasiteetti muiden käyttöön, tarvittaessa **saadaan muilta** (runsaasti) kapasiteettia.

Samanaikainen järjestelmä/ohjelmointi/jne (concurrent)

- Toiminnot ovat **näennäisen yhtäaikaisia** (tapahtuvat yhtä aikaa).
 - Samanaikaisuus voidaan ajatella **hitaan havainnoijan** näkemänä.
 - Erityisesti yksittäisten toimintojen **keskinäistä järjestystä** ei voida nähdä/tietää/määrätä, ellei sitä ole erikseen varmistettu.
 - Voi olla myös aidosti yhtäaikaista kun prosessoreita on monta.
- Toteutetaan joko yhdellä prosessorilla **aikajakoisesti** (time sharing), virtuaaliprosessoreilla, tai usealla oikealla prosessorilla tai tietokoneella.
- Tehtävät/tapahtumat eivät välttämättä ole toisistaan riippuvia.
- **Hajautetut järjestelmät** ovat samanaikaisia luonnostaan.
 - Eri tietokoneiden prosessit suorituvat yhtäaikaisesti ja riippumattomasti.
- **Kommunikaatio** ei-synkronoiduissa HJ on **samanaikaista**.
 - Joustavuuden, vikasietoisuuden ja suorituskyvyn vuoksi hajautetun järjestelmän (kommunikoivat) komponentit ovat yleensä samanaikaisia.

Tosiaikainen järjestelmä (real time)

- Tosi aikajärjestelmän on pystyttävä vastaamaan ympäristön antamiin **määräaikoihin** ja muihin aikarajoitteisiin.
 - Esim. (lentokoneen) autopilotti, (video)puhelu, prosessin hallinta, hälytysjärjestelmä, jne.
- Oleellista ei ole järjestelmän nopeus vaan **nopeuden ennustettavuus (lupaus tietystä vasteajasta)**.
 - Esim. hälytykseen vastataan sekunnissa (tai μ s, ms, t).
 - Lämpötila mitataan kerran minuutissa (ja säädetään lämmitystä).
 - Viesti **kuutataan** 10 s kuluessa.
 - Näppäimen painallukseen **reagoidaan** 10 ms:ssa.
 - Purske lähetetään **sovitulla hetkellä** 0,1 ms tarkkuudella.
 - Aika**eroja** mitataan 10 ns tarkkuudella.
- Tosi aikaohjelmoinnissa kullekin **ohjelmanosalle määritellään aikarajat**, käyttöjärjestelmä jakaa **aikasiivuja**.
- Hajautetuissa järjestelmissä yleensä käytetään väljemmin nimitystä **synkroninen järjestelmä**, kts. jäljempänä.

Säikeistys ((multi)threading)

- Mekanismi **samanaikaisuuden** saavuttamiseksi yhdessä **prosessissa**, "prosesseja prosessissa".
- Yhden prosessin **säikeet jakavat saman muistiavaruuden**, eri **prosesseilla on omat muistiavaruutensa**.
- Vrt. monisäieprosessorit sivu 19.

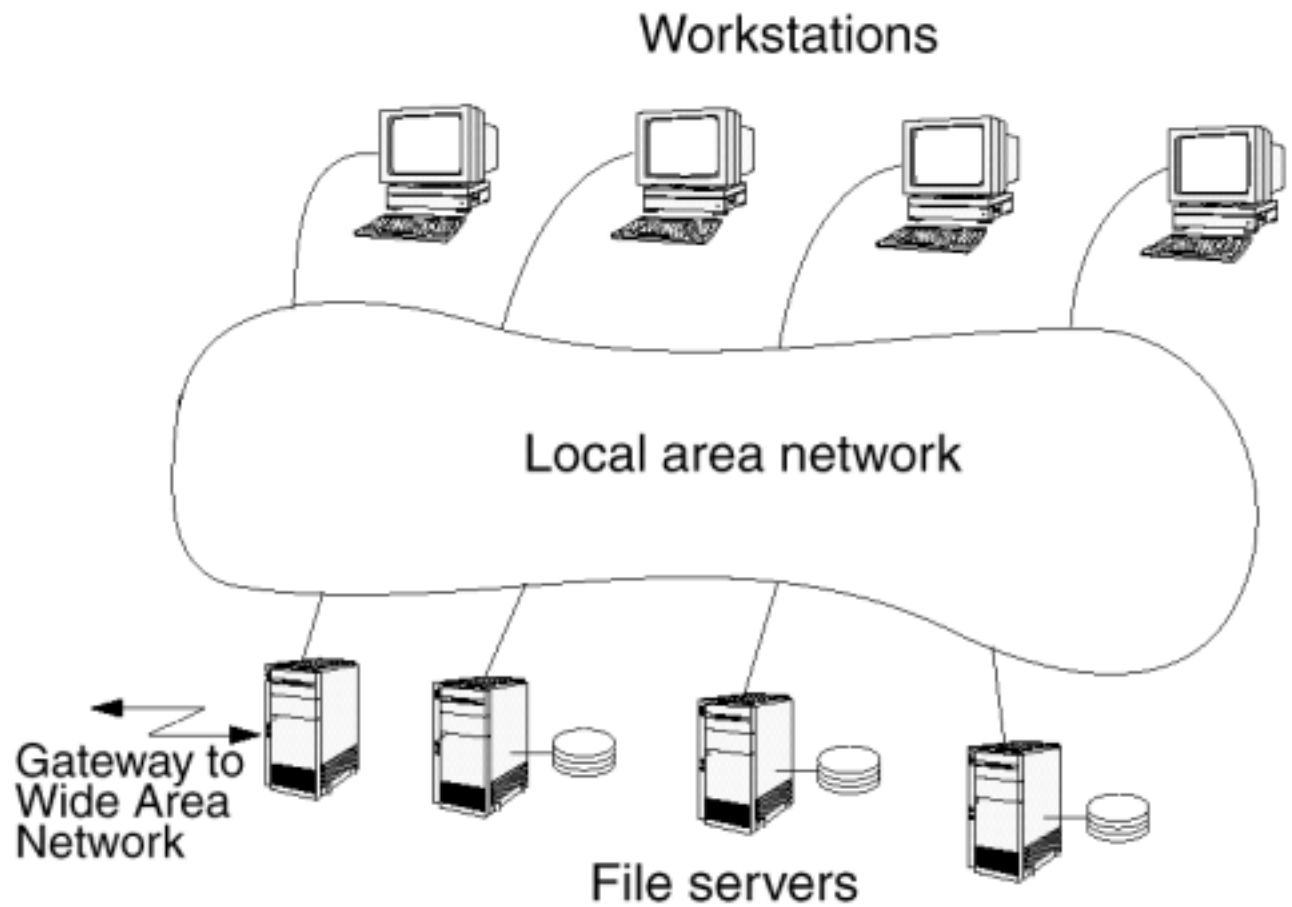
Hajautettu käyttöjärjestelmä

- Yhtenäinen käyttöjärjestelmä/**ympäristö** usealle tietokoneelle.
 - Käyttäjä ei välttämättä tiedä tarkkaan mitä konetta käyttää, prosessit voivat siirtyä koneelta toiselle.

Esimerkkejä hajautetuista järjestelmistä

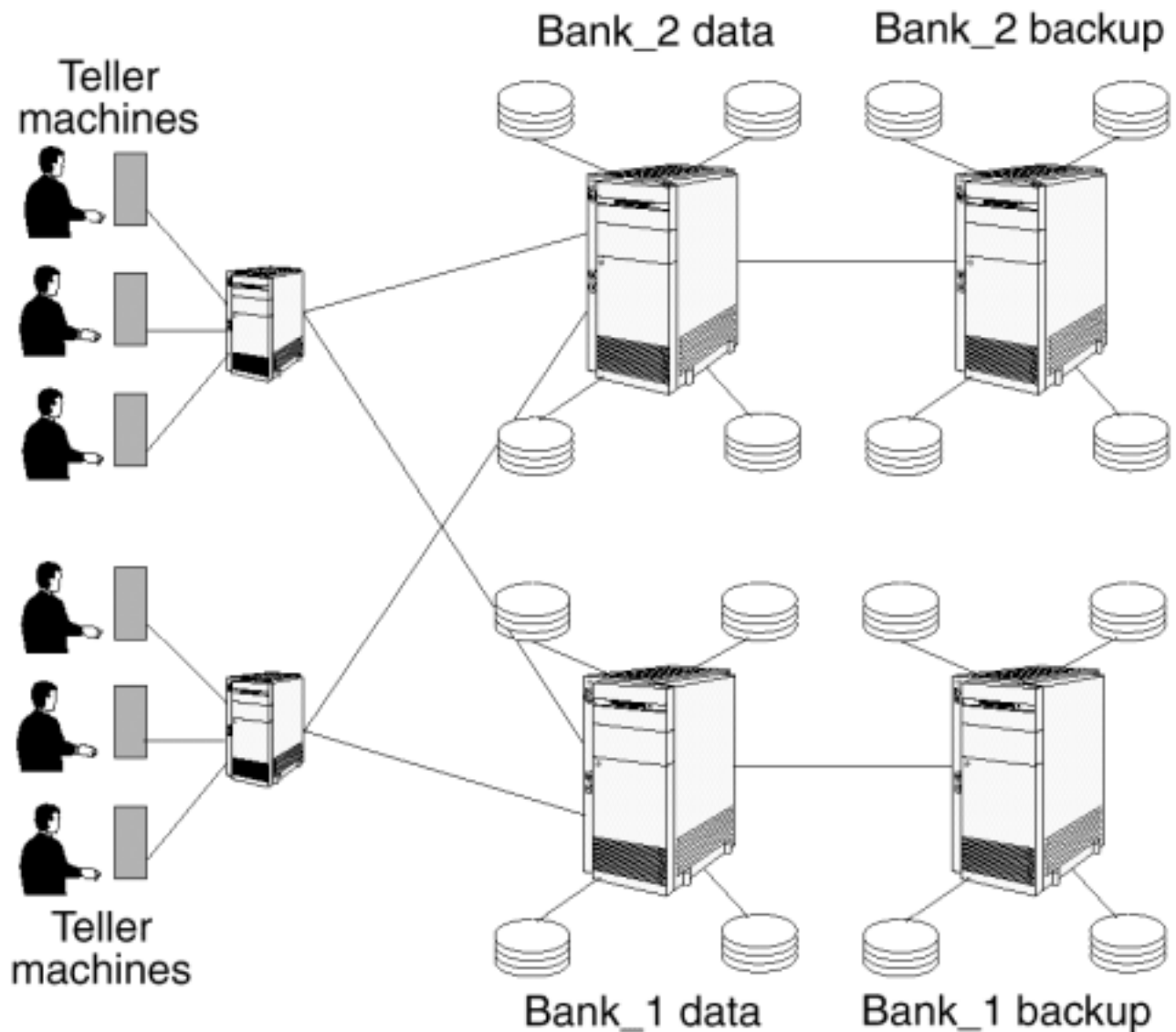
Työasemaverkko

- Henkilökohtaisia työasemia + palvelimia / erikoiskäyttöisiä tietokoneita.
- Yksi **tiedostojärjestelmä**, kaikki tiedostot käytettävissä kaikilta tietokoneilta samalla tavalla ja nimellä.
- Jotkin toiminnot voidaan suorittaa muuallakin kuin käyttäjän "omalla" tietokoneella.
- **Resurssien jakaminen**:
 - Tiedostopalvelimet, laskentapalvelimet, tulostimet, jne.
 - **Hajautettu käyttöjärjestelmä** (tai hajautuksen tuki käyttöjärjestelmässä) parantaa tuntumattomuutta
 - “Metacomputer” CSC:llä, Mosix



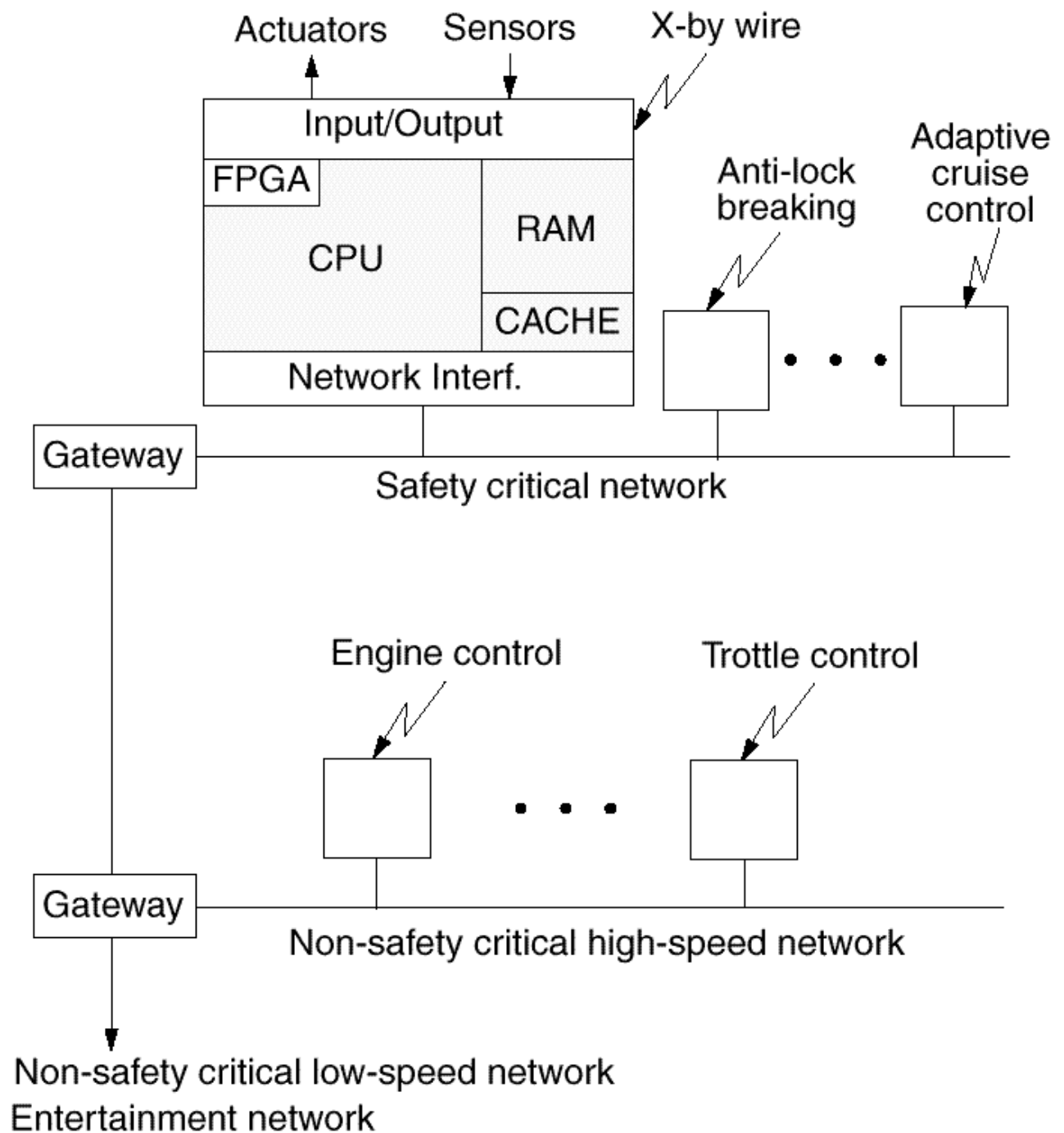
Pankkiautomaattijärjestelmä

- Oltava ehdottoman **turvallinen** ja **luotettava**.
- Kahdennetun tiedon on pysyttävä **eheänä**.
- Samanaikaisia tapahtumia
 - **kaikki-tai-ei mitään**
 - tapahtumaan ottaa osaa useampi pankki ja tili
- Vikasietoisuus
- Korkea saavutettavuus
- **Otto**.
 - 2000 automaattia, ~10 pankkia, ulkomaanyhteydet, 4M käyttäjää, 6 nostoa/s).



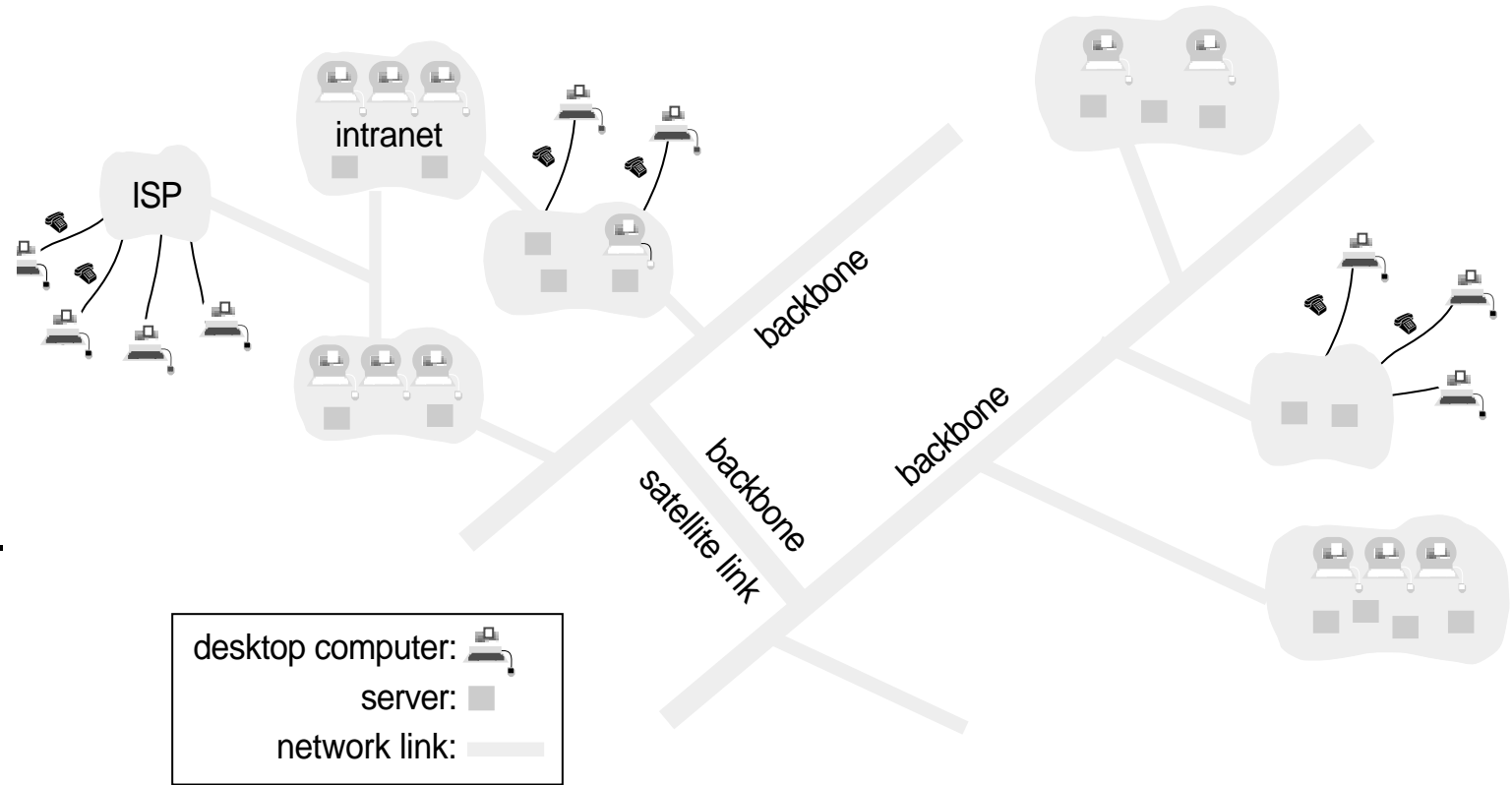
Auto

- Hajautettu **reaaliai-**
kajärjestelmä.
- **Useantaisia** reaali-
aikaisuus- ja luotet-
tavuusvaatimuksia.
- Vikasietoisuus ja
vioista toipuminen.
- **Vian eristäminen**
erittäin tärkeää.
- Jopa kymmeniä
yhteenkytkettyjä
mikroprosessoreja.



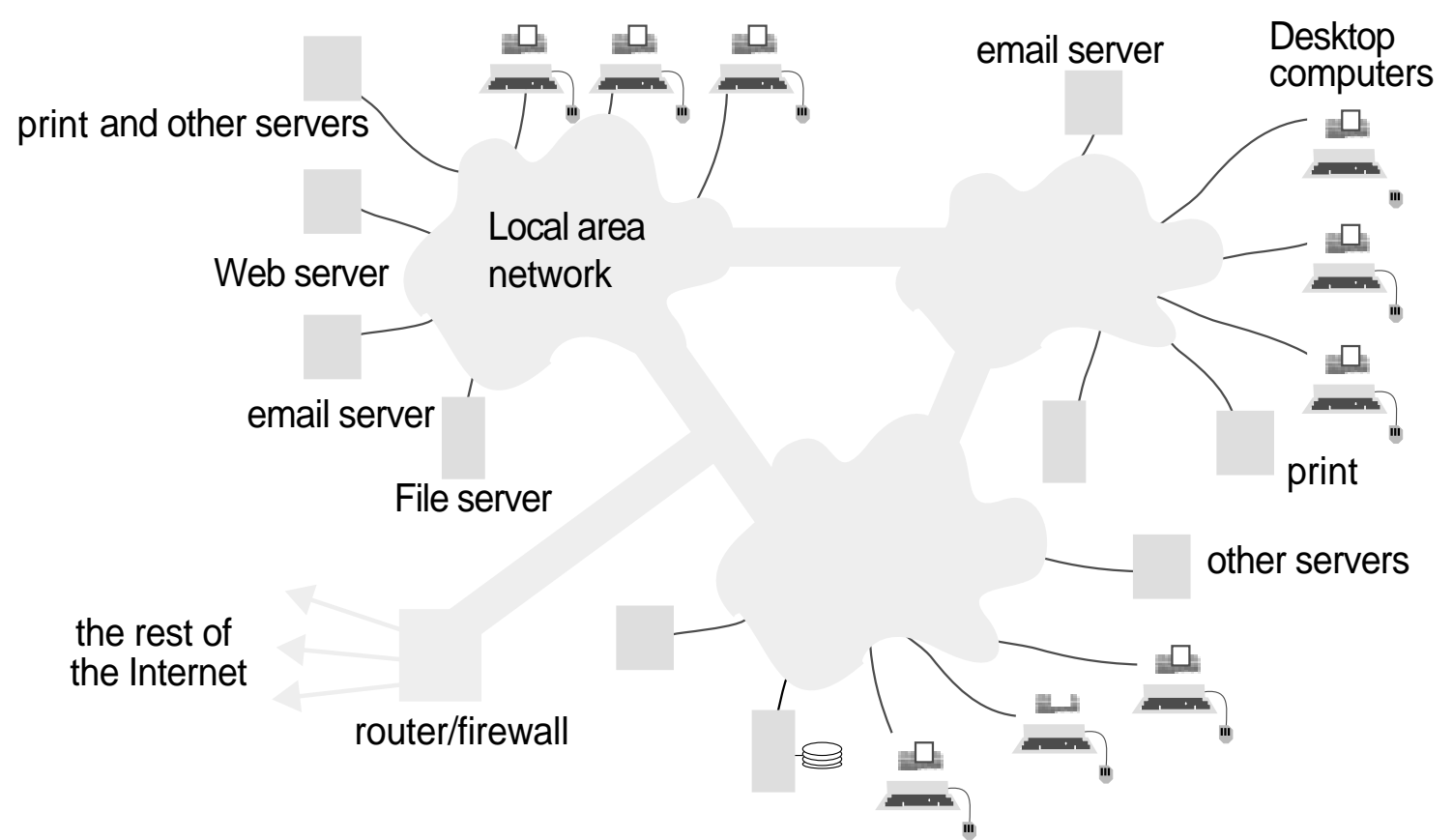
Internet

- Ei sellaise-
naan hajau-
tettu järjes-
telmä.
- **Alusta**
useille hajau-
tetuille jär-
jestelmille.
- Erittäin **epä-
yhtenäinen** laitekanta, käyttöjärjestelmät, fyysiset yhteydet, protokollat, nopeudet, kielet, palvelut, tavoitteet, politiikka, jne.
- **Mahdollistava** tekniikka: **IP** (Internet Protocol)
- **Ei (juurikaan) keskitettyä ylläpitoa/hallintoa.**
 - Perustuu sopimukseen ja **standardeihin**, keskinäiseen **luottamukseen**.
 - Sallii&kestää/jättää huomiotta/kiertää (paikalliset) **pikkuvirheet**.



Intranet

- Paikallisesti hallinnoitu.
- Yleensä IP protokolla.
- Kontrolloitu pääsy Internetistä/iin.
- Yleensä pidetään turvallisempänä luvattoman pääsyn suhteen.
- Mahdollisesti fyysisesti suojellut yhteydet ja laitteet.
- Voidaan toteuttaa virtuaalisesti tavallisena julkisena IP-liikenteenä salauksin ja luotettavin tunnistein (Virtual Private Network, VPN).
- Alusta hajautetuille järjestelmille (kuten Internet).



Mobiilit ja läsnäolevat verkot ja laitteet (ubiquitous)

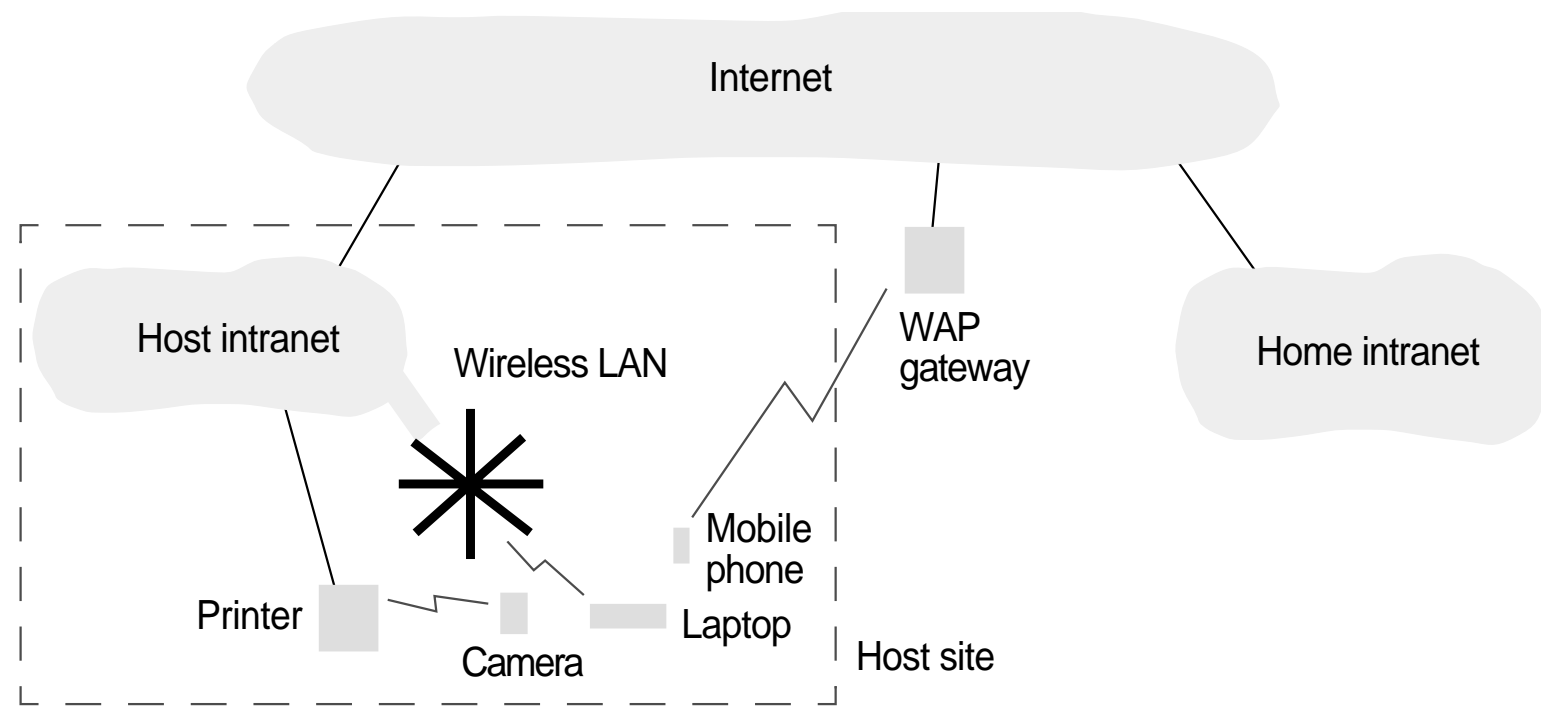
30

- Jokapäiväiset laitteet (kännykkä, TV, digiboksi, digikamera, tulostin, PDA, audio-järjestelmä, auto, kodintekniikka, kodinkoneet, jne.).
- Usein (osallisena) pieniä, **langattomia**, akkukäyttöisiä ja/tai halpoja laitteita, **uusien** laitteiden lisäämisen (olisi) oltava mahdollista.

- **Helppo** liityntä verkkoon tärkeää.

- **Vierailut** toisissa verkoissa.

- **Noviisikäyttäjiä.**



- **Turvallisuus** silti (useimmiten) tärkeää (jossain muodossa).
 - Langattomuus tekee **turvallisuuden ja helppokäyttöisyyden** yhdistämisestä vaikeampaa.
- Etätunnistaminen (RFID).

Miten tietokoneet kommunikoivat?

⇒ Lähettämällä viestejä toisilleen.

Viestivätkö tietokoneet todella? Kyllä/Ei.

- **Sovellukset** (tai käyttöjärjestelmät) kommunikoivat.
- **Prosessit** kommunikoivat.

Viestejä vain lähetetään ja vastaanotetaan

- Onko se nyt niin vaikeaa??
- kts. Lämmittelypeli sivulla 33.

Muunnelmia (**abstrahointeja**) viestien lähettämisestä ja vastaanottamisesta:

- **Tietovirrat** (stream)
- **Monilähetys**, kaikillelähetys (multicast, broadcast)
- Etäproseduurikutsu (Remote Procedure Call)
- **Etämetodikutsu** (Remote Method Invocation)
- Olioiden siirtyminen (Object Migration)
- Monikkomuisti (Tuple Space)
- Verkkotiedostojärjestelmä (Network File System)
- ...

Lämmittelypeli

Säännöt

- Fyysiset viestit, kirjoitetaan paperilapulle.
 - Viesti voi sisältää tietoa, ohjeita, osoitteita, jne
- Yhteydet istualtaan **vain naapureihin** (↔).
- Viestin **lähettäminen**:
 - **Pyydä** naapuria vastaanottamaan, **odota** kunnes hän on valmis.
 - Ojenna lappu hänelle käteen.
- Viestin **vastaanottaminen**
 - **Lupaudu** vastaanottamaan
 - Ota lappu käteen
- **Ei muita viestintätapoja.**
- Näet ja kommunikoi vain naapuriesi kanssa.
- **Itseksesi** voit tehdä mitä vain.

Tehtäviä

- **Luku**
- Yhteisymmärrys jostakin, järjestäminen, minimi, ...

Lisää vaikeutta

- Jatkuvasti **muuttuva** opiskelijamäärä (luku reaaliajassa)
- Opiskelijoiden **siirtyminen**.
- Ei näkö/kuuloyhteyttä
- **Pysähtyvä** (blocking) lähetys ja vastaanotto.
- **Vialliset** (tai vilpilliset) yhteydet / opiskelijat.
- Niin **nopeasti** / tehokkaasti kuin mahdollista.
 - Minimoidaan kokonaisaikaa, yhteistä työaikaa, viestien määrää, ...

Helpotuksia

- Tieto (kartta) kaikista osanottajista (sijainneista).
- Yksiselitteinen peräkkäinen järjestys.
- Kommunikointi myös muiden kuin naapurien kanssa (kenelle tahansa).
- Viestien automaattinen **reititys**.
- Lähetys **kaikille** / osajoukolle kerralla (broadcast / multicast).
- Luotettavat yhteydet / virheilmoitukset.
- Vastaanotto keneltä tahansa (kaikista suunnista yhtä aikaa).
- Peräkkäinen ratkaisu.

HJ suunnittelulähtökohtia

⇒ Kurssin hajautettujen järjestelmien osuus pikakelauksella.

- Kuten muunkin järjestelmän suunnittelu,
 - mutta joitakin lisäpiirteitä on huomioitava.

Erityisesti hajauttamisesta seuraa tarpeita/haasteita

- Kommunikaatio (s. 37)
- Samanaikaisuus (s. 39)
- Suorituskyky ja skaalautuvuus (s. 43)
- Monimuotoisuus (heterogeneity) (s. 47)
- Avoimuus ja joustavuus (s. 51)
- Luotettavuus ja vikasietoisuus (s. 53)
- Turvallisuus (s. 57)
- Tuntumattomuus (läpikuultavuus, transparency) (s. 59)

⇒ Järjestelmän osien on pystyttävä **kommunikoimaan** jotta ne voisivat **toimia yhteen**.

Tukea tarvitaan kahdella (pää)tasolla

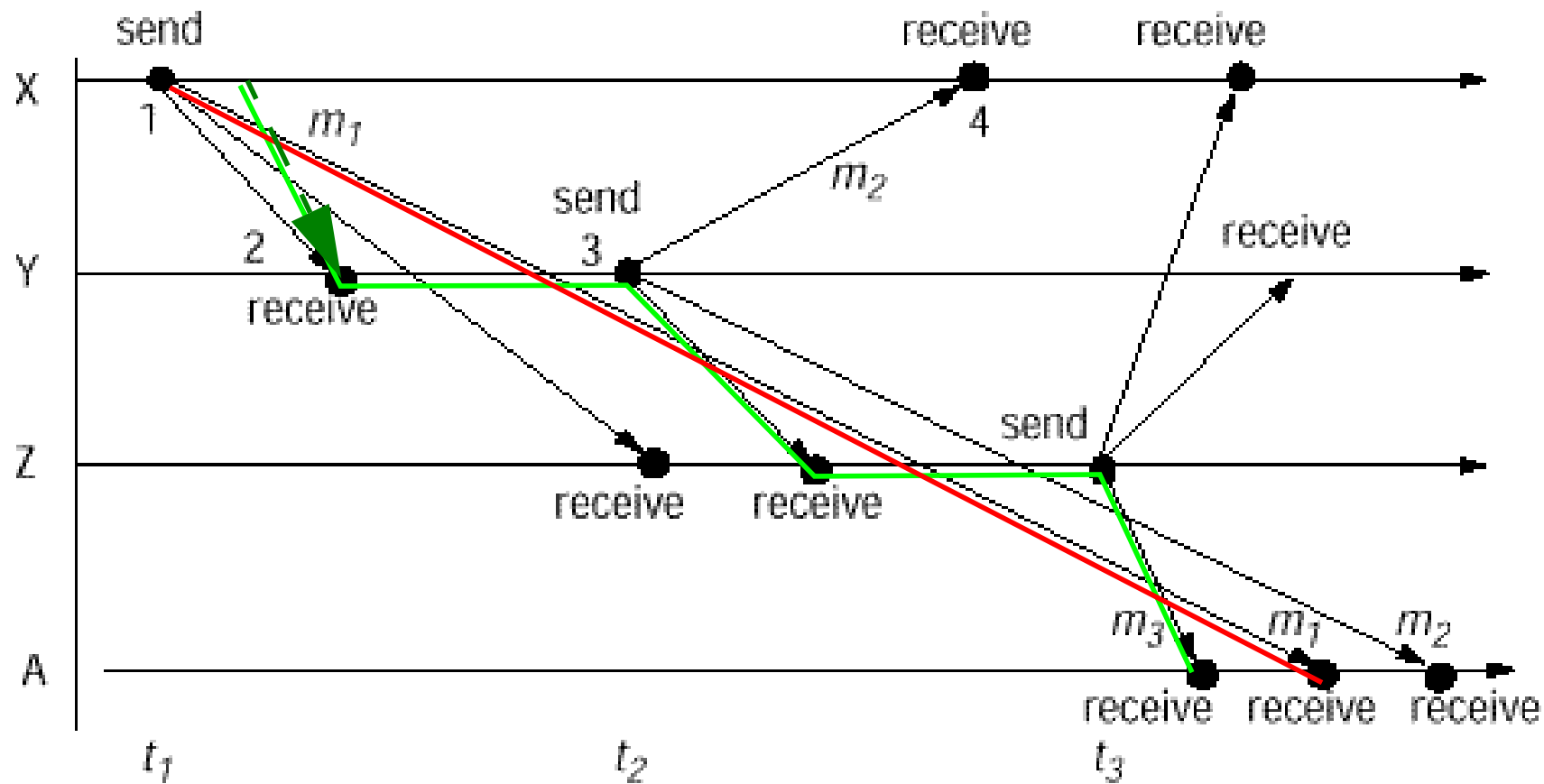
- Kommunikaatio**infrastrukturi** (fyysiset yhteydet, verkko-ohjelmistot).
 - Luku 4: Kommunikaatio hajautetuissa järjestelmissä (sivu 189).
 - Verkkoprotokollat (IP, jne).
 - Kommunikaatio-**operaatiot**, esim:
 - **send & receive** (viestinvälitys)
 - etäproseduurikutsu (RPC)
 - **etämetodikutsu** (RMI)
 - Tietoliikenteen kursseilla lisää.

- Sopivat kommunikaatiomallit (toiminnot) (ja niiden toteutus):

- asiakas-palvelin (client-server): asiakas pyytää ja palvelin vastaa
- ryhmän monilähetys (multicast): viestin kohteena on prosessijoukko
- vertaisverkko (peer-to-peer)
- Luku 3: Hajautettujen järjestelmien mallit (sivu 101)

- ⇒ Hajautettu järjestelmä on samanaikainen luonnostaan.
- ⇒ Samanaikaisuudesta seuraa **epädeterministisyyttä** (joka meidän on saatava kuriin).
 - Samanaikainen tiedon päivitys voi johtaa **epämääräiseen tilaan** jollei päivityksiä koordinoida.
 - Tapahtumien **täydellinen peräkkäistäminen** johtaa hitaampaan ja skaalautumattomaan järjestelmään ja jopa huonontaa vikasietoisuutta.

- Jollekseen erikseen synkronoi, kaikki (“toisistaan riippumattomat”) tapahtumat voivat toteutua **missä järjestyksessä tahansa**, tai yhtä aikaa.
 - Erityisesti **viestinvälitysaika** on epädeterministinen.



Jotta kommunikaatio olisi joustavaa, osaaottavien komponenttien on oltava samanaikaisia.

- Esim. samanaikaisesti:
 - odotetaan käyttäjän toimenpiteitä
 - päivitetään näytön ikkunan sisältöä
 - lähetetään uusia pyyntöjä (asiakkaana)
 - odotetaan (useiden) edellisten pyyntöjen valmistumista
 - odotetaan uusia pyyntöjä (palvelin)

- **Palvelimen** pitäisi pystyä vastaanottamaan uusia yhteyksiä ja palvelemaan useita "**pitkäkestoisia**" (persistent) asiakkaita samanaikaisesti.
 - Jopa käsitellä useita eri asiakkaiden pyyntöjä samanaikaisesti.
 - Paljon käytetyssä palvelussa tämä on jopa pakollista.
 - Jotta kokonaissuorituskyky (läpäisy) paranisi, tarvitaan myös lisää laskentavoimaa (rinnakkaislaskentaa).
 - Joissakin palveluissa voidaan (hetkittäin) palvella muutamaa asiakasta rinnakkain jopa yhdellä prosessorilla (yksi prosessorilla, yksi levyhakua tekemässä, yksi lähetyksessä, yksi vastaanotossa).
- Helpoimmillaan perustetaan **uusi prosessi** (tai säie) kutakin uutta yhteyttä kohti.
- Prosessit on synkronoitava (vaikkapa semaforeilla) jos ne käyttävät (päivittävät) samoja resursseja.
- Saman prosessin sisällä samanaikaisuus tehdään **säikeillä**.

⇒ Peräkkäisjärjestelmiin verrattuna monet tekijät vaikuttavat hajautetun järjestelmän suorituskykyyn, esim.:

- Kunkin yksittäisen **palvelimen ja asiakaskoneen suorituskyky**.
 - Kumpi tahansa voi olla pullonkaula.
 - Prosessoriteho, muistiväylä, levyväylä, I/O-väylä.
 - Esim. puhelinnumeropalvelin vs. karttapalvelin.
 - Esim. karttapalvelin valmiilla/generoiduilla kartoilla.
 - haku avaimella / vapaasanahaku / säännöllisen lausekkeen haku.
- **Kommunikaatioverkon** nopeus (**viive**, **kaista** eri kohdissa).
- Luotettavuus- (vikasietoisuus-) mekanismit (palvelimien monistus voi toisaalta nopeuttaa, toisaalta hidastaa).
- **Kuormantasauksen joustavuus**.

Hajautus voi **parantaa tai/ja huonontaa** suorituskykyä

- Useita monistettuja palvelimia
 - **Lisää kapasiteettia** käsitellä asiakkaiden pyyntöjä.
 - On **hitaampaa päivittää monistettuja** tietokantoja säilyttäen edelleen kantojen yhtenäisyys.
- Verkkoviive, resurssien sijainti.
 - Hajauttaminen voi **lisätä uuden viiveen** jos osa palvelusta siirretään kauemmas.
 - Hajauttaminen voi **vähentää viivettä** jos osa/kopio palvelusta tuodaan lähemmäs.

- Järjestelmän tulisi säilyttää tehokkuutensa vaikka käyttäjien ja resurssien määrä kasvaisi huomattavasti, esim. Internet **DNS**

Tietokoneiden määrä Internetissä [http://www.isoc.org/]		
Vuosi	Lukumäärä	WWW-palvelimia
1979	188	0
1989	130 000	0
1999	56 000 000	5 500 000
2005	353 000 000	

- Resurssien **lisäämisen kustannus** pitäisi olla kohtuullinen;
- Skaalattaessa hyötysuhde (asiakasta/palvelin) ei saisi huonontua .
- Järjestelmän rajoitukset eivät saa tulla vastaan (esim. osoitteiden määrä, taulukkokoot, jne).
 - Esim. IPv4:n 32-bittinen osoiteavaruus.

- **Vertaisverkko** (peer-to-peer) mahdollistaa hyvän skaalautuvuuden (kunhan sitä ei toteuteta kaikki-yhteen tai kaikki-kaikille) (esim NTP, SMTP).
- **Asiakas-palvelin** on yhtä skaalautuva kuin sen palvelin ja palvelimen verkkoyhteys.
- Palvelinten **hierarkia** toimii usein hyvin, esim. DNS.
- $O(N)$ (lineaarinen solmujen määrän suhteen) aikavaativuus ei ole skaalautuva.
 - $o(N)$ ehdoton edellytys, **$O(\log N)$ toivottava**, jopa $O(1)$ mahdollinen, enemmän kuin $O(N)$ on kestävä.

Monimuotoisuus (heterogeneity)

⇒ Hajautetut järjestelmät ovat tyypillisesti **monimuotoisia**:

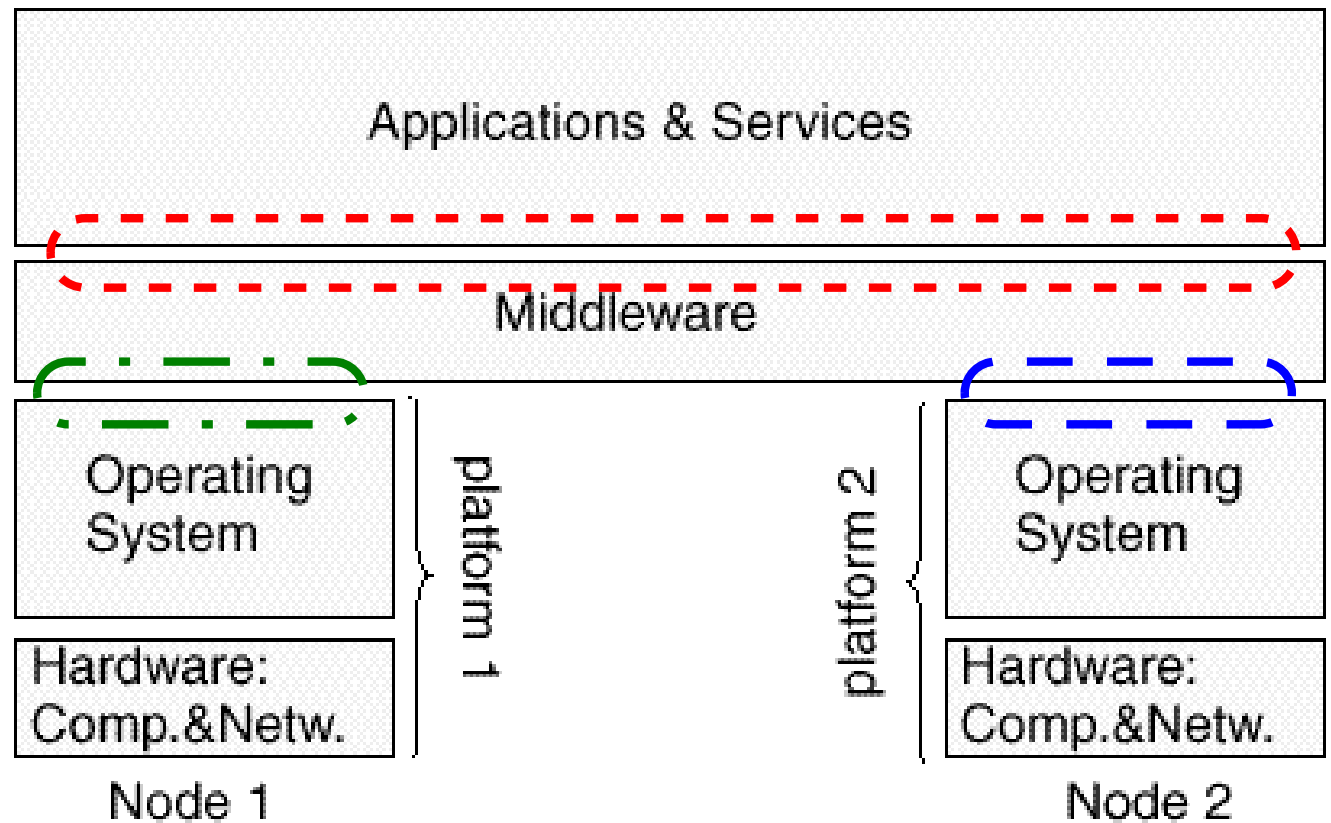
- Monenlaista **laitteistoa**
 - keskuskoneita, työasemia, PC, PDA, erilaisia palvelimia, ...
- Monenlaisia **käyttöjärjestelmiä**
 - UNIX, MS Windows, tosiaikakäyttöjärj., Symbian, PalmOS, ...
- ohjelmointikieliä
- usean eri **kehittäjä**(organisaatio)n toteutuksia/ohjelmistoja
- **erikoislaitteita**
 - automaatteja, puhelinlaitteita, verkkolaitteita, robotteja, kämmenlaitteita, älykortteja, teollisuusautomaatiota, jne
- erilaisia **verkkoja** ja protokollia
 - RS-232, USB, BT, ADSL, SDSL, Ethernet, FDDI, ATM, GSM-CS, GPRS, W-CDMA, LTE, WLAN, Flash-OFDM, WAP, TCP/IP, Novell Netware, jne.

Siirtyvä (vaeltava) ohjelma (mobile code)

- Eräs ratkaisu (ja haaste) monimuotoisuuteen.
- Esim., Java appletit, ActiveX.
- Vaatii virtuaalikoneen (tai yhtenäisen alustan).
- Vaatii yhä huolellisemman lähestymisen turvallisuuteen.

Eräs ratkaisu monimuotoisuuteen

- **Väliohjelmisto (middleware)**, ylimääräinen ohjelmistokerros jolla monimuotoisuutta piilotetaan.
- Tarjoaa **alustariippumattomia palveluita** sovelluksille.
 - Sama sovellus voidaan kääntää erilaisille alustoille muuttamatta (kunhan valittu väliohjelmisto on toteutettu ko. alustalle).
 - Eri alustat voivat kommunikoida ilman yhteensopivuusongelmia.
 - Tarjoaa usein **korkeamman tason** palveluita kuin käyttöjärjestelmä.
 - Voi sisältää myös **sovellusakohtaisia** palveluita.



- Toteuttaa **kommunikaatioprimitiivit**, tiedon (esitystapa)**muunnokset**, virtuaaliyhteydet, palveluiden **paikallistamisen**, viittaukset, jne.
- **CORBA** (Common Object Request Broker Architecture)
 - kieliriippumaton, alustariippumaton
- Java, **JavaRMI** (alustariippumaton, kielisidonnainen)
- DCOM/**.NET** ("kieliriippumaton", alustasidonnainen)
- Eräällä tavalla jopa TCP/IP voidaan tulkita matalan tason väliohjelmistoksi, mutta TCP-kutsut ovat erilaiset Unix:ssa ja WinNT:ssä,
 - Tosin WinNT:stä voidaan tehdä Posix yhteensopiva ja Linux:lle voidaan kääntää .NET sovelluksia.
- XML, XML-RPC

- Palveluiden tulisi olla yhtäläisesti kaikkien (paikallisten ja etä-) käyttäjien saatavilla (**määriteltyjen käyttöoikeuksien rajoissa**).
- Tulisi olla helppoa (tai ainakin mahdollista) toteuttaa, asentaa ja testata **uusia (lisä) palveluita** ja uusia liittymiä.
- Käyttäjien ja sovelluskehittäjien pitäisi pystyä kirjoittamaan ja käyttämään omia **asiakasohjelmiaan** (ja jopa palveluitaan).

Avoimuuden avaimet:

- **Standardoidut julkiset liittymät ja protokollat** (esim. Internetin kommunikaatioprotokollat).
- Yhtenäiset liittymät jaettuihin resursseihin.
- Monimuotoisuuden tukeminen (sopivan väliohjelmiston avulla).

⇒ **Avoimuus ei heikennä turvallisuutta tai yhteensopivuutta, itseasiassa päinvastoin.**

- Kaikkien turvakäytäntöjen pitäisi olla **turvallisia** vaikka niiden **määrittelyt ja toteutukset julkaistaisiin**.

Päivitettävyys

- Jos liittymää, protokollaa, tms. **muutetaan epäyhteensopivaksi** edelliseen versioon nähden, olisi pystyttävä päivittämään kaikkia palvelimia ja asiakkaita yhtä aikaa!
 - Yhtäaikainen päivitys **mahdotonta** (kohtuutonta) jos palvelimia ja asiakkaita on paljon, tai palvelun olisi oltava jatkuvassa käytössä.
 - On varsin mahdotonta jos asiakkaita ja palvelimia ovat **toteuttaneet eri organisaatiot**.
- Liittymien päivitysten olisi siis oltava **yhteensopivia alas- ja ylöspäin**.
 - Alkuperäinen liittymä (protokolla) on suunniteltava **joustavaksi** (laajennettavaksi).
 - Sovellusten tulisi pystyä ohittamaan tuntemattomat kentät.
 - Protokolla(versio)**neuvottelu** on joustavin (ja vaativin).

Luotettavuus ja vikasiETOisuus

- ⇒ N -komponenttiossa HJ on ainakin N kertaa enemmän vikaantumismahdollisuuksia, ja siis **vikoja**, ja **vikatyyppejä**.
- ⇒ Päinvastoin kuin paikallisessa järjestelmässä, vika on yleensä **osittainen** (vain yksi/muutama komponentti vikaantuu).
 - Ei-vikaantuneiden komponenttien on selvittävä katkenneista yhteyksistä / vikaantuneista solmuista siististi.
 - Ainakin aiheuttamatta **lisätuhoja**.
- ⇒ Yksi HJ rakentamisen **tavoitteista on parantaa luotettavuutta**.
 - Vaikka vikapaikkoja on enemmän.

Vikapaikkoja

- Tietokone, komponentti, johto, auringonpilkut, verkkolaite, reititys, törmäys, lukkiuma, ohjelmointivirhe, virransaanti, murtautuja, jne.

- Kun osa järjestelmästä poistuu vahvuudesta (syystä tai toisesta), **lopun järjestelmästä pitäisi toimia edelleen** (vähemmän resurssein).
 - Rajoittuneet palvelut (tai miel. vain rajoittunut suorituskyky).
- Järjestelmässä tulisi olla mahdollisimman vähän (jos ollenkaan) **kriittisiä resursseja** (joita ilman mikään ei toimi).
- Avainosat (laitteet ja ohjelmistot) tulisi voida **kahdentaa**.
- (Järeän) hajautetun järjestelmän pitäisi toimia **24/7/365**.
 - Päivitykset, varmistukset, käynnistykset jne. kestävät paljon kauemmin kuin yksittäisessä järjestelmässä.
 - Yksittäisiä vikaantuneita osia pitäisi pystyä vaihtamaan lennosta.
 - Monistetussakin järjestelmässä palvelun pitäisi pelata 24/7/365.
 - Jopa **varmistusten, päivitysten** aikana (vrt. kuitenkin pankkiautomaatit (sivu 26)).

- www.amadeus.com
 - Matkatoimistoja 102 000
 - Lentoyhtiöitä: 729
 - Autovuokraamoita: 36 000
 - Hotelleja: 84 000
 - Varauksia (2008): 526 miljoonaa (keskimäärin 17/s).
 - Tapahtumia 2 500/s (kyselyitä / varaustoimien aloituksia jne).
 - 7000 viestiä/s.
 - Keskimääräinen vasteaika 0,3 s (?).
 - Varaus voi olla alustavassa tilassa minuutteja tai pari päivää, sitten se toteutetaan tai puretaan.
 - Useaosaisia varauksia.

- Pääasia on luotettavuus, järjestelmän on havaittava viat ja toimittava sen mukaan järkevästi:
 - **Ohita** virhe: jatka toimintaa vaikka hitaammin, mutta ilman tiedon häviämistä.
 - Esim. ohita viallinen paketti, pyydä uusi kopio.
 - **Vikaannu tahdikkaasti**: reagoi virheeseen **odotettavalla tavalla** (ehdottomasti dokumentoidulla tavalla), lopeta toiminta vaikka hetkeksi, mutta ei hävitetä tietoa.

Tiedon eheys

- Järjestelmä ei saa hävittää tietoa ja eri palvelimilla olevien kopioiden on oltava **yhteneväisiä** (consistent).
- Mitä useampia kopioita ylläpidetään, sitä **parempi saavutettavuus** ja luotettavuus, mutta sitä **vaikeampaa ylläpitää yhtäpitävyyttä**.

- Paikalliset järjestelmät on helppo suojata (estetään kaikki yhteydet).
- Hajautettujen järjestelmien on **pakko kommunikoida**.
 - Yleensä myös "**vihamielisellä**" alueella (kuten Internetissä).

Informaation/palvelujen turvallisuus

- **Luottamuksellisuus**
 - Salattava materiaali ei saa paljastua asiaankuulumattomalle taholle
 - Osanottajien luotettava tunnistaminen.
- **Eheys**
 - Suojaus väärää muuttamista ja muuttumista vastaan.
- **Saavutettavuus**
 - Resurssien olisi oltava luvallisten käyttäjien saavutettavissa (palvelunestohyökkäyksiä vastaan).

⇒ Kommunikointi tietokoneiden/ohjelmien/organisaatioiden/henkilöiden välillä kuuluu hajautettujen järjestelmien luonteeseen jo määritelmän mukaan.

- Turvariskit ovat aina olemassa.
- Turva-asiat liittyvät läheisesti kaikkiin yllä oleviin aiheisiin, erityisesti luotettavuuteen ja vikasietoisuuteen.

Tuntumattomuus (läpikuultavuus, transparency)

59

- Ideaalisessa hajautetussa järjestelmässä **käyttäjä ja sovellusohjelmoija** toimivat kuten paikallisessa järjestelmässä.

⇒ **Kuinka saavuttaa illuusio yhdestä järjestelmästä?**

ISO määrittää kahdeksan tuntumattomuuden muotoa (tavoitetta (tai unelmaa)) (ja me (CDK4) lisäämme vielä kaksi):

⇒ **Näitä ei tietenkään aina kaikkia ole mahdollista / järkevää tavoitella tai toteuttaa.**

Saannin (access) tuntumattomuus

- Paikallisia ja etäresursseja **käytetään samanlaisilla operaatioilla**.
- Esim. NFS vs. FTP.

Sijainnin (location) tuntumattomuus

- Käyttäjä ei tiedä (tarvitse tietää) missä laitteisto- tai ohjelmistoresurssit ovat. Niiden nimet eivät myöskään sisällä tietoa **sijainnista**.
- Esim. Simo.Juvaste @ iki.fi.

Paikanvaihtotuntumattomuus (migration, relocation)

- Resursseja on voitava muuttaa **paikasta toiseen** ilman, että niiden nimiä muutetaan tai niiden siirtäminen vaikuttaa käyttämiseen.
- Esim. palvelin voidaan siirtää toiseen paikkaan jos nimi ja osoite pidetään samoina.

Siirtymistuntumattomuus (mobility) (ei ISO:n listalla)

- Resurssit voivat olla **käytettävissä myös siirron aikana**.
- Esim. kännykän tukiasemanvaihto.

Monistustuntumattomuus (replication)

- Järjestelmä voi tehdä tiedostoista tai muista resursseista **kopioita** (esim. suorituskyvyn tai luotettavuuden parantamiseksi) ilman, että käyttäjä sitä huomaa/häiriintyy.
 - Esim. usea kopio samasta tiedostosta, kuhunkin pyyntöön vastataan lähimmästä kopiosta.

Samanaikaisuustuntumattomuus (concurrency)

- Käyttäjä ei huomaa/häiritse järjestelmän **muita käyttäjiä** (vaikka he käyttäisivät samoja resursseja yhtä aikaa).

Vikatuntumattomuus (failure)

- Sovellusten tulisi pystyä suorittamana tehtävänsä vaikka järjestelmän muissa osissa tapahtuisi tilapäisiä/osittaisia virheitä.
- Esim. TCP, SMTP.

Suorituskyvyn (performance) tuntumattomuus

- Kuorman muutosten ei pitäisi vaikuttaa toimintaan / vasteaikaan. Vasteajan tuntumattomuutta on usein vaikea/tarpeeton toteuttaa, lähinnä (automaattisilla) dynaamisilla konfiguraatiomuutoksilla (kuormantasauksella) voidaan jotain tehdä.

Skaalaustuntumattomuus (scaling)

- Järjestelmää pitää voida skaalata (lisätä palvelimia, asiakkaita, yms.), muuttamatta järjestelmän rakennetta tai rajapintoja.

Pysyvyystuntumattomuus (persistence) (ei ISO:n listalla)

- Resurssit voidaan tallentaa nopeaan tai pysyvään (online, offline) **varastoon** ilman, että se näkyy käyttäjälle (paitsi ehkä viiveenä).

Luku 2

Samanaikaisohjelmointi

Mitä samanaikaisuus on? (s. 65)

Mihin samanaikaisuutta ja sen hallintaa tarvitaan? (s. 66)

Prosessit, säikeet (s. 70)

Säiepeli (s. 74)

Javan säikeet ja niiden synkronointi (s. 78)

Tietorakenteita Javan säikeidenväliseen kommunikaatioon (s. 96)

Mitä samanaikaisuus on?

⇒ Näennäisesti yhtä aikaa tapahtuvia asioita.

- Missä järjestyksessä tahansa (tai epäjärjestyksessä).
- Toisistaan riippumattomassa järjestyksessä.
- Hajautetussa järjestelmässä aidosti yhtä aikaa.
- Rinnakkaistietokoneessa aidosti yhtä aikaa.
- Yksiprosessorikoneessa aikajaetusti näennäisesti yhtä aikaa.

⇒ Keskitytään nyt yhteeseen koneeseen kerrallaan.

- Samanaikaisilla tapahtumilla on (voi olla) samanaikainen pääsy jaettuihin resursseihin.
 - Pääsyä on kontrolloitava, erityisesti jos resurssia muutetaan.

Mihin samanaikaisuutta ja sen hallintaa tarvitaan?

66

- **Usea henkilö**/järjestelmä/toiminto käyttää/päivittää samaa resurssia.
 - Editoi samaa tiedostoa.
 - Tekee muutoksia samaan tietokantaan.
- Ohjelma tekee **useita asioita** yhtä aikaa.
 - Odottaa käyttäjän toimenpiteitä.
 - Päivittää näyttöä.
 - Vastanottaa tietoa verkosta.
 - Lähettää tietoa verkkoon.
 - Käyttää massamuistia.
 - Tulostaa.
- Palvelin palvelee **useaa yhteyttä** yhtä aikaa.
 - Ja odottaa uusia yhteyksiä.
 - Ehkä tekee jotain paikallista työtä, esim. päivittää tietokantaa tai lisää tietoja.

Mihin samanaikaisuutta tarvitaan (v2)?

67

- **Suorituskyvyn** parantamiseen
 - Esim. turha odottaa hidasta siirräntää, tehdään jotain hyödyllistä saman aikaan.
 - Hyödynnetään moniprosessori/ydin/säie arkkitehtuureja.
- **Luotettavuuden** parantaminen
 - Muu ohjelma voi toimia vaikka yksi osa vikaantuu tai joutuu ongelmiin esim. siirräntän kanssa.
- Komponenttien **yksinkertaistaminen**
 - Erilliset toimet voidaan koodata erikseen miettimättä (paljoa) kokonaisuutta (vuorottelua muiden toimien kanssa).

Mitä haittaa (vaikeutta) samanaikaisuudesta on?

68

- Yhteisten resurssien käytön **koordinointi** tehtävä (huolella).
- Ohjelman **lopun** havaitseminen vaikeampaa.
- **Virheiden havaitseminen** (ja virheestä ilmoittaminen) on vaikeampaa.
 - Muiden säikeiden/prosessien on varauduttava toisten säikeiden virheisiin.
- Virheistä **toipuminen** voi olla vaikeampaa (mutta mahdollista).
 - Vikaantunut säie/prosessi voidaan käynnistää uudelleen (jos sitä joku vahtii ja tila saadaan palautettua).
- Ohjelman kattava **testaaminen** (paljon) vaikeampaa, jopa mahdotonta.
 - Ohjelman suunnittelussa ja toteutuksessa oltava systemaattisempi ja huolellisempi.
 - Esim. kaksi prosessia käyttävät samaa kriittistä resurssia keskimäärin 15 minuutin välein, 10ms kerrallaan.
 - Törmäys tapahtuu keskimäärin 2,5 vuoden välein!

Esimerkkejä koordinoinnin tarpeesta

- **Tuottaja-kuluttaja**, yhteinen jono.
 - Monta tuottajaa, monta kuluttajaa, yhteinen jono.
- Mikä tahansa **lue tieto, kirjoita päivitetty tieto** -operaatio.
- **Kysely- ja varausjärjestelmä** samoihin resursseihin.
- Käyttäjä (käyttöliittymäsäie) keskeyttää toisen säikeen toiminnan kesken operaation.
- Tietojen levyille tallentaminen kesken toiminnan (käyttäjää häiritsemättä).
- Tiedonhaku merkki merkiltä kirjoitettaessa (täydennys, tms.).
- Taustalla toimiva tulostusjono tai koko tulostussäie käyttää samaa dataa.

- **Prosessi** (process): **käyttöjärjestelmän suoritusyksikkö** jolla on **oma muistiavaruus** ja muut resurssit.
 - Prosessit voivat toki useimmissa käyttöjärjestelmissä perustaa yhteisen muistilohkon.
- **Säie** (thread): **prosessin sisäinen suoritusyksikkö** joka käyttää prosessin **yhteisiä resursseja** (muuttujia, tiedostokahvoja).
 - Säikeet toki voivat varata omiakin muuttujia (samassa muistiavaruudessa).
- Jos koneessa on monta prosessoria/ydintä/prosessorisäiettä (käyttöjärjestelmälle näkyvää prosessoria):
 - **Prosessit** suorituvat aidosti **rinnakkaisesti**.
 - **Säikeet** yleensä suorituvat **aikajakoisesti** yhdessä prosessissa (yhdellä prosessorilla).
 - Joissakin järjestelmissä säikeitä voidaan suorittaa **rinnakkainkin**, esim. Solaris pthread, Java säikeet joillakin alustoilla (uudemilla JVM).

Prosessit (C:llä)

- Suorituksen perusyksikkö useimmissa käyttöjärjestelmissä.
 - Prosessit suorituvat samanaikaisesti ja toisistaan riippumattomasti.
- Jokaisella prosessilla on oma muistiavaruus, rekisterit, ymäristö, ohjelmalaskuri, jne.
- Uusia prosesseja perustetaan (unix, Posix) kutsulla

```
pid_t fork();
```

 - joka luo kutsuneesta prosessista **identtisen kopion** ja palauttaa lapsiprosessin **prosessitunnistenumeron** (PID).
 - Lapsiprosessi ryhtyy suorittumaan samasta kohtaa (*fork()*:n jälkeinen lause) ainoana eronaan *fork()*:n palautusarvo 0.
- Useimmat käyttöjärjestelmät toteuttavat kopioinnin vasta sitä mukaa kun jommankumman prosessin muistia muutetaan (copy-on-modify).
 - Vaikka kukin prosessi saa oman virtuaalimuistialueensa, aluksi kopioidaan siis vain vähän, ja jollei jotain muistisivua muuteta, sitä ei kopioida koskaan.
 - Prosessit **eivät voi kommunikoida suoraan "jaetun" muistin** kautta, sillä jos jompikumpi muuttaa tavuakaan, toinen ei näe muutosta.

- **IP**
 - **TCP/UDP** yhteys toimii mainiosti myös localhost:ssa.
 - Helposti muutettavissa hajautetuksi järjestelmäksi.
 - Tuttu (kohta).
- **Putket** (pipe)
 - Tavuvirta prosessien välillä.
 - Käytetään kuten tiedostoa (tai sokettia).
- Yhteisen **muistin** segmentit
 - Prosessi voi varata käyttöjärjestelmältä lohkon yhteistä muistia ja asettaa sille saantioikeudet.
 - Prosessi **kuvaa** (kytkee, map, attach) itse (tai toisen prosessin) varaa-mansa yhteisen muistin lohkon **omaan muistiavaruuteensa**, minkä jälkeen sitä voi käyttää kuten mitä tahansa dynaamisesti varattua muistialuetta.
 - Tehokas, käytännöllinen SMP tietokoneissa.
 - Vaatii huolellisen **synkronoinnin**.
 - Muistettava vapauttaa.

- **Semaforat** (opastin?)
 - Ei tiedonsiirtoon vaan **synkronointiin**.
 - Tarjoavat **atomiset** testaa/**odota&muuta** operaatiot.
 - Semafora on oleellisesti ei-negatiivinen kokonaisluku.
 - Luonti: *semget*(ID, ...) // järjestelmän laajuinen id
 - Alustus: *semctl*(1, ...) // alkuarvo, voi olla muutakin
 - Varaa semafora: *semop*(-1, ...) // vähentää arvoa kunhan semafora oli (on) positiivinen.
 - Vapauta semafora: *semop*(1, ...) // kasvattaa arvoa (joku muu voi nyt vähentää).
 - Syntaksit eivät ole aivan yo. mukaiset, kts. esimerkki.
 - Koska semaforat ovat KJ rajattu resurssi, on **muistettava vapauttaa**.

Säiepelit

- (Aluksi) yksi opiskelijoista on **suorittaja** (säie).
 - Prosessori ohjaa suorittajaa ("antaa virtaa").
- Muut opiskelijat ovat **metodeja**.
- Kukin **rivi on objekti**, kullakin objektilla **yhteistä tietoa** (rivin yhteinen paperilappu), rivin opiskelijat siis objektin "eri" metodeja.
- Kullakin metodilla on paikalliset automaattiset muuttujat (uusi paperilappu säikeelle aina kun metodin suoritus alkaa).
- Säie suorittaa metodeja ja kuljettaa mukanaan omia kutsuparametrejaan.

Toiminta (luokka ja metodi):

75

```
class rivi { 1
    int yhteinen = 0; // luokan (rivin) yhteinen jäsen 2
    int mina(int x) { // kaikkien metodi on sama 3
        int oma = x; 4
        int summa = oma + yhteinen; 5
        if (summa % 2 == 0) 6
            yhteinen = oma; 7
        if (summa >= 4) 8
            return summa; 9
        else 10
            return [kutsu naapuria omalla tai toisella rivillä] 11
                mina(summa); 12
        } 13
    } 14
```

Peräkkäinen suoritus:

- Yksi opiskelija suorittaa koodia jostain lähtien.

Samanaikainen suoritus yhdellä prosessorilla:

- Usea opiskelija suorittaa koodia, mutta vain yksi on kerrallaan aktiivinen, muut odottavat paikallaan.
- Käyttöjärjestelmä suorittaa vaihtoja (tai ohjelma jos jää odottamaan).

Samanaikainen suoritus riittävän **monella prosessorilla:**

- Usea opiskelija suorittaa koodia yhtä aikaa, vain lyhyitä katkoksia.

k samanaikaista suoritusta $p < k$ prosessorilla:

- Kukin p :stä prosessorista suorittaa aina hetken yhtä suorittajaa ja vaihtaa sitten johonkin odottavaan suorittajaan.

Mikä säie on?

- Javassa säie (suoritus) on myös objekti (luokka *Thread*).
- Säikeen suoritus ja muu toiminnallisuus voidaan (mutta ei ole pakko) sijoittaa samaan luokkaan.

```
class rivi {
    int yhteinen = 0;
    int mina(int x) {
        ...
    }
    void run() {
        ...
        mina(42);
        ...
    }
}
```

1
2
3
4
5
6 // säie (suorittaja) onkin täällä
7
8
9
10
11

-

Javan säikeet ja niiden synkronointi

⇒ Säie on peräkkäinen suorituskohta prosessissa (ohjelmassa).

- Monisäikeisessä ohjelmassa on useita samanaikaisesti suorituvia säikeitä (ohjelma-osoittimia).
- Kullakin säikeellä on oma pino ja siten paikalliset muuttujat (niistä ohjelmalohkoista joita se **itse on aloittanut**).

Säikeen ohjelmoiminen

- Säie toteutetaan **luokkana** jolla on metodi ***run(void)***.
 - Säikeen mahdolliset **parametrit annetaan konstruktorille**.
- Luokka joko laajentaa (extends) luokkaa ***Thread*** tai toteuttaa (implements) liittymän ***Runnable***.
- ***Thread***:n laajentaminen on hieman helpompaa, jos se on mahdollista.
 - Jos luokkamme laajentaa (perii) jo jotain toista luokkaa, tämä ei onnistu sillä Java ei salli moniperimystä.
 - Tällöin on toteutettava ***Runnable***.
 - ***Runnable*** sopii myös säikeiden suorittamiseen ***Executor***:illa.

- Toiminnot sijoitetaan metodiin *run()* ja siitä kutsuttuihin metodeihin.

```
public class OmaSaiet extends Thread {
    public OmaSaiet() {
        start();        // voidaan käynnistää muualtakin
    }
    public void run() {
        // säikeen "pääohjelma" alkaa
        // tähän ja tästä kutsuttaviin metodeihin
        // kirjoitetaan säikeen koko elämä ...
        // säie loppuu
    }
}
```

Runnable:n toteuttaminen

- Toiminnot sijoitetaan metodiin *run()* ja siitä kutsuttuihin metodeihin.
- Konstruktorissa luodaan säie (*Thread*) ja annetaan sille tämä *Runnable*.
- Säie käynnistetään *start()* -metodilla.

```
public class OmaSaieR implements Runnable {
    public OmaSaieR() {
        Thread s = new Thread(this);
        s.start();          // voidaan käynnistää muualtak.
    }
    public void run() {
        // säikeen "pääohjelma" alkaa
        // tähän ja tästä kutsuttaviin metodeihin
        // säikeen koko elämä ...
        // säie loppuu
    }
}
```

- Uusi säie **käynnistetään** metodilla *start()* jolloin se ryhtyy suorittamaan *run()* metodia.
 - Joko **konstruktorissa** (kts. yllä) tai **myöhemmin** (kunhan säikeeseen (objektiin) on **viite**).

```
(new OmaSaie()).start();           // lähtee heti           1
```

- tai:

```
OmaSaie saie = new OmaSaie();           2
```

```
...                                       3
```

```
saie.start();                                       4
```

```
...                                       5
```

- Jos säie aiotaan käynnistää kutsuvasta ohjelmanosasta, se ei toki saa käynnistää itseään (kuten yo. *OmaSaieT* ja *OmaSaieR*).

- Säie vuorottelee **suorituksessa** muiden säikeiden kanssa.
- Säie **pysähtyy väliaikaisesti** (not runnable) jos
 - se nukkuu (*sleep()*),
 - odottaa jotain tapahtumaa (*wait()*),
 - odottaa syötettä (tai tulostusta) (*read()* jne).
- Säie **pysähtyy lopullisesti** kun *run()* päättyy (kts. alla).
- **Säiettä (objektia) ei voi käynnistää uudelleen** (ts. metodia *.start():*ia ei saa kutsua uudestaan!)
 - Voit toki luoda **uuden ilmentymän** luokasta ja käynnistää sen.
- Säikeen tilan voi testata *Thread.getState()* metodilla.
- **Objekti jää olemaan ja käyttöön suorituksen jälkeenkin!**
 - Metodeja voi kutsua muista säikeistä.
 - Jäseniä voi käyttää (jos ne ovat julkisia).
- Toki olio (metodit, julkiset jäsenet) on käytössä normaalisti muista säikeistä **suorituksen aikanakin**.

Säikeiden suoritus

- Kun *run()* käynnistetään *Thread.start()*:lla, alkaa uusi suoritus.
- Suoritus voi **vaellella** ohjelmassa mihin tahansa (muihin luokkiin, muihin saman luokan ilmentymiin, jne).
 - Säikeen luokan metodeja (paitsi *run()*) voivat suorittaa muutkin säikeet.
- Samassa oliossa, **jopa saman objektin samassa metodissa** voi olla useita samanaikaisesti suorituvia säikeitä.
 - Kullakin säikeellä omat kopiot paikallisista muuttujista.
 - Yhteiset olion jäsenet.

⇒ Meidän on nyt ajatuksissamme erotettava **luokat/oliot/metodit ja niiden suoritus**.

- Selkeintä on ehkä erottaa säikeiden suorittaminen ja varsinainen ohjelman toiminnallisuus eri luokkiin.
- Samanaikaisia tehtäviä voidaan suorittaa myös **säiejoukon** (*Executor*, *thread pool*) avulla. Säiejoukko suorittaa jonosta tehtäviä (*Runnable*)

Mikä säie suorituu milloinkin?

⇒ Yksiprosessorikoneessa vain yksi säie suorituu kerrallaan.

- Moniprosessori(ydin)koneessa säikeet voivat suoritua rinnakkain **jos** JVM sitä tukee (Sun JVM:n käytös riippuu käytetystä käyttöjärjestelmän säiekirjastosta).
- Säikeillä on prioriteetit (1-10), korkeaprioriteettisin suorituskelpoinen säie suorituu.
- Useimmilla alustoilla samaprioriteettiset suorituskelpoiset säikeet suorituvat **aikajakoisesti**, mutta tätä ei taata!
- Säikeellä on metodi *yield()* jolla se voi (pitää) antaa suorituvuoron toiselle suorituskelpoiselle säikeelle.
- Jos säie on itsekäs (ei siis anna vuoroa muille), eikä järjestelmä tuo aikajakoa, yksi säie tukehduttaa muut.

Säikeiden lopettaminen

- Säie pysähtyy lopullisesti kun *run()* päättyy.
 - **Säikeen pitää tehdä se itse.**
 - Jos muiden säikeiden pitää pystyä kertomaan tälle säikeelle että "nyt olisi aika lopettaa", voidaan se välittää esimerkiksi jonkun säikeen jäsenen kautta. Säie tarkkailee jäsenen arvon muuttumista ajoittain.
 - Voidaan myös käyttää erillistä lippuluokkaa jota säikeet tarkkailevat, kts. *LopetusLippu.java*.
 - Ohjelman suorituksen ajan "taustalla" pyörivät säikeet voidaan määrittellä "demoneiksi", jolloin ne **pysähtyvät automaattisesti** kun kaikki muut säikeet ovat pysähtyneet.
 - *setDaemon(true)* ennen säikeen käynnistämistä.
 - Kts. *JonoTarkkailija.java*
 - *Thread.stop()* -metodilla säikeen on voinut "tappaa", mutta metodi on vanhentunut.
- **Säiettä ei saa käynnistää uudelleen** (kts. *SaieEsim.java*).

Keskeytykset (interrupt)

- **Nukkuvan** (sleep), **odottavan** (wait, join), tai **jonoa tai kanavaa odottavan** säikeen voi **herättää** ulkoa metodilla *interrupt()*, tällöin ko. säie saa poikkeuksen.

```
try {
    sleep(1000);
} catch (InterruptedException ie) {
    // herätetty kesken unien...
}
try {
    x = pysahtyvaJono.take(); // odottaa alkiota
} catch (InterruptedException ie) {
    // herätetty kesken datan odottelun ...
}
for (int i = 0; i < 1000000000; i++)
    // tätä ei pysäytetä keskeytyksellä
```

- **Suorituksessa oleva säie ei keskeydy**, mutta sen keskeytysstatus muutetaan, jolloin säie voi sen **itse tarkistaa** (*isInterrupted()*).
- Kts. tuottaja-kuluttaja -esimerkki.

- Future -konseptin esitti Halstead 1985 (Multilisp)
- Ajatuksena on käynnistää (aikaa vaativa) suoritus taustalle **uudeksi säikeeksi**, ja joskus myöhemmin ottaa **tulos käyttöön** (*get()*).
 - Tulos voidaan ottaa vasta **säikeen valmistumisen jälkeen**.
 - Valmistumista voidaan tarkkailla *isDone()* metodilla tai käyttämällä *get()*:n aikakatkaisua.
 - Future-suorituksen voi myös pyytää peruuttamaan (*cancel()*).
- <http://java.sun.com/javase/6/docs/api/java/util/concurrent/Future.html>
- <http://java.sun.com/javase/6/docs/api/java/util/concurrent/FutureTask.html>
 - Valmis paketti johon annetaan *Callable<V>* (tai *Runnable*)
 - *Callable<V>*: sisältää vain metodin *V call()*.
 - *Callable* annetaan yleensä anonyyminä luokkana, kts. käyttöesimerkki JavaAPI: *Future*.
- *SwingWorker* käyttää samaa (lisäksi siinä voidaan ottaa välituloksia).

Säikeiden synkronointi

⇒ Yleensä säikeet jakavat yhteisiä resursseja.

⇒ Yleensä säikeiden on tehtävä yhteistyötä tietyssä järjestyksessä.

Jokaisella objektilla (*Object*) on **lukko**

- Näin ollen mm. säikeen toteuttavalla objektilla on oma lukko.
- Kaikilla tietorakenteilla on oma lukkonsa.

⇒ Lukkoa voi pitää hallussaan vain yksi säie kerrallaan. Muut yrittäjät jäävät odottamaan.

- Lukkoa hallussapitävä säie voi lukon haltuun "uudestaan", sen on myös luovuttava siitä yhtä monta kertaa
 - Hyödyllinen esim. rekursiivisissa metodeissa.

- Säikeen synkronoinnin voi hoitaa monella tavalla, ja oikea tapa riippuu tilanteesta.
- **Helpoin** tapa on **metodin määre *synchronized*** joka ilmaisee, että vain yksi säie kerrallaan voi suorittaa **ko. objektin synkronoituja metodeja**.
 - Synkronoituja metodeja voi olla useita, jos joku niistä on suoritettavana, mitään niistä ei voi aloittaa (paitsi se säie joka on lukon alkuaan saanut, eli aloittanut ensimmäisen synkronoidun suorituksen).

```
public class kasvataVahenna {
    private int data;

    public synchronized int kasvata() {
        return ++data;
    }

    public synchronized int vahenna() {
        return --data;
    }
}
```

- Vain toinen metodeista suorituu kerrallaan.
- Konstruktoria ei voi synkronoida näin

Lauseloikon synkronointi olion (lukon) suhteen

```
synchronized( olio ) {  
    // vain yksi säie voi suorittaa ...  
}
```

- Edellinen metodin synkronoiminen on itse asiassa vain siistimpi versio varsinaisesta synkronoinnista jolla lauseloiko synkronoidaan olion suhteen.

```
void synkronoituMetodi() {  
    synchronized(this) {  
        // runko  
    } //  
}
```

- Lohkosynkronointi on joustavampi ja mahdollistaa **hienosäikeisemmän** lukituksen (jos metodit ovat pitkiä).
- Se myös mahdollistaa **synkronoinnin toisen olion suhteen**.

- *synchronized* on hankala jos lukitustarve on usean metodin laajuinen, tai sitä ei muuten voi **lohkoksi** pukea.
- *Lock* -rajapinnalla (*ReentrantLock*) voidaan lukita kriittisiä kohteita vielä joustavammin (ohjelmarakenteesta riippumattomasti).
<http://java.sun.com/javase/6/docs/api/java/util/concurrent/locks/ReentrantLock.html>
- Lock-lukkoa voidaan **yrittää varata** (*tryLock()*)
- *Lock*-lukko voidaan varata siten, että odottava säie voidaan tarvittaessa **keskeyttää** (*lockInterruptibly()*).
- *ReentrantLock* voi järjestää odottajat (fairness).
- *ReentrantLock*:ia laajentamalla voidaan tarkkailla (ja mm. keskeyttää) odottajia (*getQueueLength()*, *getQueuedThreads()*, ...).
- Lukitusta ja avausta ei ole sidottu mihinkään ohjelmarakenteeseen, joten on **aina huolehdittava, että lukko tulee avattua yhtä monta kertaa kuin se lukittiin**.
 - Erityisesti poikkeusten kanssa on oltava huolellinen.
 - Käytä *try... catch... finally* (lukon avaus *finally* -lohkossa).

Säikeiden väliset viestit (huomautukset)

- Olion (*Object*) *wait()* metodilla säie voi luovuttaa omistavansa olion lukon (varattu *synchronized* -lohkolla) ja odottaa, että joku toinen säie viestii sille ko. olion *notify()* tai *notifyAll()* -metodilla (tai aikakatkaistu kuluu umpeen tai se keskeytetään).
 - Kun jokin näistä tapahtuu, säie odottaa kunnes saa lukon takaisin ja jatkaa suoritusta.
 - Bonus: *wait()*:ssä odottavan säikeen voi keskeyttää *interrupt()*:lla.
- Säie voi odottaa toisen säikeen valmistumista *join()* -metodilla.

- Esim. **tehokas**, keskeytettävä jonon käyttö:

```

ConcurrentLinkedQueue jono; 1
// kuluttaja 2
try { 3
    synchronized(jono) { // varataan lukko 4
        while ((alkio = jono.poll()) == null) 5
            jono.wait(); // vapautetaan lukko väliaik, odotetaan 6
    } // vapautetaan lukko 7
} catch (InterruptedException ie) { ... } 8

// tuottaja 9
synchronized(jono) { // varataan lukko 10
    jono.offer(x); 11
    jono.notify(); // kuluttajan herätys 12
} // vapautetaan lukko 13

```

- Toisto tarvitaan koska *wait()* saattaa keskeytyä muutenkin kuin tuottajan laittaessa alkio sinne.

- *Timer* -luokan ilmentymillä voidaan ajastaa *TimerTask*:n aliluokan ilmentymiä (**säikeitä**) käynnistymään (*run()*) annetun ajan kuluttua, tarvittaessa **toistuvasti** halutulla välillä (*fixed-delay*) tai halutulla syklillä (*fixed-rate*).
- <http://java.sun.com/javase/6/docs/api/java/util/Timer.html>
- Toistossa olio säilyy, mutta ***run()* suoritetaan useasti**.
- Suoritussäikeitä on oikeasti vain yksi (jota koordinoi *Timer*), joten *TimerTask*:n aliluokkien ***run()* metodien olisi oltava lyhyitä**.
 - Jollei se lopu ajoissa, muut saman *Timer*-olion ajastuksessa olevat säikeet eivät pääse suoritukseen.
- Varsinainen ajastinsäie (*Timer*) ei pääty automaattisesti, vaan se on lopetettava (*.cancel()*) jotta ohjelma loppuisi.
- Kts. *TimerEsimerkki.java*, *TimerEsimerkki2.java*.

- <http://java.sun.com/javase/7/docs/api/java/util/concurrent/atomic/package-summary.html>
- Pakkauksessa *java.util.concurrent.atomic* on joukko yksinkertaisia tietotyyppejä joiden käsittely on atomista, ts. joko **operaatio tapahtuu kokonaan kerralla, tai sitä ei tapahdu ollenkaan**.
- Näillä voidaan usein korvata erillisen synkronoinnin käyttö.
- Näitä voi myös käyttää tavallisten muuttamattomien (immutable) olioiden sijasta kun halutaan välittää viite yksinkertaista tyyppiä olevaan olioon ja muuttaa sen arvoa.

Tietorakenteita Javan säikeidenväliseen kommunikatioon

- *java.util.concurrent*
- <http://java.sun.com/javase/7/docs/api/java/util/concurrent/package-summary.html>
- Kts. *Tuottaja.java*, *Kuluttaja.java*, *SyncKT.java*, *BuffKT.java*

⇒ Kukin Javan vakiokirjaston luokka (ainkin kokoelmat) on dokumentoitu joko synkronoiduiksi tai synkronoimattomiksi (engl. Thread-safe, not Thread-safe), **muista tarkistaa**.

Erityisesti tuottaja-kuluttaja-arkkitehtuuriin

- Jono, rajapinta *Queue*<E>, ja sen toteutukset.
 - *offer()*, *peek()*, *poll()*, *isEmpty()*
 - Tämän toteuttavista esim. *LinkedList*<E> ei ole säieturvallinen.
- Rajapinta *BlockingQueue*<E>
 - *E take()* ottaa alkion jonosta, odottaa jos jono tyhjä
 - *put(E o)* laittaa alkion jonoon, odottaa jos jono täysi.
 - *boolean offer(E o, long timeout, TimeUnit unit)*
 - *E poll(long timeout, TimeUnit unit)* tai *poll()*
 - **Toteutuksia:** *ArrayBlockingQueue*, *DelayQueue*, *LinkedBlockingQueue*, *PriorityBlockingQueue*, *SynchronousQueue*
- *ConcurrentLinkedQueue*<E>
 - Rajattoman pituinen jono
 - Operaatiot vakioaikaisia, eivät odota vaan palauttavat null jos jono tyhjä (paitsi *size()*!)

- *SynchronousQueue*<E>
 - *put* ja *get* -operaatioiden tahdistukseen
 - *put()* odottaa kunnes vastaava *get()* tehdään
 - *get()* odottaa kunnes vastaava *put()* tehdään
 - ei *peek()* operaatiota
- *ArrayBlockingQueue*<E> – rajattu kapasiteetti.
- *PriorityBlockingQueue*<E> – Prioriteettijono.
- *DelayQueue*<E extends Delayed>
 - Kullakin alkiolla on viive, jonka jälkeen se voidaan ottaa jonosta. Pisimpään viiveen ylittäneet ensimmäisenä.
- Kts. Tuottaja-kuluttaja esimerkit.

Muiden kokoelmien synkronointi

- Oletusarvoisesti kokoelmat eivät ole säieturvallisia (synkronoituja).
- Synkronoinnin voi ottaa käyttöön *java.util.Collections* -luokan staattisilla apumetodeilla:
 - *Collection<T> synchronizedCollection(Collection<T> c)*
 - Vastaavasti *List, Set, SortedSet, Map, SortedMap*.
- Kokoelman kaikki käytöt on kuitenkin synkronoitava eksplisiittisesti.
- Esim.

```
List l = Collections.synchronizedList(new LinkedList());  
...  
synchronized(l) {  
    Iterator i = l.iterator();  
    while (i.hasNext())  
        ...;  
}
```

Pikaohjeet samanaikaisohjelmajalle

- Piirrä kuva säikeistä ja niiden yhteyksistä.
- Älä perusta säikeitä turhaan.
- Ole aivan varma mitkä säikeet sinulla on käynnissä missäkin vaiheessa.
- Minimoi yhteisten resurssien määrä.
- Minimoi olioiden/luokkien välinen "ristiliikenne".
- Kontrolloi pääsy yhteisiin resursseihin.
- Varmista käyttämiesi kirjastojen monisäie-kelpoisuus.
- Älä lukitse turhaan/liian pitkäksi aikaa.
- Älä pollaa jatkuvasti (busy-loop).
- Varmista, että kukin resurssin varannut säie myös vapauttaa sen.
- Varmista, ettei mikään säie nälkiinny (odota resurssia ikuisesti).
- Testaa useita kertoja (joskaan se ei todista mitään).
- Pöytätestaa kuvan kanssa, varmista kuvan oikeellisuus.

Luku 3

Hajautettujen järjestelmien mallit

Täsmällisempi terminologia, roolit

Rakennemallit (architectual models) (s. 103)

Vuorovaikutusmallit (interaction) (s. 118)

Vikamallit (failure) (s. 150)

Turvallisuusmallit (security) (s. 157)

HJ:ssä käyttäjät **jakavat resursseja** (tietoja, palveluja).

- Kukin **resurssi** on yleensä "koteloitu" johonkin järjestelmän tietokoneeseen (**prosessiin**) ja muut tietokoneet (prosessit) käyttävät sitä kommunikoiden ko. tietokoneen kanssa.
 - Kommunikaatio hoituu viestein.
 - Yksi prosessi aloittaa kommunikaation lähettämällä pyynnön toiselle prosessille (jonka on oltava "**kuuntelulla**").
 - Pynnön vastaanottava prosessi yleensä vastaa pyyntöön.
- Resurssia ylläpitävää prosessia (ohjelmaa) sanotaan **resurssimanageriksi** (-hallitsin) (resource manager).
 - Se tarjoaa **kommunikointirajapinnan** jolla käyttäjät pääsevät käsiksi resurssiin.
 - Resurssimanageri voidaan ajatella **objektina oliomallissa**.
 - Mutta silti objekti koteloidaan **johonkin prosessiin** (johon viitataan objektina toisesta prosessista).

Rakennemallit (architectual models)

⇒ Miten vastuut jaetaan eri komponenttien kesken ja miten yhteydet järjestetään?

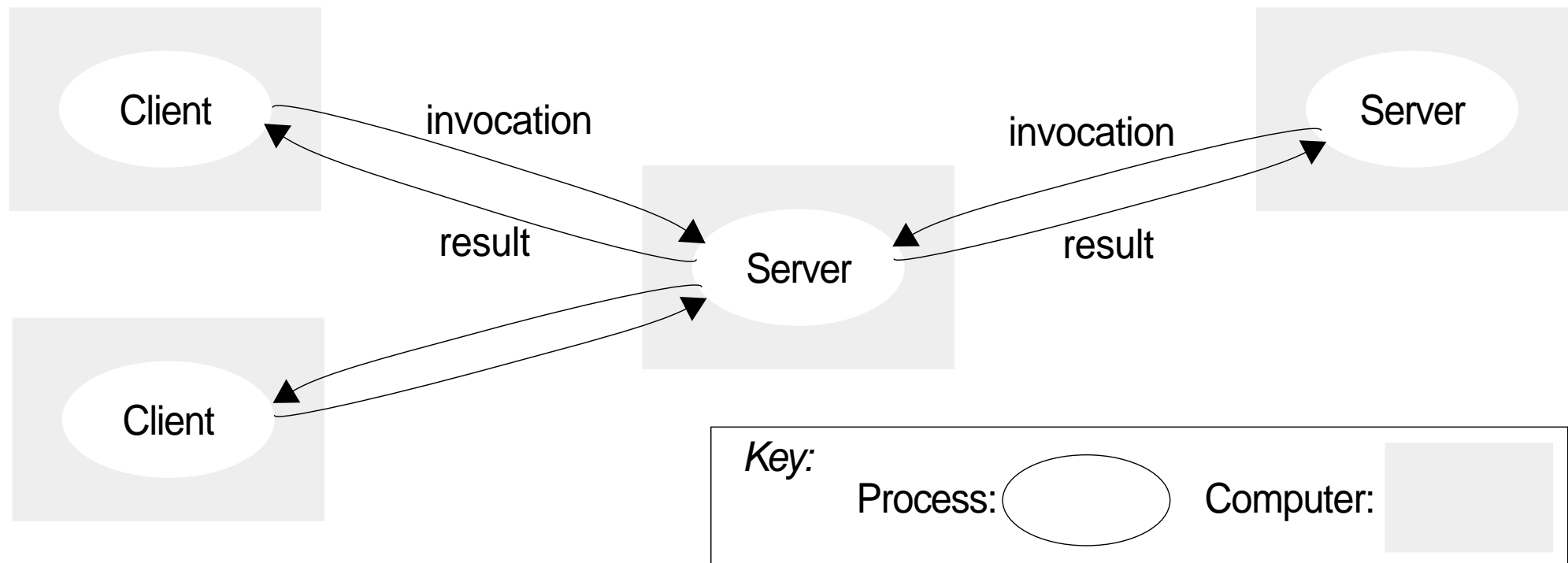
- “Kommunikaatiokuviot”

- Asiakas-palvelin (client-server) (s. 104)
- Välityspalvelin (proxy server) (s. 107)
- Liikkuva agentti (mobile agents) (s. 110)
- Verkkotietokone (network computer) (s. 111)
- Ohut asiakas (thin client) (s. 112).
- Vertaisverkko (peer processes, peer-to-peer, p2p) (s. 113)

Asiakas-palvelin (client-server)

⇒ Järjestelmässä on joukko palveluita tarjoavia prosesseja (palvelimia) ja niitä käyttäviä asiakkaita.

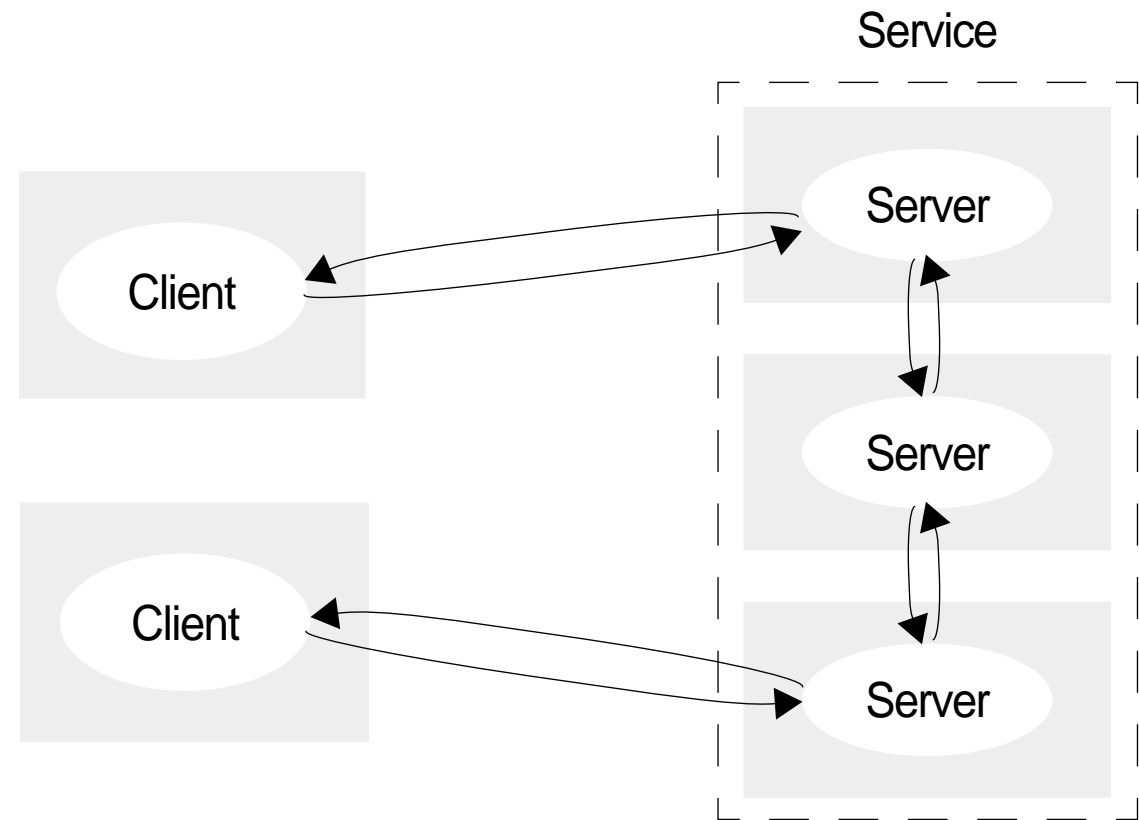
- Asiakas-palvelin malli perustuu yleensä **pyyntö/vastaus** -protokollaan toteutettuna **lähetä/vastanota** -mekanismeilla (tai RPC/RMI).
 - **Asiakas pyytää palvelua** lähettämällä viestin.
 - Palvelin tekee työn ja vastaa tuloksella (tai virhekoodilla).



- Palvelin voi vuorostaan pyytää palvelua toiselta palvelimelta, ja siten tässä suhteessa **toimia asiakkaana**.

Palvelun voi tarjota useampikin palvelin

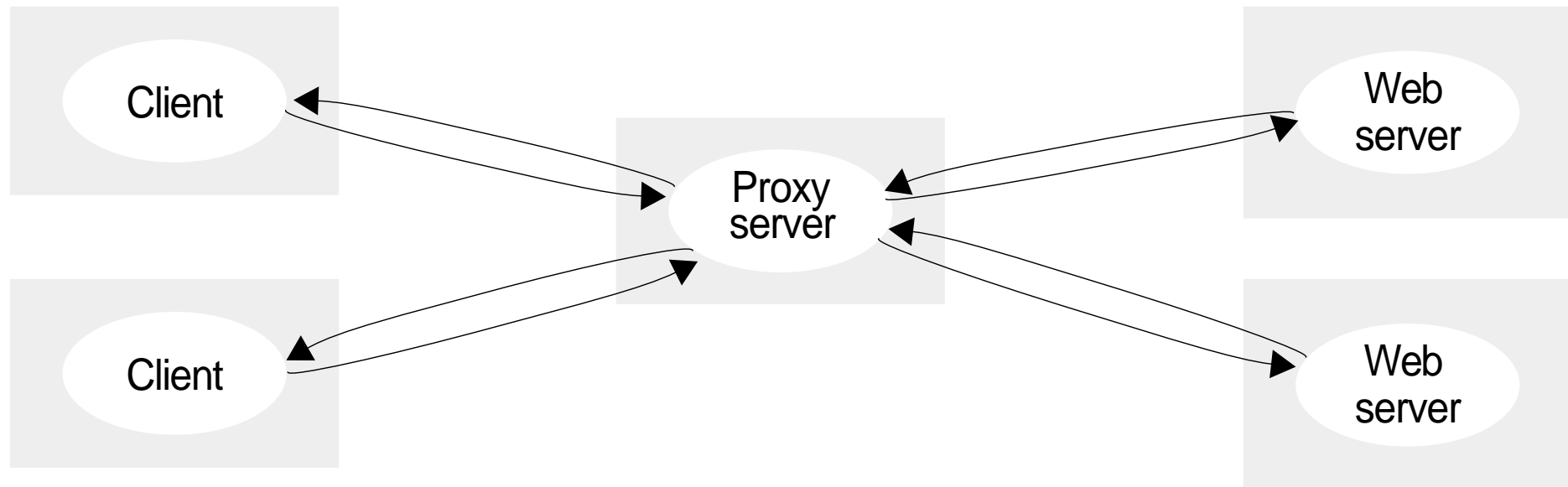
- Palvelun osat (objektit) voivat jakautua useaan eri palvelimeen (prosessiin).
 - Esim. **looginen** www-sivusto voi jakautua usealle tietokoneelle (joilla voi olla eri osoitteet ja nimet).
 - Jako voi olla asiakkaalle näkyvä tai näkymätön.



- Palvelun objekteja voidaan myös **monistaa** usealle palvelimelle.
 - Esim. sama www-sivusto voidaan monistaa ja eri asiakkaat ohjataan (**mahdollisimman aikaisessa vaiheessa**) eri palvelimille
 - Eri palvelimet voivat myös käsitellä **erityyppiset palvelupyynnöt** tai asiakkaat.
 - Saantirajoitukset, protokollaerot, kts. välityspalvelin.
 - Voidaan myös parantaa **skaalattavuutta** jos palvelu on laskentaintensiivinen.
 - Esim. dynaaminen kuvangenerointi.

Välityspalvelin (proxy server)

⇒ Välityspalvelin tarjoaa (ja säilyttää) kopioita (tai versioita) **muiden palvelimien hallinnoimista palveluista**.



- WWW-välimuistipalvelin (cache).
- Välityspalvelin voi olla **asiakkaalle** yksityinen, jaettu usealle asiakkaalle, tai olla vastaavasti **lähempänä palvelinta**.

- Tavoite on lisätä **suorituskykyä** ja **saavutettavuutta**, sekä **vähentää verk-** 108 **koliikennettä**.
- Välityspalvelimia voidaan käyttää myös eräänlaisena väliohjelmistona.
 - Yhdenmukaistetaan **asiakkaan näkymä** (esim. HTML-WML -välityspalvelin) tai yhdenmukaistetaan **palvelimen rajapinta** (eri välityspalvelin erilaisille asiakkaille).
 - Samoin välityspalvelinta voidaan käyttää kontrolloimaan asiakkaan, palvelimen tai välisolmun liikennettä (esim. sensurointi, laskutus).
- **Palvelimen päässä** välityspalvelinta voidaan käyttää keventämään varsinaisen palvelimen kuormaa useimmin pyydettyjen sivujen osalta (esim. uutispalvelun tuoreimmat uutiset).
 - Välityspalvelin voi myös sijaita keskeimmällä verkossa, periaatteessa missä tahansa asiakkaan ja palvelimen välillä (tkt, JoY, Funet, NordUNET, ...).

- Erityisesti läsnäolevissa (ubiquitous) järjestelmissä uusien sovellusten ja **palvelujen lisääminen** on oltava sujuvaa.
- Samoin suurissa ei-IT organisaatioissa järjestelmien tulisi olla keskitetysti ylläpidettävissä.
- Seuraavat arkkitehtuurit on kehitetty nimenomaan sovellusten lisäämisen ja **päivittämisen helpottamiseksi**.

Liikkuvat ohjelmat (mobile code)

- Palvelin lähettää asiakkaalle ohjelman joka suorittaa sen paikallisesti.
- Palvelin konfiguroi sovelluksen valmiiksi.
- Paikallisesti asiakkaassa/-na suoritettava ohjelma voi ylläpitää automaattisesti yhteyttä alkuperäiseen palvelimeen (tai muualle).
- Vaatii yhteisen alustan tai virtuaalikoneen.
- Käytetään yleensä selainalustalla.

Liikkuva agentti (mobile agents)

- Koodin lisäksi, myös ajonaikainen tieto siirtyy.
- Saapumisen jälkeen käyttää myös paikallisia resursseja, voi ottaa tietoja mukaansa ja palata palvelimelle (tai edetä toiselle asiakkaalle).
- Käyttökelpoinen esim. hajautetuissa simulaatioissa.
- Harvoin käytetty, potentiaalinen turvariski (kuten liikkuva koodikin).

Verkkotietokone (network computer)

- Käyttäj., sovellukset ja käyttäjän tiedostot **talletetaan palvelimella**.
- Käynnistettäessä (sisäänkirjautuessa) käyttäj., sovellukset ja (viitteet) käyttäjän tiedot **ladataan palvelimelta paikalliselle** tietokoneelle.
- Sovellukset **suoritetaan paikallisesti**, tiedot säilytetään palvelimella.
- Tukee käyttäjien **siirtymistä**.
- Asiakastietokoneet identtisiä.
- Vähentää levytilan tarvetta, helpottaa ohjelmistopäivityksiä.
- Paikallista massamuistia voidaan käyttää **välimuistina** (käynnistettäessä vain tarkistetaan josko tiedostot ovat muuttuneet palvelimella).
- Varmuuskopiointi (ja vikasietoisuus) suoraan palvelimelta.
- **Verkkoviiveet eivät** välttämättä näy käyttäjälle (ennen kuin tarvitaan massamuistia).
- Paikallinen massamuisti "välimuistina".
- Asiakkaan vikaantuminen vie vain tallentamattoman tiedon.

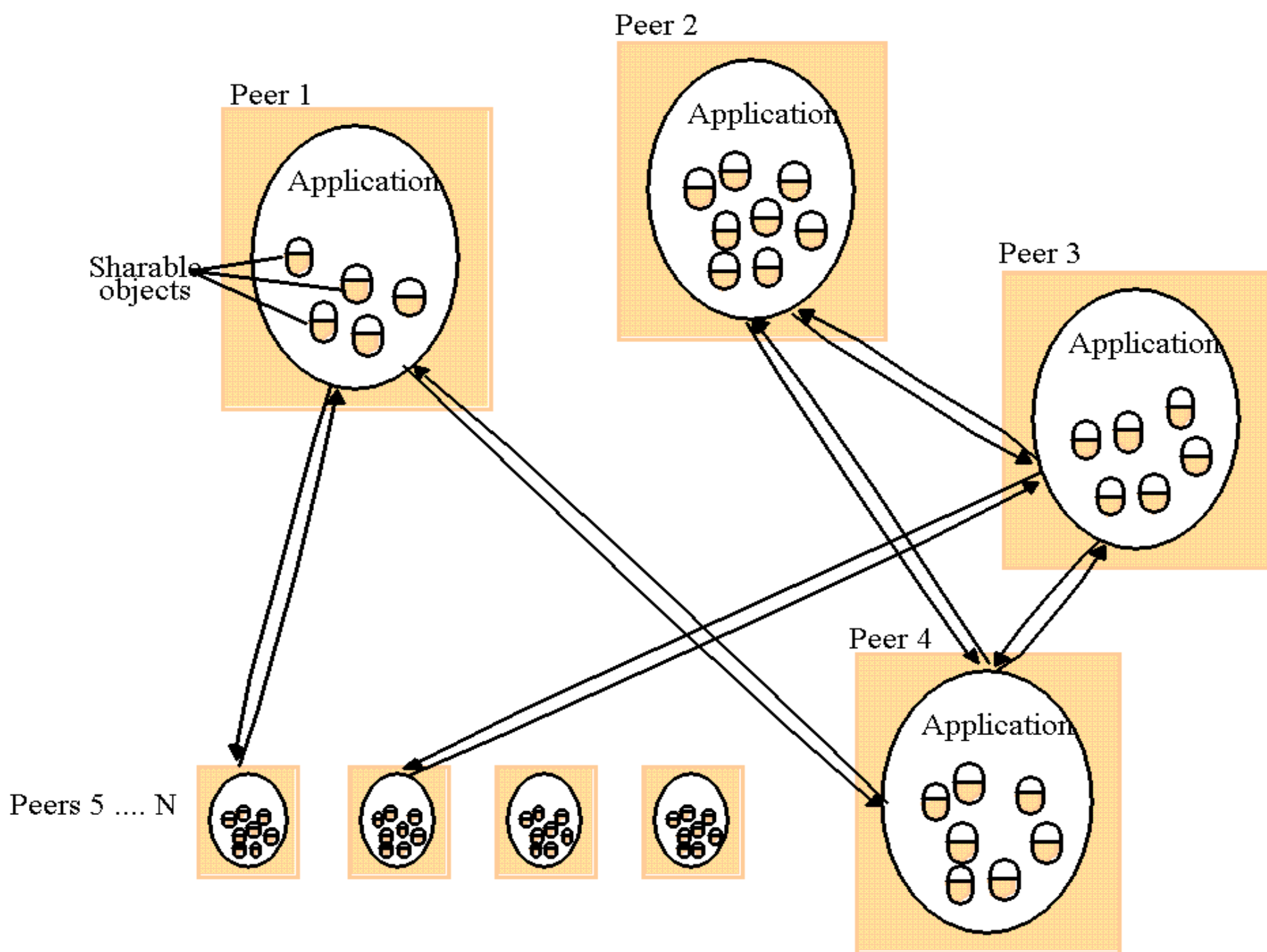
Ohut asiakas (thin client)

- Sovellus ja sen käyttöjärjestelmä **suoritetaan palvelimella**.
- Asiakas tarjoaa vain **käyttöliittymän** (ikkunointi, tai vain tekstipäätte)
- Hyödyllinen erityisesti jos sovellus on massiivinen ja se voidaan jakaa palvelimella. Samoin jos sovellus vaatii laskentavoimaa vain hetkittäin.
- Palvelin **aikajakaa laskentavoimansa** asiakkaille.
 - 10-50 aktiivista käyttäjää / prosessori; liki rajatta ei-aktiivisia.
- Asiakastietokoneet **eivät vanhene** "koskaan".
- **Verkkoviiveet näkyvät** välittömästi käyttäjälle.
- Asiakkaan/verkon vikaantuminen ei hävitä mitään.
- Esim. X11, Citrix, rdesktop, VNC, SUN Ray.
- **Web-liittymä** sovellukseen toimii myös kuten ohut asiakas, joskin www-selaimia joudutaan päivittämään jos niitä käytetään muuhunkin.
 - Näin ollen käyttöjärjestelmä ja www-selain joudutaan toteuttamaan verkkotietokoneena.

Vertaisverkko (peer processes, peer-to-peer, p2p)

⇒ Kukin prosessi toimii samanlaisessa roolissa.

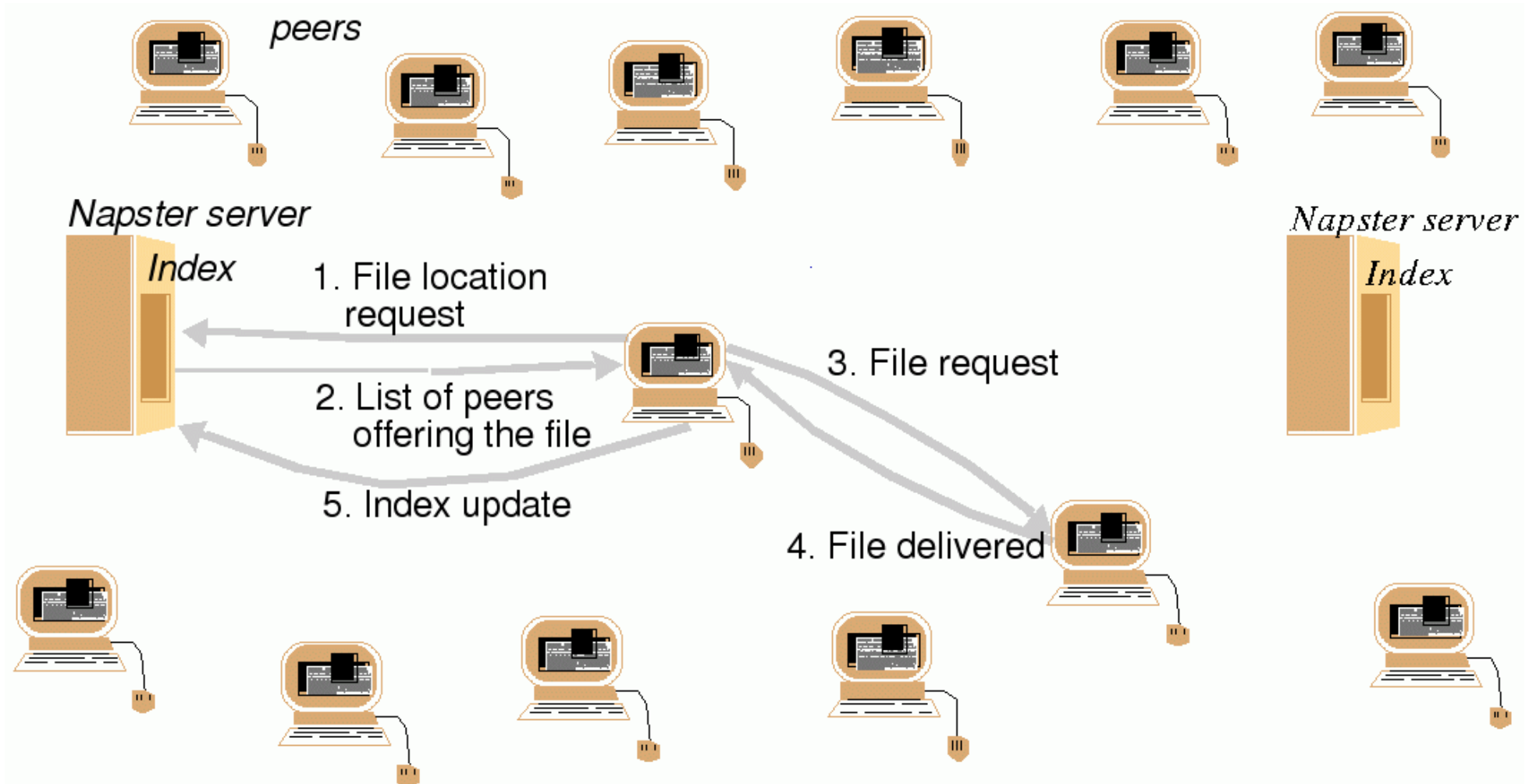
- Prosessit toimivat **ilman jakoa** asiakkaisiin ja palvelimiin (kukin voi toimia "asiakkaana" ja "palvelimena").
- Geneerisempi ja joustavampi kuin asiakas-palvelin.
- Kommunikaatiokuva riippuu sovelluksesta ja algoritmista, yleensä jossain määrin hierarkkinen.
- Erittäin skaalautuva jos saadaan aidosti pysymään vertaisverkkona.
- Koordinointi (luottamus) ja lupapolitiikat erittäin tärkeitä (ja vaikeita).
- Esim., usenet news, SMTP, Napster & al, jne.



⇒ Asiakas-palvelin -mallissa asiakkaan riittää tietää palvelimen osoite, palvelin tarjoaa kaikki palvelut.

- Vertaisverkossa jokaisen jäsenen ei kannata (voi) tuntea kaikkia jäseniä, saati niiden palveluita.
- Jaotellaan palvelun löytäminen ja saaminen:
 - **Havainto**palveluihin (discovery) – löydetään **muita jäseniä**, erityisesti hakemistopalveluita.
 - **Hakemisto**palveluihin (directory) – löydetään **haluttua sisältöä**.
 - **Sisältö**palveluihin (content) – **suoritetaan** (siirretään) palvelua.
- **Puhtaassa vertaisverkossa** kaikki palvelut on hajautettu – uuden käyttäjän on löydettävä joku toinen solmu, jota kautta hän pääsee käsiksi muihin solmuihin.

- Hieman keskitetyimmässä ratkaisussa **havaintopalvelu** on keskitetty (ja replikoitu) ja tunnettu (osoite asiakasohjelman mukana tms.) (esim. Kazaa), palvelimelta saa joukon hakemistosolmujen yhteystietoja.
- Edelleen keskitetyimmässä mallissa **hakemistot** ovat yhdellä (tai useammalla monistetulla) palvelimella (esim. Napster).



Mikä tekee (musiikki)tiedostonvaihdosta/jakelusta hyvän sovelluksen vertaisverkolle?

- Tiedostot (sisällöt) eivät muutu.
- Yksittäisen tiedon jatkuvasta saatavuudesta ei tarvitse olla varmuutta.
- Hakemistojen ei tarvitse olla (aivan) yhdenpitäviä.

Muita piirteitä

- **Kuormantasaus**
 - Napster: haetaan tiedosto lähimmältä solmulta.
 - BitTorrent: Tiedosto tallennetaan pienehköinä palasina ja haetaan **eri palat eri solmuilta** (miksi?).
- Tiedostojen tunnistaminen ja eheys
 - Tiedostojen tunnisteet varustetaan **tiivisteellä** (hash).
- Anonymiteetti, tietämättömyys
 - Freenet: tiedostot säilytetään solmujen levyllä salattuna.

Vuorovaikutusmallit (interaction)

⇒ Miten käsittelemme aikaa, aikarajoja, viestinvälitysaikaa?

- Synkroninen hajautettu järjestelmä (s. 119)
- Asynkroninen hajautettu järjestelmä (s. 121)
- Looginen aika (s. 132)

Ominaisuudet:

- Jokaiselle prosessin toiminnolle asetetaan ajankäytön **ylä-** ja **alaraja**.
- Lähetetyt **viestit** vastaanotetaan tunnetun aikarajan kuluessa.
- Paikallisten kellojen **käyntipoikkeamat** tunnetaan (eron raja).

⇒ Ala- ja ylärajat voivat olla joko varsin **väljiä** tai varsin **vaikeita** saavuttaa luotettavasti.

Synkronisuudesta saavutettavat edut:

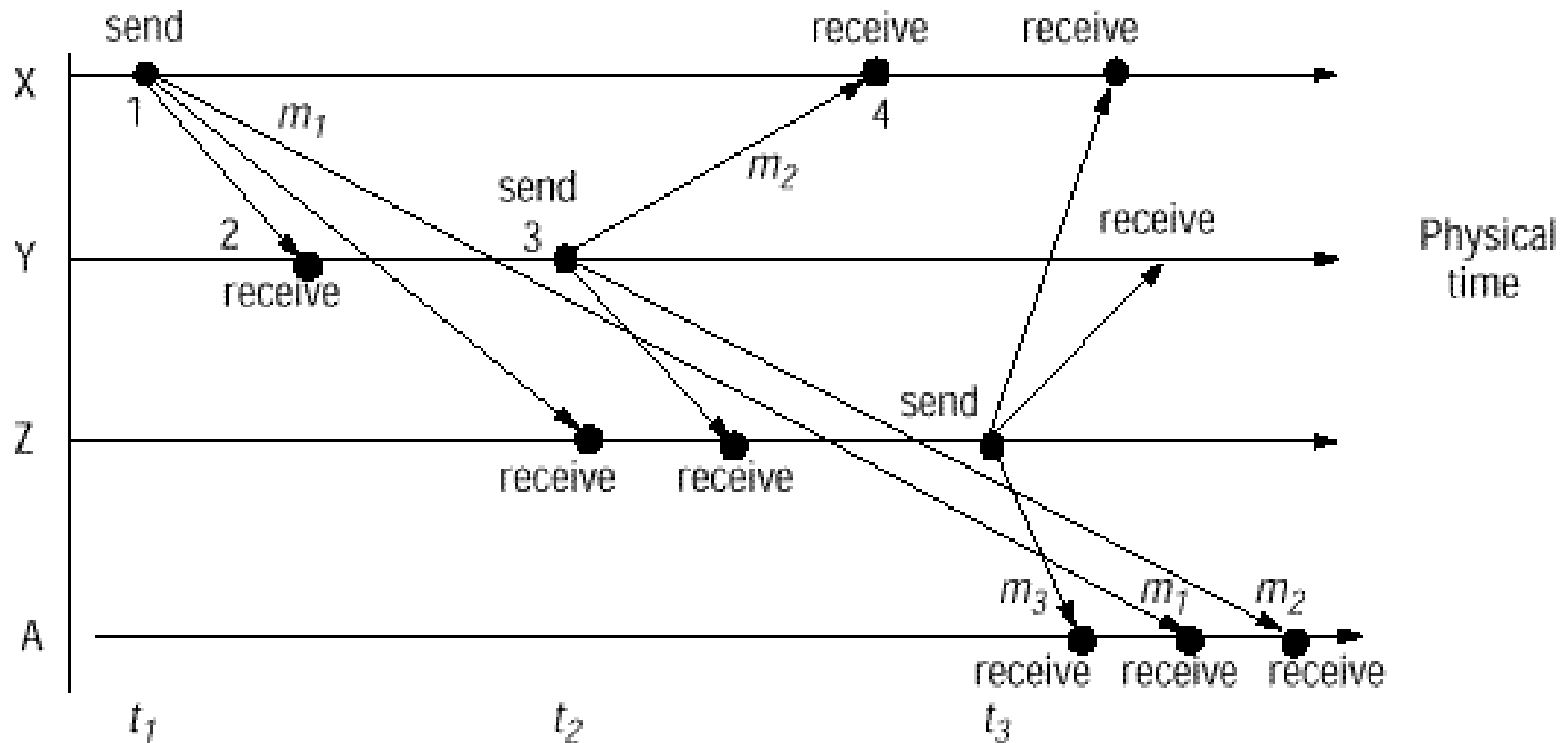
- Synkronisessa järjestelmässä meillä on käytössä **globaalin ajan** käsite (jonka (**epä**)**tarkkuus** riippuu synkronointialgoritmista ja kellojen jättämisen/edistämistarkkuudesta).
- Synkronisessa järjestelmässä on mahdollista ja turvallista käyttää **aikakatkaisua** (timeout) **vikojen** havaitsemiseen.
- Aikarajoista lipsuminen tulkitaan vikaantumiseksi.
- Ylä- ja alarajan **erotus** helposti johtaa **resurssien tuhlaukseen**.

⇒ **Synkronisten hajautettujen järjestelmien tekeminen on vaikeaa.**

Asynkroninen hajautettu järjestelmä

⇒ Useimmat hajautetut järjestelmät ovat ei-synkronisia, erityisesti IP-protokolla on asynkroninen.

- Ei rajoja prosessin suoritusajalle, viestinvälitykselle tai kellojen keskinäiselle tarkkuudelle.



Entä jos uskottelisimme asynkronisen järjestelmän olevan synkroninen?

- **Synkronoidaan kellot** ajoittain, väljä(hkö)t aikarajat, **aikaleima** kaikissa viesteissä.
- Myöhästyneet viestit tulkitaan kadonneiksi (vikaantumisiksi).
- **Todellisia vikoja ei** kuitenkaan voida havaita luotettavasti.
- Väljät rajat yleensä tekevät järjestelmästä tehottoman.
- Tätä voidaan toki jossain määrin käyttää.
 - Myöhästymisen havaitseminen yhtäpitävästi on tosin hankalaa.

(Automaattinen säännöllinen) kellojen **synkronointi**.

- Tietokoneiden kellot synkronoidaan toistensa kanssa ja synkronoinnin (epä)tarkkuus tunnetaan.
 - **Synkronointiviestien kiertoaika** (Round Trip Time, RTT) ja käyttöjärjestelmän keskeytysten tarkkuus rajoittavat tarkkuutta.
 - Käytetään useita viestejä ja tilastollisia menetelmiä tarkempaan synkronointiin, esim., Christian:n algoritmi:
 - Lähetetään viesti, paluuviestissä saadaan palvelimen aikaleima.
 - Lisätään saatuun aikaan arvioitu välitys- ja käsittelyaika $((RTT - I) / 2, I = \text{käsittelyaika})$
 - Toistetaan useasti, **hyödynnetään nopeimmat** viestit.
 - Ei koskaan korjata taaksepäin, vaan hidastetaan kelloa.
 - Paikallisverkossa päästään (kymmenien) millisekuntien tarkkuuteen (jos käyttöjärjestelmä sitä tukee).
 - Paikallisten kellojen ryömiminen yritetään mitata ja tehdä siitä virhearviointi. Toivotaan ryömimisen pysyvän annetussa rajassa.
 - **Tämän tarkkuuden puitteissa** voimme koordinoida toimintoja.

- The Network Time Protocol (**NTP**) [Mills 1995].
 - Tilastollisia menetelmiä.
 - Useita palvelimia (**hierarkia**).
 - UDP-viestit (ja monilähetys).
 - Paikallisten tietokoneiden aikavirhe mitataan ja **kellon käyntiä säädetään** vastaavasti.
 - **Dynaaminen synkronointiväli** kellojen tarkkuuden mukaan.
 - Jopa (parin) **millisekunnin tarkkuus** paikallisverkossa (joskin sitä voi olla vaikea hyödyntää ei-reaaliaikaisella käyttöjärjestelmällä).
- Hitaammassa verkossa ja lähemmäs UCT:ta päästään vastaanottamalla **atomikellojen aikasignaalia** GPS-satelliiteista tai maanpäällisestä radioverkosta, jopa μ s tarkkuus jos käyttöjärjestelmä tukee sitä.

Johtopäätökset asynkronisuudesta:

- Asynkronisessa (hajautetussa) järjestelmässä ei ole globaalia kelloa, järjestys voidaan määrätä ainoastaan loogisilla kelloilla (sivu 128).
- Asynkroniset järjestelmät ovat **ennustamattomia** ajoitusten suhteen.
- Teoriassa aikakatkaisua ei voida käyttää.
- **Käytännössä aikakatkaisua** voidaan (on pakko) käyttää. On muistettava suodattaa moninkertaiset (myöhästyneet) viestit, suoritukset, jne.

⇒ Asynkronisia järjestelmiä käytetään käytännössä.

⇒ Synkronisia ja asynkronisia toimintoja usein käytetään samassa järjestelmässä.

⇒ Käytetään loogisia kelloja toimintojen järjestämiseen.

Synkronointiesimerkki: Pepperland divisions (CDK3) 126

- Kaksi puolustavaan armeijan osastoa (**sininen** ja **vihreä**).
 - Leiriytyneenä lähekkäin, mutta eivät näe toisiaan.
 - Kommunikoivat lähetein.
 - Ei yhteistä kelloa.
- **Keltainen** vihollinen.
 - Vahvempi kuin kumpikaan puolustavista osastoista, mutta heikompi kuin nämä yhteensä.
 - **Samanaikainen** hyökkäys (sekä **sininen**, että **vihreä**) voittaisi **keltaisen** vihollisen.

⇒ Miten sinisen ja vihreän osaston pitäisi päättää kumpi johtaa hyökkäystä ja miten hyökkäys ajoitetaan?

- Kumpi?
 - Lähetetään omien taistelijoiden lukumäärä toiselle, vahvempi johtaa hyökkäyksen (tai sininen jos yhtä vahvat).
 - Toimii jopa asynkronisesti.

- Koska? **Synkronisessa maailmassa** (*min..max* viestinvälitysaika):
 - Johtaja lähettää “Hyökkäys!”, odottaa *min* minuuttia, ja hyökkää.
 - Tukiosasto odottaa yhden minuutin, sitten hyökkää.
 - Ero on $1..max-min+1$ minuuttia.
- **Asynkroninen** maailma (rajoittamaton viestiaika, ei yhteistä kelloa):
 - **Ei ole keinoa** sopia aikaa.
 - **Yhteinen kello** riittää: auringon noustessa:
 - Sattuuko päivä oikein?
 - Täydenkuun jälkeen seitsemäs auringonnousu:
 - Ok, jos viestinvälitykselle annetaan **aikaraja** (n. 1 kk).

⇒ Koska jokaisessa tietokoneella hajautetussa järjestelmässä on oma kello, ei ole olemassa luotettavaa globaalia fyysistä aikaa.

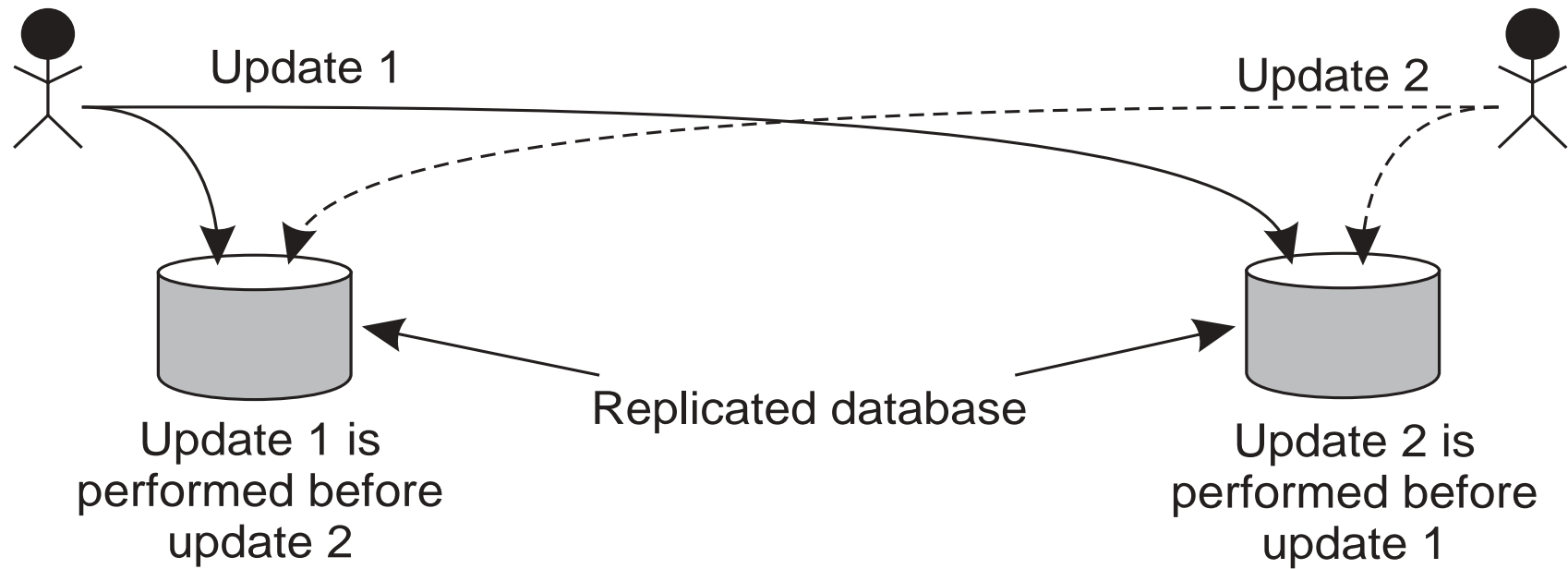
- Jokaisen tietokoneen kellokiteet käyvät hieman eri nopeuksilla ja kellot ryömivät (drift) (edistävät/jätättävät) toisiinsa nähden.
- Tarkka **synkronointi ei ole mahdollista** hajautetussa ympäristössä.
 - Synkronointiviestien välitysaika vaihtelee.

Joskus fyysinen aika on lähtökohta:

- Aikalaukaisut: toimenpiteen pitäisi tapahtua ennalta määrätyllä hetkellä.
 - Esim. reaaliaikajärjestelmissä.

- **Hajautetun tietokannan eheyden säilyttäminen** perustuu usein muutosten ajanhetkiin ja niiden järjestykseen

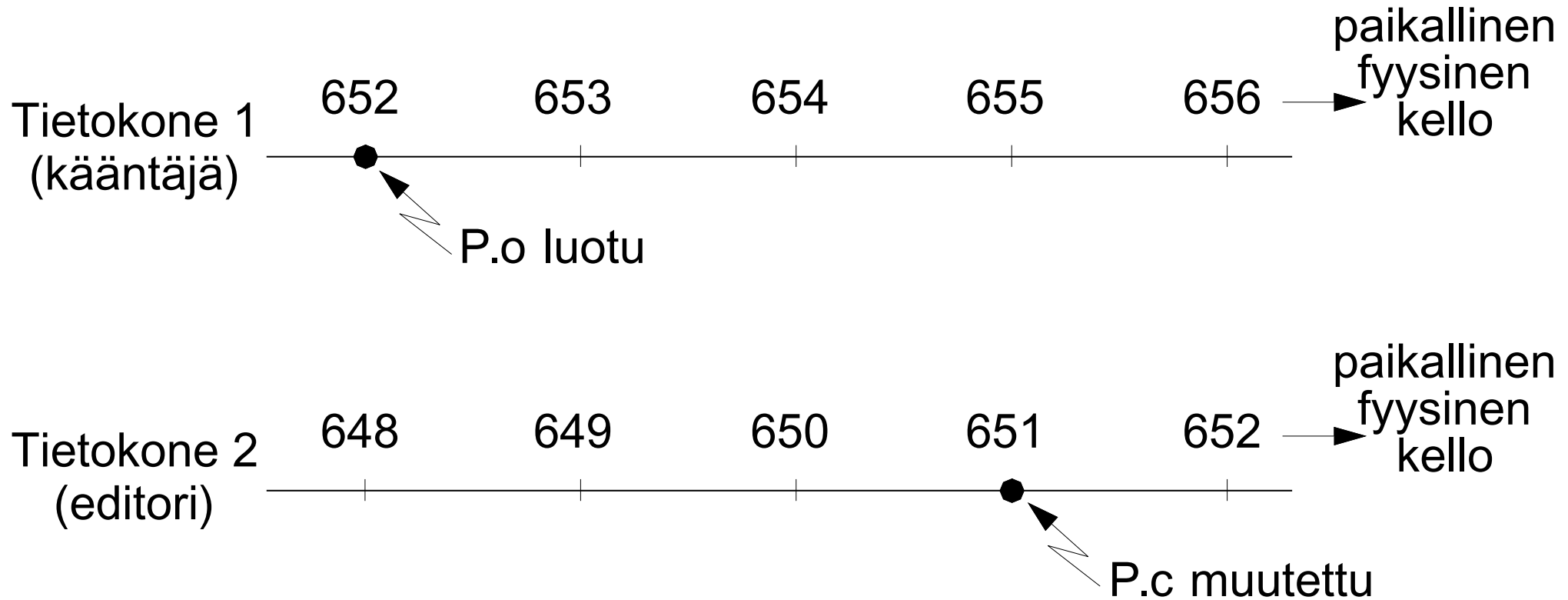
- Esim. kahdennetun tietokannan samanaikaiset päivitykset.



- **Priorisointi (järjestys) ajan** suhteen ratkaisisi ongelman oikein.

- Minkä ajan???
- 2 asiakasta, 2 palvelinta = 4 eri aikaa.
- Mikä tahansa yksiselitteinen järjestys kelpaisi.

- Kun ohjelmoija on muuttanut lähdekoodia, hän ajaa ohjelman make.
 - make tarkastelee kunkin lähdekoodisen ja käännetyn tiedoston muutosaikaleimaa ja päättää mitkä tiedostot on käännettävä uudelleen.



- Vaikka *P.c* on muutettu *P.o* luomisen jälkeen, kellojen eron vuoksi *P.c*:n aikaleima on pienempi.

⇒ *P.c* :tä ei käännettäisi!

Ratkaisuvaihtoehtoja:

⇒ (Automaattinen säännöllinen) kellojen **synkronointi**, (sivu 123).

- Ongelmia edelleen:
 - Vaikka pääsemmekin ms (tai μ s) tarkkuuteen, kaksi erillistä tapahtumaa voi silti vaihtaa järjestystä.
 - Erityisesti **synkronoinnin tarkkuus on parhaimmillaankin viestinvälityksen** luokkaa, joten viestin **lähetystä ja vastaanottoa ei voida erottaa** mittaustarkkuuden puitteissa.

- Useimmiten tapahtuman (tarkka) fyysinen aika ei ole tärkeä.

⇒ Tärkeää on toisistaan riippuvien tapahtumien keskinäinen järjestys!

- Make -esimerkki (s. 130)
- Esim. kahdennetun tietokannan samanaikaiset päivitykset. (s. 129).
- Fyysistä aikaa/kelloa ei tarvita, riittää virtuaalinen, looginen aika.
- Looginen aika toteutetaan loogisilla kelloilla.
- Loogisilla kelloilla tehdään loogisia aikaleimoja.

⇒ Eri prosesseissa tapahtuneiden tapahtumien keskinäinen järjestys on kriittinen.

Järjestys voi perustua kahteen tilanteeseen:

1. Jos kaksi tapahtumaa tapahtuu **samassa prosessissa** (säikeessä), ko. prosessi näkee järjestyksen yksiselitteisesti.
2. Kun viesti lähetetään prosessista toiseen, **lähettäminen tapahtuu ennen vastaanottamista**.

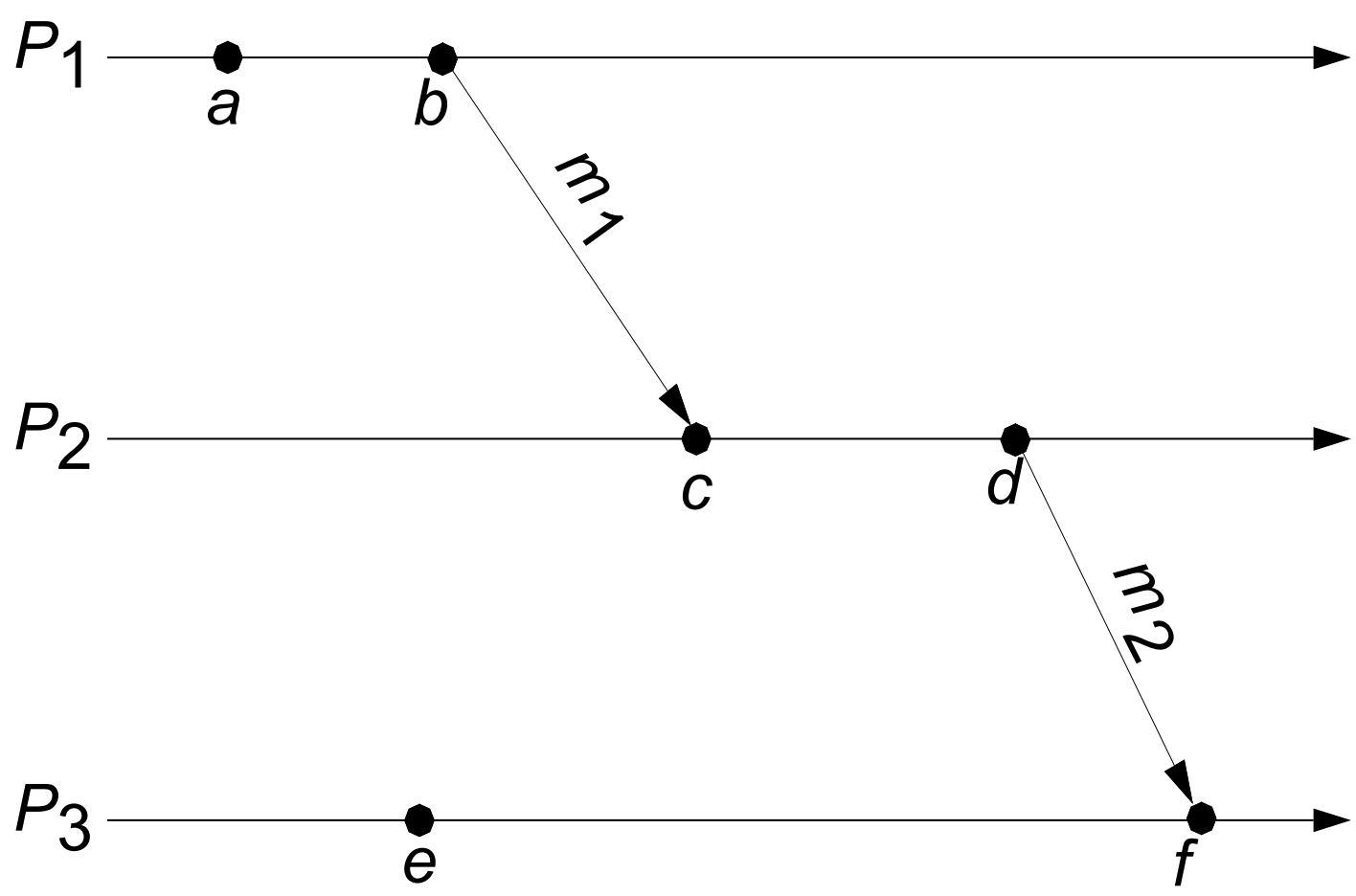
Looginen järjestäminen perustuu *tapahtui ennen* -suhteeseen (happened-before relation), käytetään merkintää \rightarrow

1. $a \rightarrow b$, jos a ja b ovat tapahtumia samassa prosessissa ja a tapahtui ennen b :tä.
2. $a \rightarrow b$, jos a on viestin m lähetys jostakin prosessista ja b on ko. viestin vastaanotto jossain toisessa prosessissa.
3. Jos $a \rightarrow b$ ja $b \rightarrow c$, niin $a \rightarrow c$ (suhde on transitiivinen).

\Rightarrow Jos $a \rightarrow b$, sanomme b :n seuraavan a :ta kausaalisesti (syyperäisesti).

- Toisin sanoen tapahtumat ovat kausaalisesti yhteydessä, niillä on syy-yhteys (causally related).
- Järjestelmässä voi olla (paljon) tapahtumia jotka eivät ole keskenään tapahtui ennen -suhteessa.
 - Jos sekä $a \rightarrow e$, että $e \rightarrow a$ ovat epätosia sanomme a ja e olevan samanaikaisia (concurrent), tai riippumattomia; merkitään $a \parallel e$.

- P_1, P_2, P_3 : prosesseja;
- a, b, c, d, e, f : tapahtumia;
- $a \rightarrow b, c \rightarrow d, e \rightarrow f, b \rightarrow c, d \rightarrow f, a \rightarrow c, a \rightarrow d, a \rightarrow f, b \rightarrow d, b \rightarrow f, \dots$
- $a \parallel e, c \parallel e, \dots$



⇒ Fyysisiä kelloja käyttäen tapahtui ennen -suhdetta ei välttämättä voida ilmaista.

- On mahdollista, että $b \rightarrow c$ ja että $T_b > T_c$ (T_b on b :n fyysinen aika (tietokoneessa jossa b suoritetaan)).

⇒ Käytetään sensijaan loogisia kelloja joilla tapahtui ennen -suhde voidaan ilmaista.

Looginen kello on jatkuvasti kasvava ohjelmallinen laskuri.

- Jokaisessa järjestelmän prosessissa P_i on looginen kello C_{P_i} .
- Loogisen kellon arvoja käytetään aikaleimaamaan tapahtumia. Tapahtumat saavat kasvavat aikaleimat.
- $C_{P_i}(a)$ on tapahtuman a aikaleima prosessissa P_i .
- Loogisen kellon ja fyysisen kellon välillä ei ole mitään yhteyttä.

Jotta looginen kello huomioisi tapahtui ennen -relaation on se toteutettava siten, että

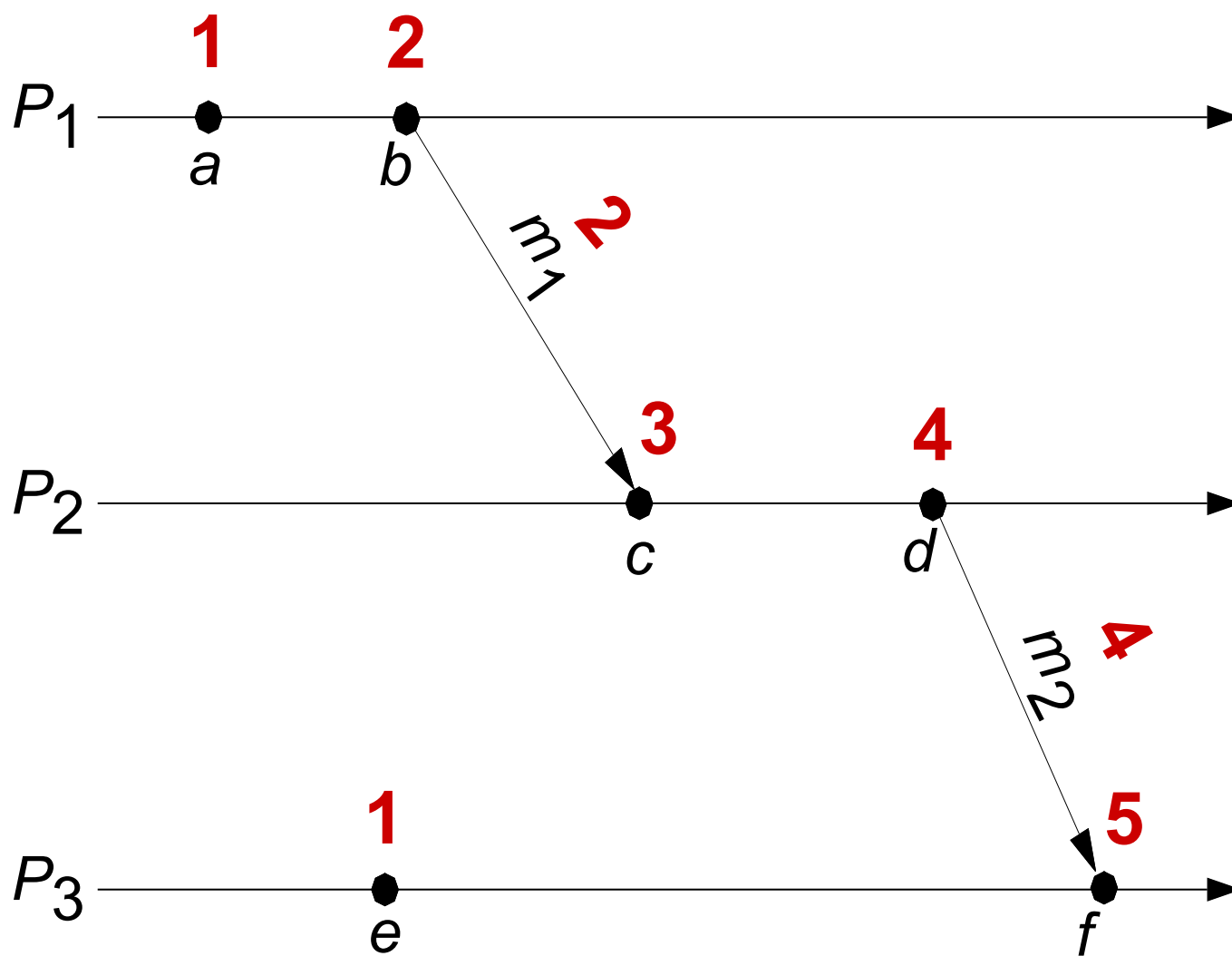
- jos $a \rightarrow b$, niin $C(a) < C(b)$

⇒ Loogiset kellot toteutetaan seuraavilla säännöillä:

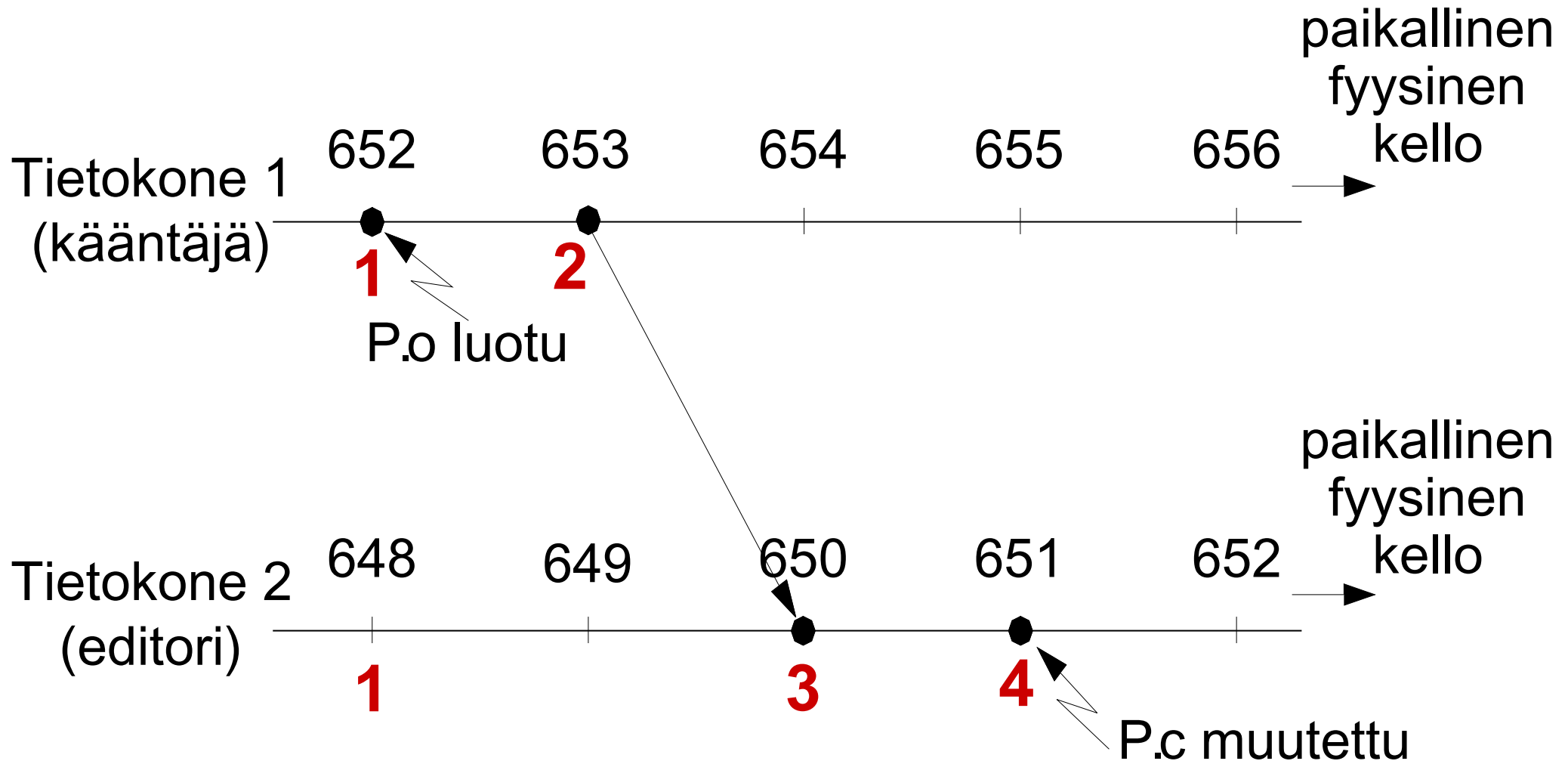
1. C_{P_i} :tä kasvatetaan ennen jokaista tapahtumaa prosessissa P_i :
 $C_{P_i} := C_{P_i} + 1$.
2. Kun a on tapahtuma joka lähettää viestin m prosessista P_i , sisällytetään aikaleima $t_m = C_{P_i}(a)$ mukaan viestiin m . ($C_{P_i}(a)$ on loogisen kellon arvo joka saatiin säännöllä 1).
3. Vastaanottaessa viestiä m prosessissa P_j , prosessin loogista kelloa C_{P_j} päivitetään seuraavasti: $C_{P_j} := \max(C_{P_j}, t_m)$.
4. C_{P_j} :n uutta arvoa käytetään aikaleimaamaan viestin m vastaanottaminen prosessissa P_j (soveltaen sääntöä 1, eli kasvatetaan yhdellä).

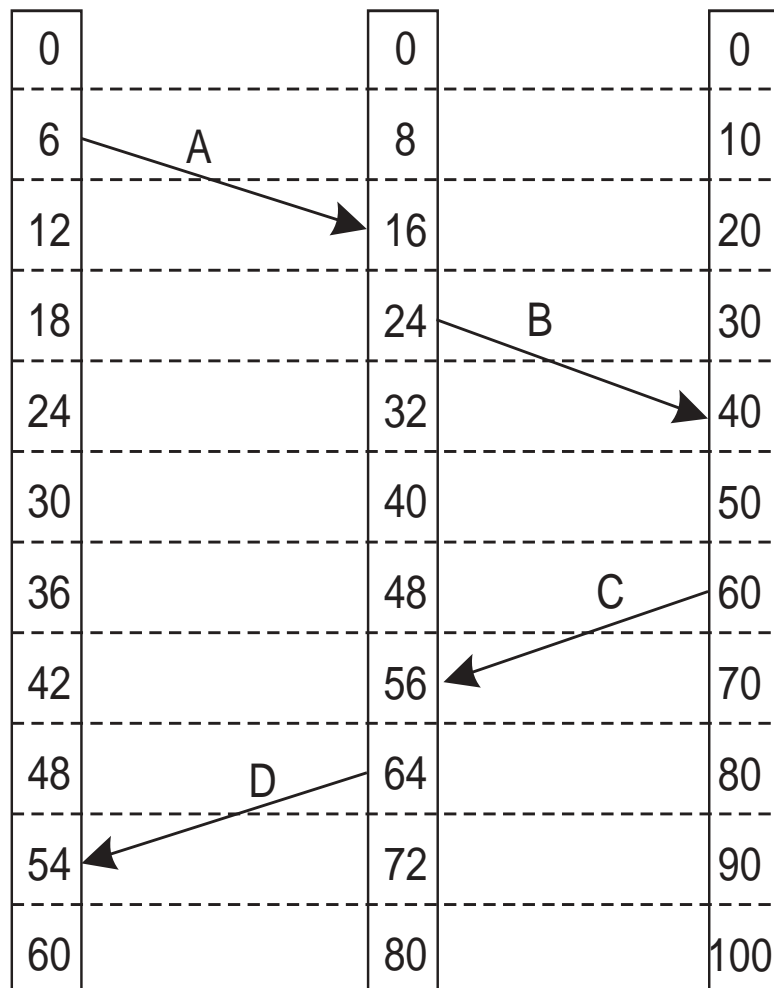
Toteuttaako tarpeet?

- Jos a ja b tapahtuvat samassa prosessissa ja a tapahtui ennen b :tä, niin $a \rightarrow b$, ja säännön 1 nojalla $C(a) < C(b)$.
- Jos a on tapahtuma joka lähettää viestin m ja b on ko. viestin vastaanotto-tapahtuma toisessa prosessissa, niin $a \rightarrow b$, ja $C(a) < C(b)$ (säännöt 2 ja 3).
- Jos $a \rightarrow b$ ja $b \rightarrow c$, niin $a \rightarrow c$, ja (induktiolla) $C(a) < C(b)$.



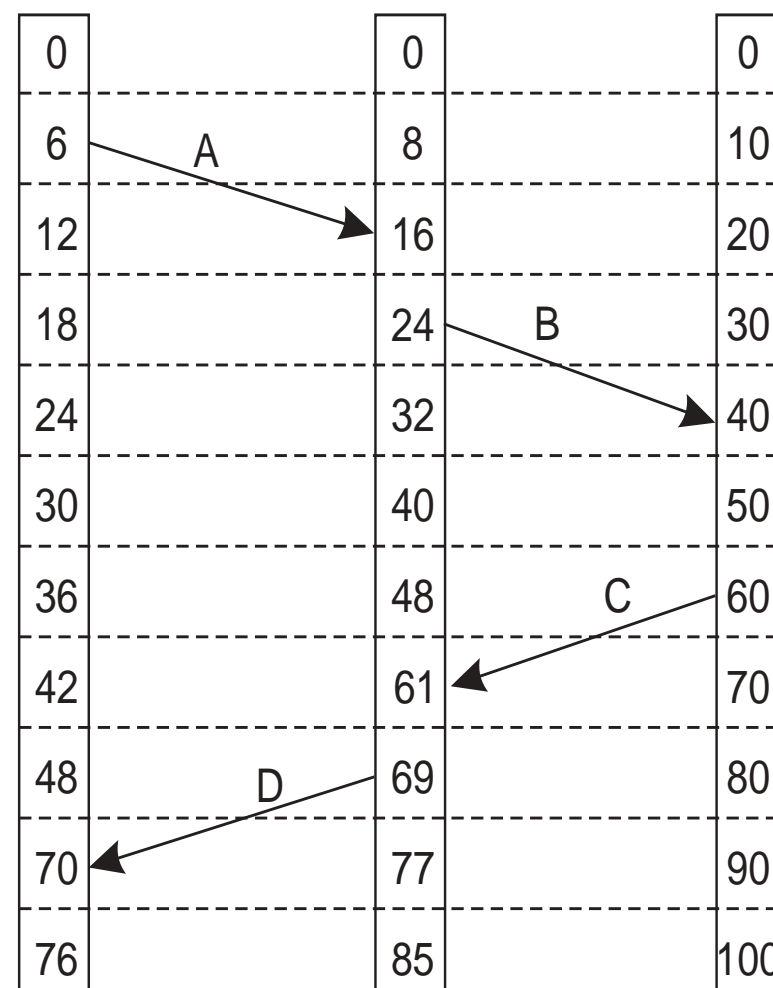
- make-esimerkissä oletamme, että käänntänyt ohjelma kuittaa käänntö-
sen valmistumisen editoriprosessille. Näin tiedosto *P.o* voidaan aikalei-
mata loogisella kellolla.





(a)

Paikallinen fyysinen kello



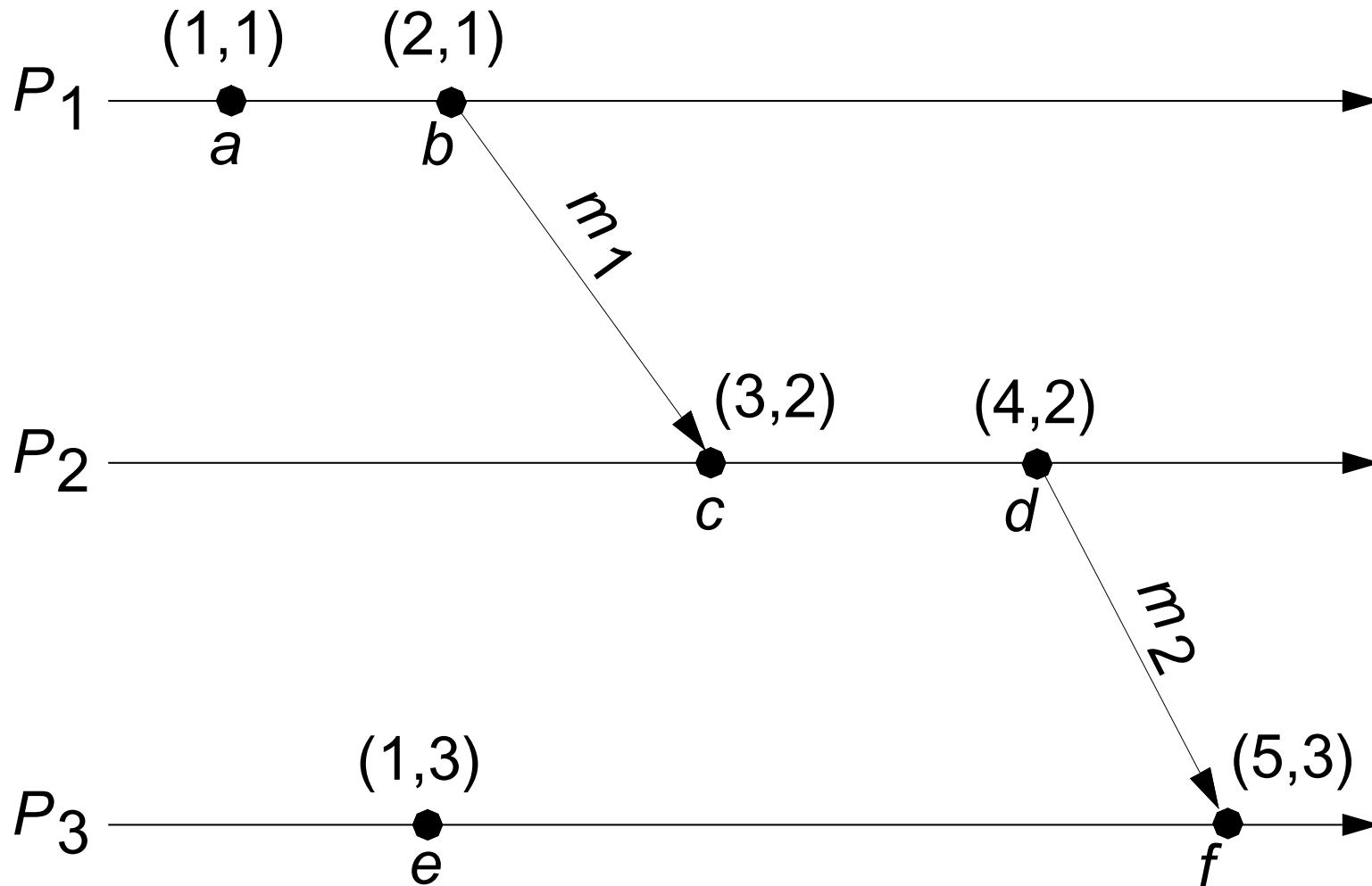
(b)

Paikallinen looginen kello [4]

- Lamportin loogiset kellot määräävät tapahtumille vain **osittaisen järjestyksen**.
 - Täysin **erillisillä** tapahtumilla voi olla **sama aikaleima**.
- Monessa sovelluksessa tarvitaan **täydellinen järjestys** (total ordering) kaikille tapahtumille, ts. kahdella tapahtumalla ei ole koskaan samaa aikaleimaa ja kaikille aikaleimoille löytyy **yksikäsitteinen järjestys**.
 - Esimerkiksi kahdennetun tietokannan päivitys (sivu 129).

⇒ **Täydellisen järjestyksen aikaansaamiseksi esitellään globaali looginen aikaleima:**

- Globaali looginen aikaleima prosessin P_i tapahtumalle a on **pari** $(C_{P_i}(a), i)$, missä i on prosessin P_i tunnus.
- Määritellään täydellinen järjestys: $(C_{P_i}(a), i) < (C_{P_j}(b), j)$ jos ja vain jos $C_{P_i}(a) < C_{P_j}(b)$ tai $C_{P_i}(a) = C_{P_j}(b)$ ja $i < j$.

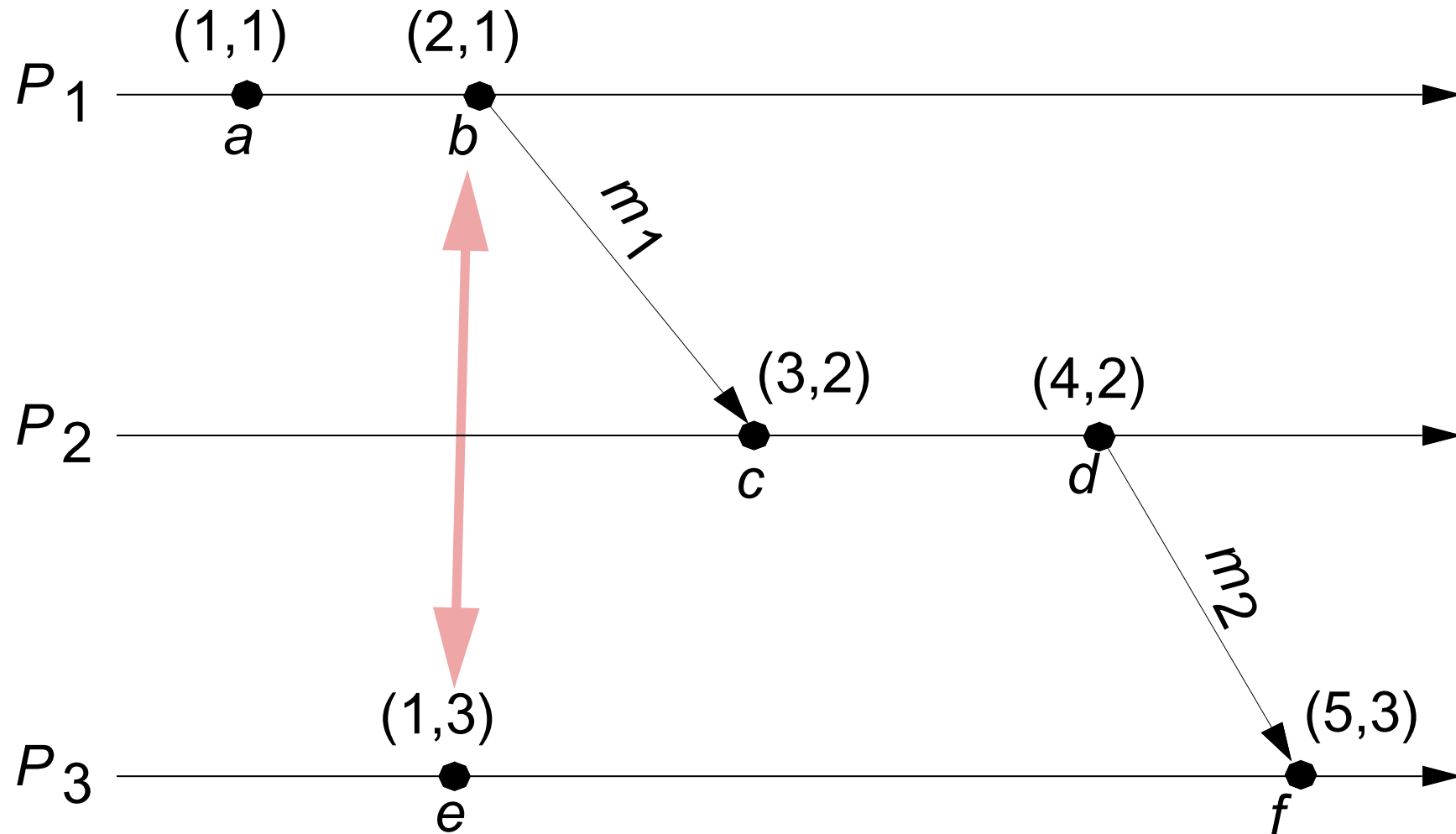


- Esimerkki täydellisen järjestyksen käytöstä: järjestetty monilähetys.

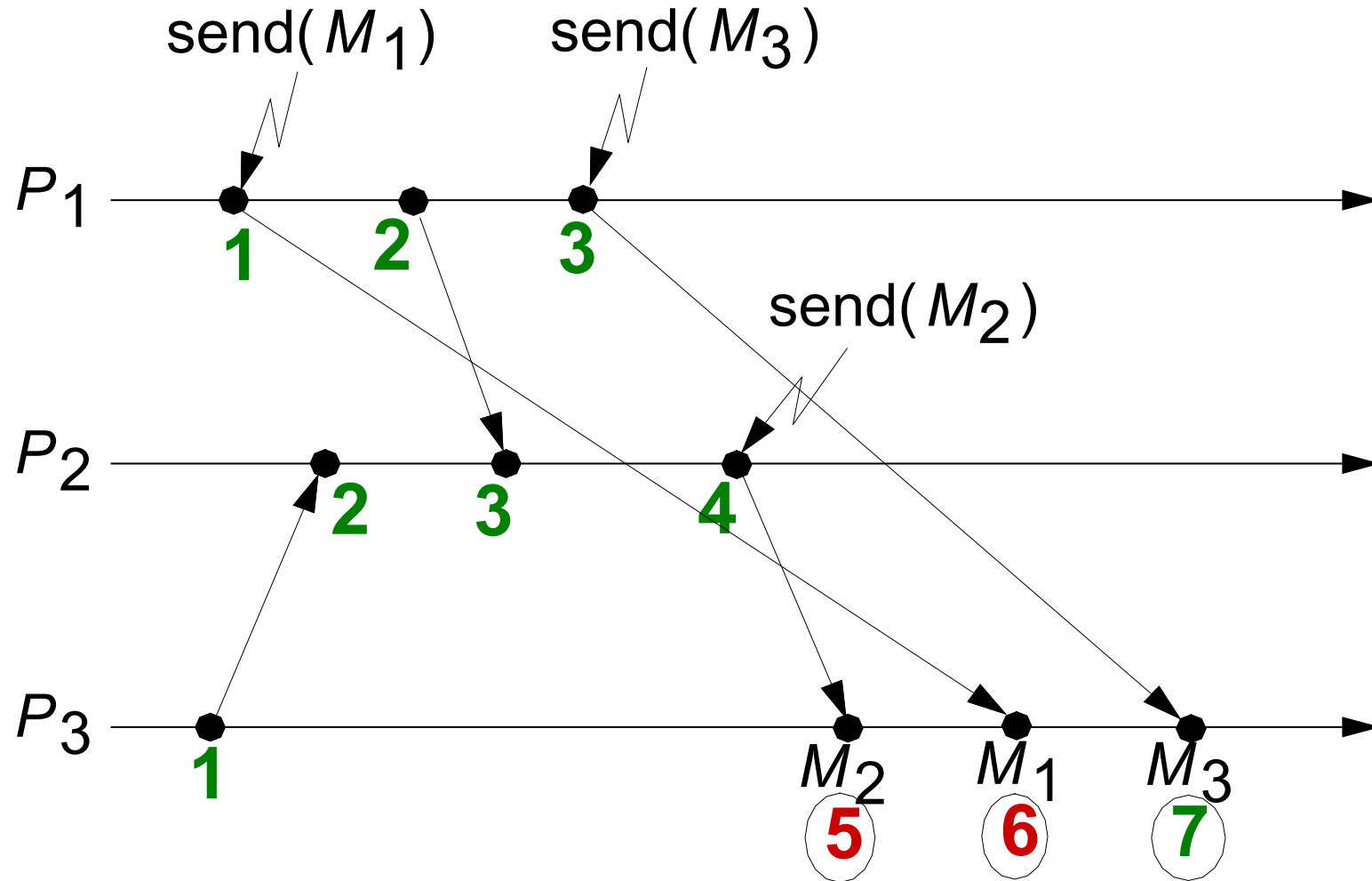
⇒ Nämäkään loogiset kellot eivät ole riittävän vahvoja määräämään kahden tapahtuman kausaalista yhteyttä. 143

- Jos $a \rightarrow b$, niin $C(a) < C(b)$.
- Mutta päättely ei välttämättä pidä paikkaansa päinvastoin (jos tapahtumat tapahtuivat eri prosesseissa):
 - Vaikka $C(a) < C(b)$, ei $a \rightarrow b$ välttämättä pidä paikkaansa.
 - On vain varmaa, ettei $b \rightarrow a$ pidä paikkaansa.

- $C(e) < C(b)$, vaikkei tapahtumien e ja b välillä ole kausaalista riippuvuutta.
- Pelkkiä aikaleimoja tarkastelemalla emme voi tietää **ovatko tapahtumat toisistaan riippuvia** vai eivät.



- Esimerkiksi viestit olisi hyvä käsitellä niiden **lähetysten kausaalissa järjestyksessä**.

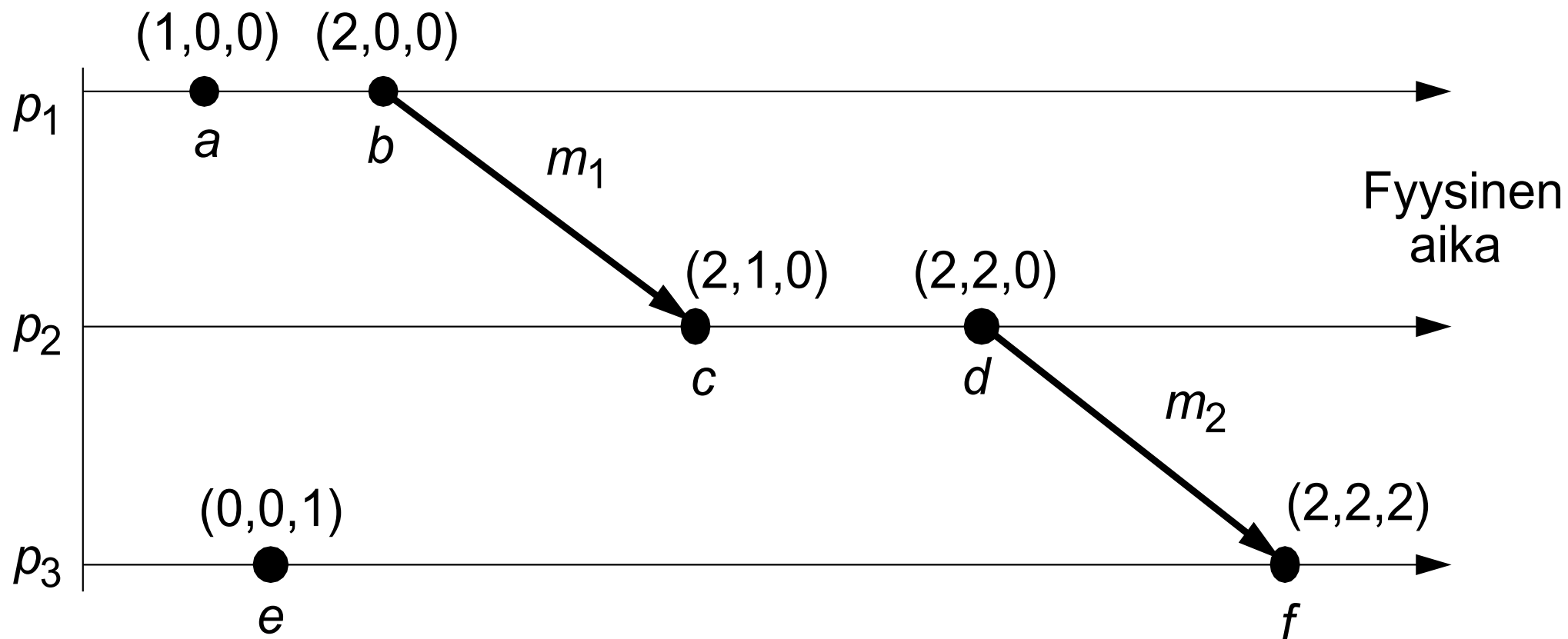


- Prosessi P_3 vastaanottaa viestit M_1 , M_2 ja M_3 .
 $M_1 \rightarrow M_2$, $M_1 \rightarrow M_3$, $M_3 \parallel M_2$
- M_1 pitäisi käsitellä ennen viestejä M_2 ja M_3 . Kuitenkaan prosessin P_3 ei tarvitsisi odottaa viestiä M_3 ennen kuin se käsittelee viestin M_2 (vaikka M_3 :n looginen aikaleima on pienempi kuin M_2 :n).

Ratkaisu kausaaliseen järjestämiseen: **vektorikello**

146

Looginen aika



- Prosessin P_i vektorikello V_i on alkio jokaista prosessia kohti.
- Jokainen prosessi ylläpitää **parasta tietoaan jokaisen toisen prosessin loogisesta kellosta**.
 - Koska loogiset kellot eivät koskaan vähene, **maksimi viimeisimmistä tiedoksi saaduista arvoista** on paras tieto.
 - Kopio omasta vektorikellosta välitetään jokaisessa viestissä.

\Rightarrow Tavoite oli siis, että $C(e) < C(e') \Rightarrow e \rightarrow e'$.

Muodollisemmin:

1. Aluksi $V_i[j] = 0$ jokaiselle $i, j = 1, 2, \dots, N$.
2. Juuri ennen kuin p_i leimaa tapahtuman, se asettaa $V_i[i] := V_i[i] + 1$.
3. P_i sisällyttää koko kellon $t = V_i$ jokaiseen lähettämäänsä viestiin.
4. Kun P_i vastaanottaa viestin aikaleimalla t , se päivittää omaa kelloaan $V_i[j] := \max(V_i[j], t[j]) \forall j$, eli **maksimikullekin taulukon alkiolle**.
 - Kellossa V_i , $V_i[i]$ kertoo montako tapahtumaa P_i on aikaleimannut.
 - $V_i[j]$ (missä $j \neq i$) kertoo niiden tapahtumien määrän prosessissa P_j jotka ovat **vaikuttaneet** prosessin P_i tapahtumiin.
 - Prosessi P_j on voinut leimata jo enemmänkin tapahtumia, mutta tieto siitä ei ole (vielä) tullut prosessiin P_i , ja näin ollen ko. tapahtumat eivät ole vaikuttaneet prosessin P_i nykytilaan.

Vektorikellojen vertailu

- $V = V'$ jos ja vain jos $V[j] = V'[j]$ kaikille $j = 1, 2, \dots, N$.
- $V \leq V'$ jos ja vain jos $V[j] \leq V'[j]$ kaikille $j = 1, 2, \dots, N$.
- $V < V'$ jos ja vain jos $V \leq V'$ ja $V \neq V'$.

- Hajautetussa järjestelmässä ei ole yhteistä fyysistä kelloa.
 - Eri prosessien/tietokoneiden omat fyysiset kellot voidaan **synkronoida jollain tarkkuudella**.
- Useimmiten **fyysinen aika ei ole tärkeä** (eikä riittävän tarkka), vaan käytetään loogista järjestystä ja loogisia kelloja.
- Lamportin loogiset kellot ovat kasvavia **kokonaislukulaskureita**.
- **Täydellinen järjestys** saavutetaan lisäämällä **prosessitunniste** loogiseen kelloon.
- Vektorikellolla saavutetaan **kausallinen järjestys**.
 - Vektorikelloonkin voidaan lisätä prosessitunniste jos halutaan täydellinen järjestys (tai määritellä jokin muu yksikäsitteinen sääntö jolla samanaikaisia vektorikelloleimoja vertaillaan).

Vikamallit (failure)

Vikatyypit	Kuvaus
Pysähtyminen (fail-stop)	Palvelin pysähtyy, mutta toimi oikein, muut havaitsevat pysähtymisen.
Kaatuminen (crash)	Palvelin pysähtyy, mutta toimi loppuun asti oikein. Muut eivät saa kaatumisesta tietoa.
Tekemättäjättövirhe (omission) Vastaanoton tekemättäjättö Lähetysten tekemättäjättö	Palvelin ei vastaa tuleviin pyyntöihin tai kommunikaatiokanava hävittää viestin. Palvelin ei vastaanota viestejä Palvelin ei lähetä viestejä
Ajoitusvirhe	Palvelin tai kommunikaatiokanava vastaa annetun aikajakson ulkopuolella tai kellot heittävät liikaa.
Vastausvirhe Arvovirhe Tilavirhe	Palvelimen vastaus on virheellinen Vain vastaus on virheellinen Palvelin joutuu väärään tilaan
Satunnainen (arbitrary) virhe	Palvelin tai kommunikaatiokanava tuottaa mielivaltaisia vastauksia mielivaltaisina aikoina

- ⇒ Vikoja voi esiintyä sekä prosesseissa, että kommunikaatiokanavissa. Vika voi olla laitteistossa tai ohjelmistossa.
- Vikaantumismalleja tarvitaan suunniteltaessa järjestelmiä jotka vikaantuvat ennustettavalla tavalla (ja suunniteltaessa vioista toipumista).

Tekemättäjättämisvirhe (omission failures)

⇒ Prosessi tai kommunikaatiokanava ei tee kaikkea mitä sen pitäisi tehdä.

Seuraavat eivät ole tekemättäjättämisvirheitä:

- Toimenpide viivästyy (paljonkin), mutta lopulta suoritetaan.
- Toiminnon tulos on virheellinen.

Synkronisessa järjestelmässä tekemättäjättämisvirheet voidaan havaita **aikakatkuilla** (timeout).

- Jos viestinvälitys on luotettavaa, ja viesti ei saavu lasketun ajan kuluessa, palvelin on kaatunut.
- Esim. Pepperland divisions:
 - Molemmat osastot lähettävät toiselle viestin säännöllisesti (väli I).
 - Jollei viestiä tule ajassa $I + max - min$, virhe on tapahtunut.

- **Asynkronisessa järj.** kyseessä saattaa olla vain tavallista pidempi **viive**. 153
- **Yhteisymmärrys asynkronisessa järjestelmässä** jossa tapahtuu tekemättäjäättämismvirheitä on **teoriassa mahdotonta** (suoralla algoritmilla).
 - Protokollan **viimeinen viesti saattaa kadota** (tai olla matkalla).
 - Viimeisen viestin lähettäjä ei voi tietää menikö viesti perille vai ei.
 - Viim. viestin vastaanottaja tietää, ettei lähettäjä voi olla varma...
 - Missä tahansa (minimaalisessa) protokollassa viimeinen viesti on ratkaiseva, ja rekursiivisesti.
- Ns. pysähtymismvirhe (fail-stop) on vikaantumisen jonka muut osapuolet voivat havaita luotettavasti, erityisesti prosessista.

Satunnaiset virheet (arbitrary, Byzantine)

⇒ Kaikkein väljin ja pahin vikatyyppejä.

- Laskentaa tai kommunikaatiota jätetään **tekemättä** tai/ja **ylimääräisiä** operaatioita/viestejä suoritetaan. Tulokset voivat **puuttua** tai olla **vääriä**.
- Esim., Pepperland divisions:
 - Keltainen vihollinen voi napata ja aivopestä viestinvälittäjän.
 - Hyviä (**todennäköisiä**) ratkaisuja saadaan aikaan salakirjoitusjärjestelmällä ja allekirjoituksilla.
- Bysanttilaisten kenraalien ongelmassa mitkä tahansa m kenraalia ovat pettureita.
 - Yhteisymmärrys on mahdollinen jos mukana on vähintään $2m+1$ vilpittöntä osanottajaa.

⇒ Ajoitusvirheitä voi tapahtua synkronisissa järjestelmissä jossa operaatioille, viesteille ja kellovirheille on asetettu rajat.

- Jos mikä tahansa aikaraja rikotaan, kyseessä on ajoitusvirhe.
- Palautuminen melko vaikeaa.

⇒ Vaikka virheitä tapahtuu jollakin palvelutasolla, voimme havaita ja korjata ne, jolloin **ylemmälle tasolle näkyy** luotettava palvelu.

- Virheen **havaitseminen**: esim: aikakatkaaisu, pakettien numerointi, tarkistussummat, kuittaukset, jne.
- Virheen **korjaaminen**: esim. uudelleenlähetys, uudelleenreititys, virheenkorjauskoodaukset, ylläpito.
- Virheen **korjaamisen havaitseminen**: toisteisten viestien havaitseminen.

Mitä on luotettava (kahdenvälinen) kommunikaatio?

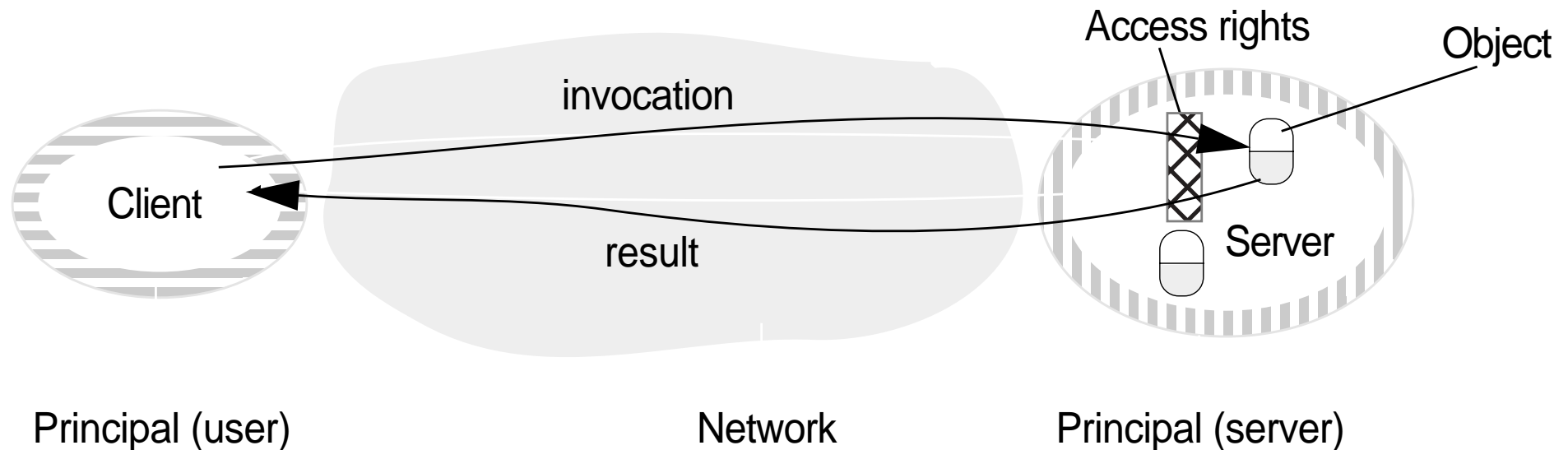
- **Oikeellisuus** (validity)
 - Kaikki lähetetyt viestit menevät lopulta perille vastaanottajalle.
- **Eheys** (integrity)
 - Kaikki vastaanotetut viestit ovat samanlaisia kuin ne olivat lähetettyinä, eikä mitään viestiä lähetetä kahdesti.

Turvallisuusmallit (security)

- Turvallisuuden määritelmä: kts. Informaation/palvelujen turvallisuus sivulla 57.

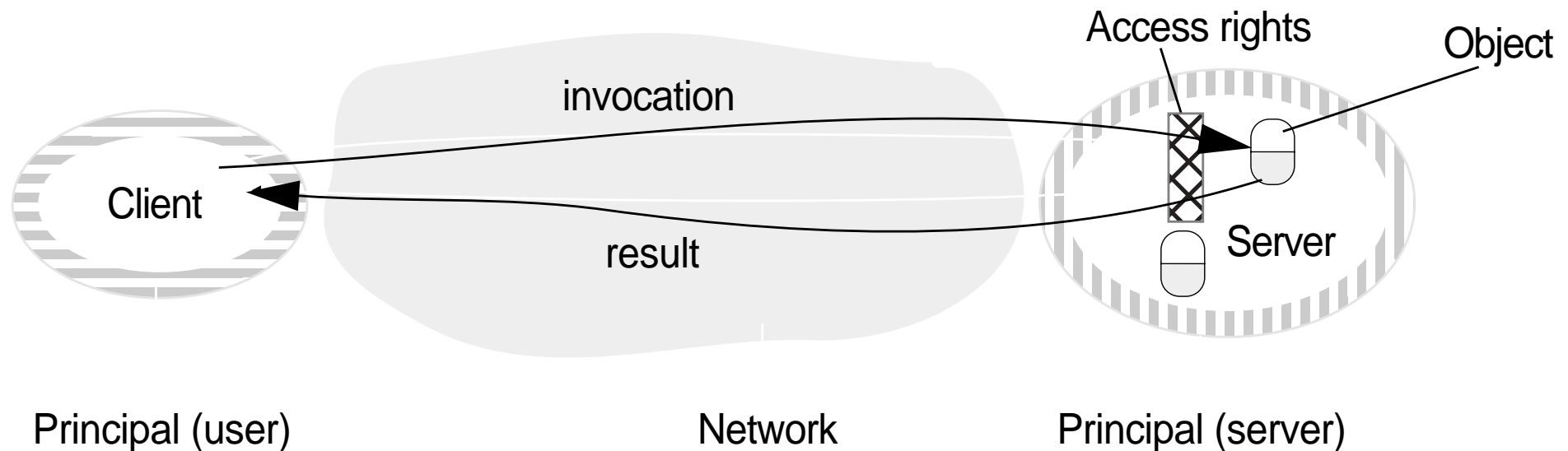
⇒ HJ yksi päämotivaatioista on **resurssien jakaminen**.

- Tämä edellyttää saanti- ja muutosoikeuksien kontrollia, eli **turvaamista**.
- Turvaaminen voidaan tehdä turvaamalla sekä **prosessit**, että **kommunikaatiokanavat** (ja niiden sisältämät objektit) luvaton käyttöä vastaan.

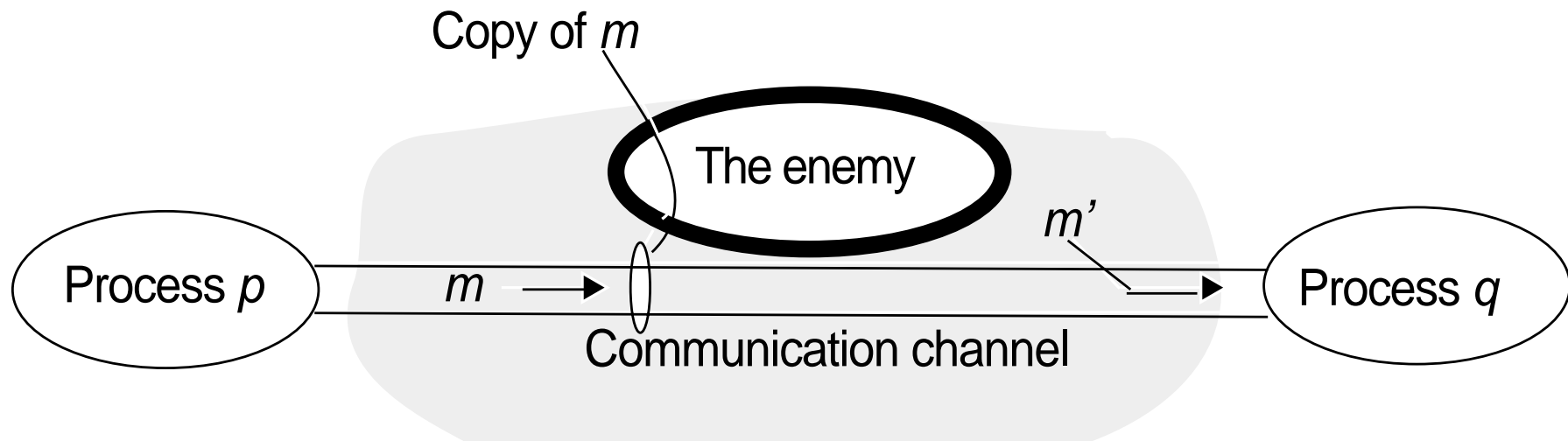


Tunnistaminen (identiteetti)

- Asiakas-palvelin -arkkitehtuurissa palvelimen on tarkastettava jokaisen **asiakkaan identiteetti** ja oikeus päästä käsiksi objekteihin.
- Asiakkaan pitää (useimmiten) tarkistaa **palvelimen identiteetti**.
- **Identiteetin tarkistaminen ei ole helppoa.**
 - Itseasiassa se on paljon **vaikeampaa** kuin itse tiedon salaaminen.



Vihollinen



- (Pahin mahdollinen) vihollinen voi
 - **lähettää** minkä tahansa viestin mille tahansa prosessille,
 - **lukea** ja/tai kopioida minkä tahansa prosessienvälisen viestin.

Uhat prosesseille

- Hajautetun järjestelmän prosessit on suunniteltu **vastaanottamaan viestejä** (pyyntöjä tai vastauksia).
- Kelvollisten viestien sijaan vihollinen voi lähettää **viallisia viestejä**.
- Viestin lähettäjän tunnistus on oletusarvoisesti epäluotettava.
 - Useimmat alemman tason protokollat (kuten IP, Ethernet, WLAN) sisältävät lähettäjän osoitteen, mutta se on **helppo väärentää**.
- Sovellusten (tai ylemmän tason protokollien) on tunnistettava tai valtuutettava lähettäjä (asiakas ja palvelin).

- Vihollinen voi poistaa, kopioida, muuttaa tai lisätä viestejä.
- Viestien **luottamuksellisuus** tai eheys voi joutua alttiiksi.
- Napattujen viestien uudelleenlähetyks voi aiheuttaa ongelmia huonosti suunnitellussa järjestelmässä...
- **Palvelunestohyökkäys** (denial of service, DoS): vihollinen häiriköi luvallisten (suojattujen) käyttäjien toimia.
 - Palvelu voi ruuhkautua/sulkeutua luvallisilta käyttäjiltä.
 - Ylikuormittava hyökkäys voi saada järjestelmän **epävakaaseen tilaan** joka voi **paljastaa uusia heikkouksia**.

Turvallisuushilta suojaus

⇒ Käytetään turvattuja kanavia.

- Kryptografia (salakirjoitus, salaus) tarjoaa keinoja turvalliseen tunnistamiseen ja turvattujen kanavien luomiseen.
 - Turvattu (tietoliikenne-) kanava tarjoaa tiedon luottamuksellisuuden ja suojan tiedon muuttamista, uudelleenlähetyksiä, jne vastaan.

Tietoturvan tekniikat [1,3]

- Perusteet
 - Kryptografiset tekniikat
 - Salaaminen
 - Oikeaksi todistaminen
 - Todistukset ja luvat
- Symmetriset ja asymmetriset salausalgoritmit
- Digitaaliset allekirjoitukset
- Suunnittelulähtökohdat

Schneier: Applied Cryptography [3] esipuheen alku:

There are two kinds of cryptography in this world: cryptography that will stop your **kid sister** from reading your files, and cryptography that will stop **major governments** from reading your files.

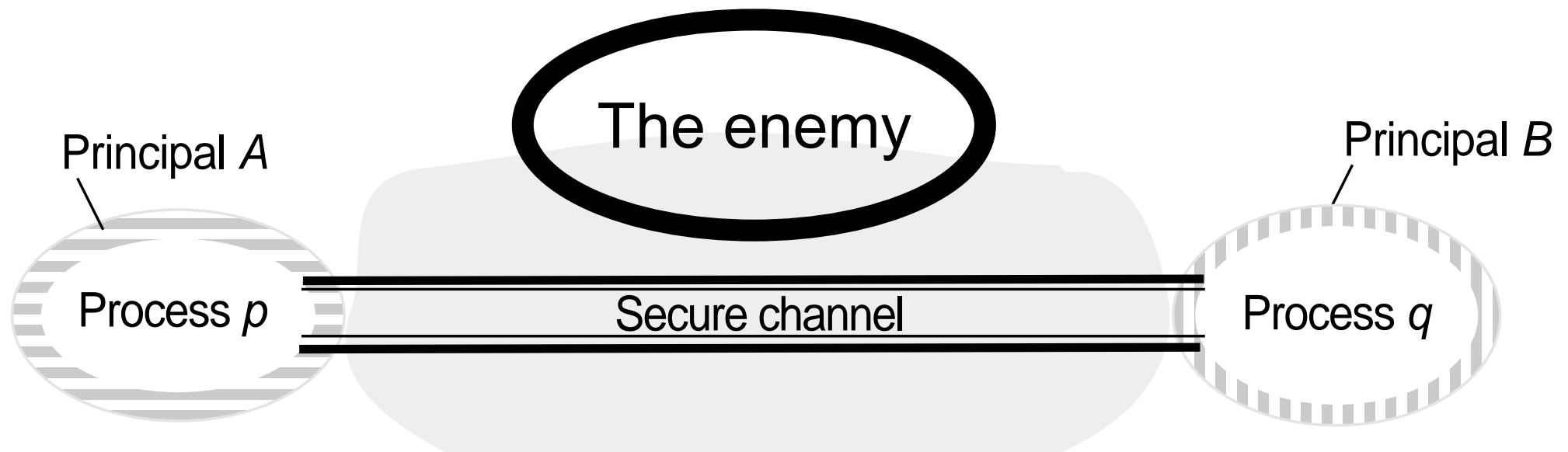
This book is about the latter.

If I take a letter, lock it in a safe, **hide** the safe somewhere in the New York, then tell you to read the letter, that's not security.

That's **obscurity**.

On the other hand, if I take a letter and lock it in a safe, and then **give you the safe** along with the design specifications of the safe and a hundred identical safes with their combinations so that you and world's best safecrackers can study the locking mechanism — and you still can't open the safe and read the letter —

that's security.



- Jokainen prosessi tuntee varmasti **vastapuolen identiteetin**.
- **Tieto(liikenne)** on yksityistä ja suojattu muuttamiselta, toistamiselta ja uudelleenjärjestelyltä.

⇒ **Hyödyntää kryptografiaa.**

- Salaisuus perustuu kryptografiseen peittämiseen.
- Tunnistus perustuu **todistuksiin salaisten avainten tuntemisesta**.
- Kryptografinen salaaminen perustuu:
 - **Sekoittamiseen** ja **levittämiseen** (confusion, diffusion).
 - Jaettuihin **salaisiin avaimiin** (ei salaisiin algoritmeihin).

- **Salakuuntelu** (eavesdropping)
 - Salaisen tiedon (tai sen kopioiden) haltuun saaminen.
- **Tekeytyminen** (masquerading)
 - Toisen toimijan identiteetin väärennetty käyttö.
- **Viestien muuttaminen** (tampering)
- **Välistä veto** (man in the middle)
 - Kaapataan jo yhteydenmuodostusviestit ja muodostetaan erilliset "turvatut" yhteydet molempien osapuolten kesken.
- **Uudelleenlähetys** (replaying)
 - Tallennetaan kaapattu viesti ja lähetetään se uudelleen myöhemmin.
- **Palvelunesto** (denial of service)
 - Kanavan tai resurssin tukkiminen.
- **Vihamielinen liikkuva ohjelma** (mobile code)
 - Liikkuva ohjelma pitäisi suorittaa **eristetysti**, mutta se voi tarvita paikallisia resursseja.
 - Niihin päästäkseen sen olisi tuotava mukanaan **valtakirja** tms.

Yleinen osanottajanotaatio salauksissa

Alice	Ensimmäinen osanottaja
Bob	Toinen osanottaja
Carol, Dave	Mahdollinen kolmas, neljäs osanottaja
Eve	Salakuuntelija (eavesdropper)
Mallory	Pahantekijä
Sara, Trent	Palvelin (salaus, tunniste, tms).
K_A	Alicen (salainen) avain
K_{AB}	Alicen ja Bobin yhteinen salainen avain
$K_{A_{priv}}$	Alicen yksityinen avain (vain hänen tuntema)
$K_{A_{pub}}$	Alicen julkinen avain
$\{M\}_K$	Viesti M salattuna avaimella K
$[M]_K$	Viesti M allekirjoitettuna avaimella K

⇒ Alice ja Bob tuntevat yhteisen salaisen avaimen K_{AB}

- Alice käyttää K_{AB} :ta ja sovittua salausfunktiota $E(K_{AB}, M)$ ja lähettää Bob:lle viestin $\{M_i\}_{K_{AB}}$.
- Bob lukee salatun viestin vastaavalla purkualgoritmilla $D(K_{AB}, M)$.
- Alice ja Bob voivat käyttää avainta K_{AB} niin kauan kuin sen voi olettaa pysyvän salaisena.

Ongelmia:

- **Avaimen jakaminen:** Miten Alice voi lähettää avaimen K_{AB} Bobille turvallisesti?
 - Tapaaminen (jos mahdollista), tunnistuspalvelin, avaimenvaihtoprotokollat, kts. eteenpäin mm. Diffie-Hellman sivu 180.
- Kommunikaation tuoreus: Kuinka Bob tietää, ettei $\{M_i\}$ ole kopio jonka Mallory lähetti uudelleen?
 - Avainkohtainen järjestysnumerointi ja aikaleimat auttavat.

Varmistettu kommunikaatio palvelimen avulla

- Bob on palvelin jonne Alicella on asiaa; **Sara on tunnistuspalvelin.**
- **Saralla** on yhteinen salainen avain K_A Alicen kanssa ja K_B Bobin kanssa.
- Alice lähettää viestin Saralle ja pyytää **lipuketta** (ticket) päästäkseen yhteyteen Bobin kanssa.
- Sara vastaa Alicelle $\{\{\text{Lipuke}\}_{K_B}, K_{AB}\}_{K_A}$.
 - Viesti sisältää **lipukkeen salattuna Bobin avaimella K_B** ja uuden salaisen avaimen K_{AB} , kaikki salattuna Alicen avaimella K_A .
- Alice purkaa viestin omalla avaimellaan K_A .
- Alice lähettää Bobille pyynnön päästä käsiksi resurssiin R :
 $\{\text{Lipuke}\}_{K_B}, \{\text{Alice}, R\}_{K_{AB}}$.
- Lipuke on itseasiassa $\{K_{AB}, \text{Alice}\}_{K_B}$. Bob avaa sen omalla avaimellaan K_B , tarkistaa Alicen nimen ja pääsylvat.
- Bob käyttää uutta avainta K_{AB} vastauksen lähettämiseksi Alicelle.

⇒ **Lipuke on salattu viesti joka sisältää sen saajan identiteetin ja uuden yhteysavaimen.**

- Tämä on yksinkertaistettu versio Needham ja Schroeder protokollasta. (ja Kerberos), kts <http://www.lsv.ens-cachan.fr/spore/>. 170
- Ei vielä sellaisenaan suojaa **uudelleenlähetysyökkäystä** vastaan.
 - Kertakäyttöisyys, ylimääräinen viesti salattuja satunnaislukuja ja aikaleimat auttavat.
- Ei oikein käyttökelpoinen sähköisessä kaupankäynnissä koska tunnistuspalvelu **ei skaalaudu** helposti.
 - Suorituskykyongelmat voidaan ratkaista, mutta palvelimella pitäisi olla **kaikkien asiakkaiden salaiset avaimet**.

⇒ Bobilla on julkinen/salainen avainpari $\langle K_{B_{pub}}, K_{B_{priv}} \rangle$.

- Alice hankkii **luotettavan tahon allekirjoittaman todistuksen** (certificate) taholta että Bobin julkinen avain on $K_{B_{pub}}$.
- Alice luo uuden yhteisen salaisen avaimen K_{AB} , **salaa sen julkisen avaimen algoritmilla** käyttäen avainta $K_{B_{pub}}$ ja lähettää sen Bobille.
- Bob käyttää vastaavaa yksityistä avainta $K_{B_{priv}}$ avatakseen sen.
- Julkisen avaimen salakirjoitus on liian hidasta varsinaiseen kommunikointiin, mutta avaintenvaihtoon se sopii.

Ongelmia

- Mallory voi **kaapata Alicen yhteydenoton** luotettavaan tahoon ja lähettää vastauksena oman julkisen avaimensa.
 - Näin Mallory voisi lukea kaikki myöhemmät viestit.
 - Alicella on oltava **etukäteen luotettavasti luotettavan** tahon tiedot (julkinen avain tai sen tiiviste).

⇒ Allekirjoitetut oikeellisuustodistukset voivat muodostaa **luottamusketjuja**.

- PKI (Public Key Infrastructure) yrittää automatisoida tämän.
 - Ei (vieläkään) kovin laajalti levinnyttä standardia.

- ⇒ Alice haluaa julkaista dokumentin M siten, että kuka tahansa voi varmistaa sen olevan aito (eikä muutettu).
- ⇒ Alice ei myöskään voi **kieltää** dokumenttia tekemäkseen.
- Alice laskee kiinteämittaisen **tiiviste**n (one-way hash, digest) dokumentista $\text{Tiiviste}(M)$.
 - Tiiviste voidaan laskea kryptografisella hajautusfunktiolla.
 - Alice **salaa tiiviste**n omalla salaisella avaimellaan $K_{A_{priv}}$, liittää sen dokumenttiin M ja julkaisee allekirjoitetun dokumentin $(M, \{\text{Tiiviste}(M)\}_{K_{A_{priv}}})$ halukkaiden saataville.
 - Bob saa allekirjoitetun dokumentin, laskee $\text{Tiiviste}(M)$:n.
 - Bob käyttää Alicen **julkista avainta** $K_{A_{pub}}$ **purkamaan** $\{\text{Tiiviste}(M)\}_{K_{A_{priv}}}$ ja vertaa sitä itse laskemaansa tiivisteeseen.
 - Jos ne täsmäävät, Alicen allekirjoitus on varmennettu.
 - **Kryptografinen hajautusfunktio** (tiiviste) (esim. MD5, SHA):
 - Nopea laskea, vaikea väärentää: ei ole keinoa muodostaa dokumenttia M (tai M') jolle hajautusfunktio antaisi annetun arvon $H(M)$.

- E = Salaus, D = Purku, K = Avain, M = Viesti.

Symmetriset salaimet (salainen avain)

$$\Rightarrow E(K, M) = \{M\}_K \qquad D(K, E(K, M)) = M$$

- Sama avain salauksessa ja purussa.
- $D(K, E(K, M))$ on oltava (liian) työlästä laskea jollei avainta K tunneta.
- Yleensä **lohko** (64/128 bittiä) kerrallaan (block cipher).
- **Vuosalaimeilla** (stream cipher) bitti/tavu kerrallaan (jatkuva virta).
 - Lohkosalaimet voidaan muuttaa vuosalaimeiksi (mm. Cipher Feedback Mode).
- Virheetöntä salausalgoritmia vastaan voidaan hyökätä vain raa'an voiman hyökkäyksellä: **kokeillaan kaikkia mahdollisia avaimia**, josko sillä purkaminen tuottaisi järkevän tuloksen.
 - Liian työlästä kun K riittävän pitkä, ~ 128 bittiä.

Epäsymmetrinen (julkinen avain)

- Erikseen **salausavain (julkinen)** ja **purkuavain (salainen)**: K_e, K_d .
 - $D(K_d, E(K_e, M)) = M$ ja $D(K_e, E(K_d, M)) = M$.
- Käyttää "salaovifunktioita" avaimenmuodostuksessa.
- Voidaan käyttää myös toisinpäin, kts. Digitaalinen allekirjoitus (s. 173).
- Salaaminen suhteellisen työlästä.
- Pitkät avaimet (> 512 bittiä, paitsi ECC).

Kertakäyttöavain (one-time pad)

⇒ Ainoa tunnettu salausalgoritmi joka on mahdoton murtaa!

- XORraa (joko-tai) viesti satunnaisella bittijonolla.
- Bitin saa käyttää vain kerran, satunnaisuuden on oltava aitoa.

Hybridiprotokollat (esim. SSL/TLS ja SSH)

- Käytetään epäsymmetristä salausta symmetrisen avaimen lähetykseen, jota sitten käytetään salaamaan varsinainen istunto.

- **TEA**: hyvin yksinkertainen, vapaa, 10 riviä C:tä. 128-bit avain, 23 Mt/s (P4 2.1 GHz) [1], XTEA: parannettu versio.
- **DES**: The US Data Encryption Standard (1977). Vanhentunut. 56-bit avain, 21Mt/sec.
- **Triple-DES**: 3 kertaa DES kahdella avaimella 112-bit avain, 9Mt/s.
- **IDEA**: International Data Encryption Algorithm (1990). 128-bit avain, (19Mt/s).
- **Blowfish**: Valinnainen avainpituus, vapaa, nopea (B. Schneier).
- **AES** (Rijndael): Uusi USA:n Advanced Encryption Standard (1997). 128/192/256 -bit avain. <http://csrc.nist.gov/encryption/aes/> (61/53/48 MB/s).
- Lisää algoritmeja, kts. Schneier [3].

TEA salausfunktio [1]

```
void encrypt(unsigned long k[], unsigned long text[]) {
    unsigned long y = text[0], z = text[1];
    unsigned long delta = 0x9e3779b9, sum = 0; int n;
    for (n= 0; n < 32; n++) {
        sum += delta;
        y += ((z << 4) + k[0]) ^ (z+sum) ^ ((z >> 5) + k[1]);
        z += ((y << 4) + k[2]) ^ (y+sum) ^ ((y >> 5) + k[3]);
    }
    text[0] = y; text[1] = z;
}
```

1
2
3
4
5
6
7
8
9
10

- Purkufunktio samanlainen, paitsi *sum* alustus ja $-=$ riveillä 6 ja 7.
- Tarkoituksena on siis **levittää** ja **sekoittaa selvätekstin ja avaimen** jokainen bitti keskenään salatekstin jokaiseen bittiin.

Epäsymmetriset salausalgoritmit

RSA: Ensimmäinen käytännössä toimiva ja edelleen käytetty (Rivest, Shamir and Adelman 1978)

- Vaihteleva avaimenpituus, yleensä 512-2048 bits. Nopeus 10-50 kt/s.
- Perustuu tekijöihinjaon työläyteen.

Elliptisten käyrien järjestelmä (elliptic curve, ECC)

- Uudempi, lyhyemmät avaimet, nopeampi.
- Ei perustu tekijöihinjakoon.

⇒ Epäsymmetriset algoritmit ovat ~100..1000 kertaa hitaampia kuin symmetriset.

⇒ Perustuu Diffie-Hellman ongelmaan: ei tunneta tehokasta (edes todennäköistä) algoritmia joka lukuja $g^a \bmod n$, $g^b \bmod n$, $g^c \bmod n$ käyttäen kertoisi onko $c = ab$ totta vai ei.

- Mahdollistaa kahden (tai useamman) osapuolen laskea salainen avain julkisessa verkossa **kunhan tunnistuksesta huolehditaan**.

Algoritmi:

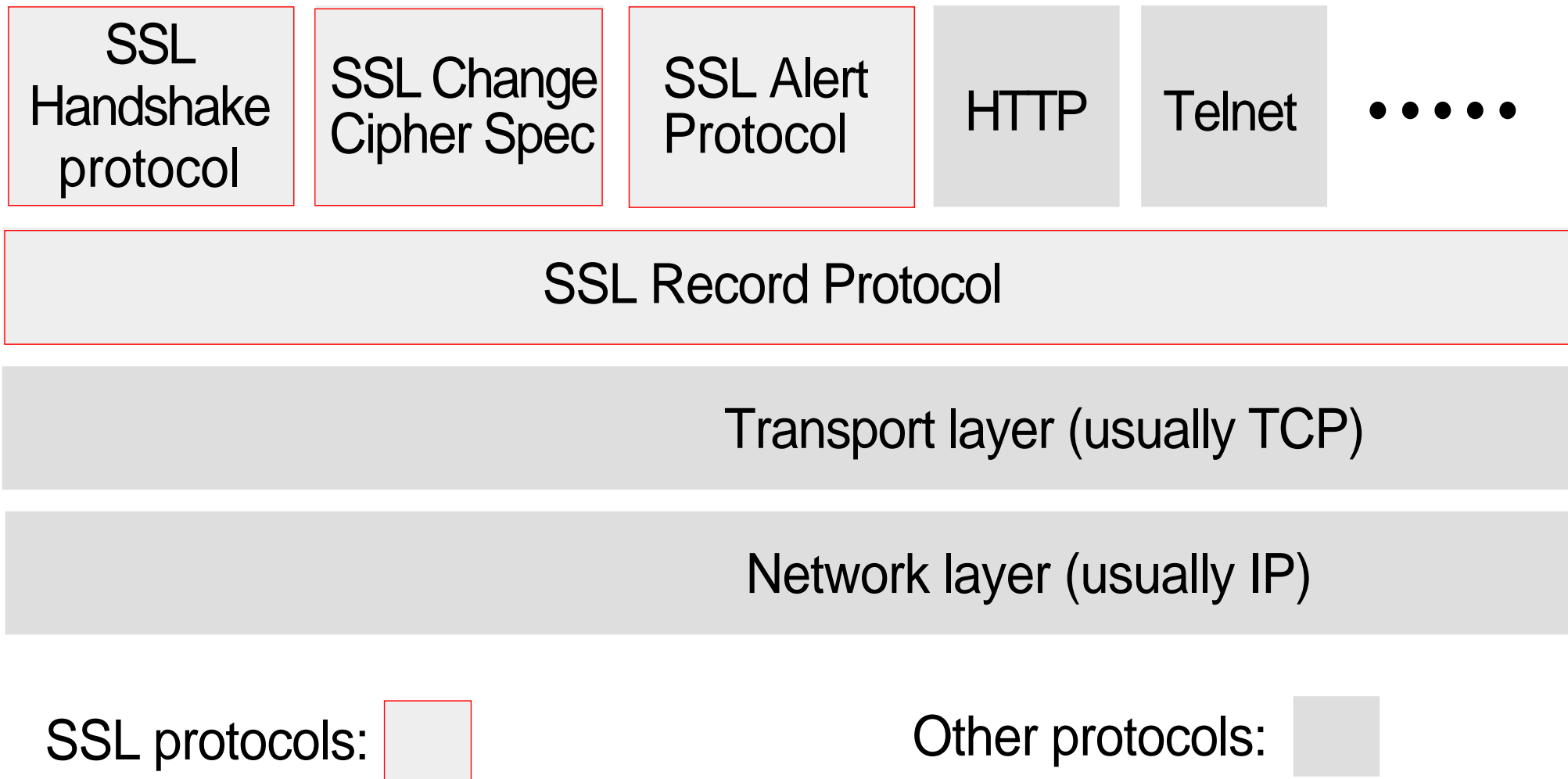
- $Z_n^* = \{1, \dots, n-1\}$, n on suuri alkuluku, g sopiva pieni luku (julkisia).

Alice	Viesti	Bob
Valitse satunnainen salainen a		
Laske ja lähetä $g^a \bmod n$	$g^a \bmod n \rightarrow$	Valitse satunnainen salainen b
	$\leftarrow g^b \bmod n$	Laske ja lähetä $g^b \bmod n$
Laske $(g^b \bmod n)^a$ $= g^{ab} \bmod n = K_{AB}$		Laske $(g^a \bmod n)^b$ $= g^{ab} \bmod n = K_{AB}$

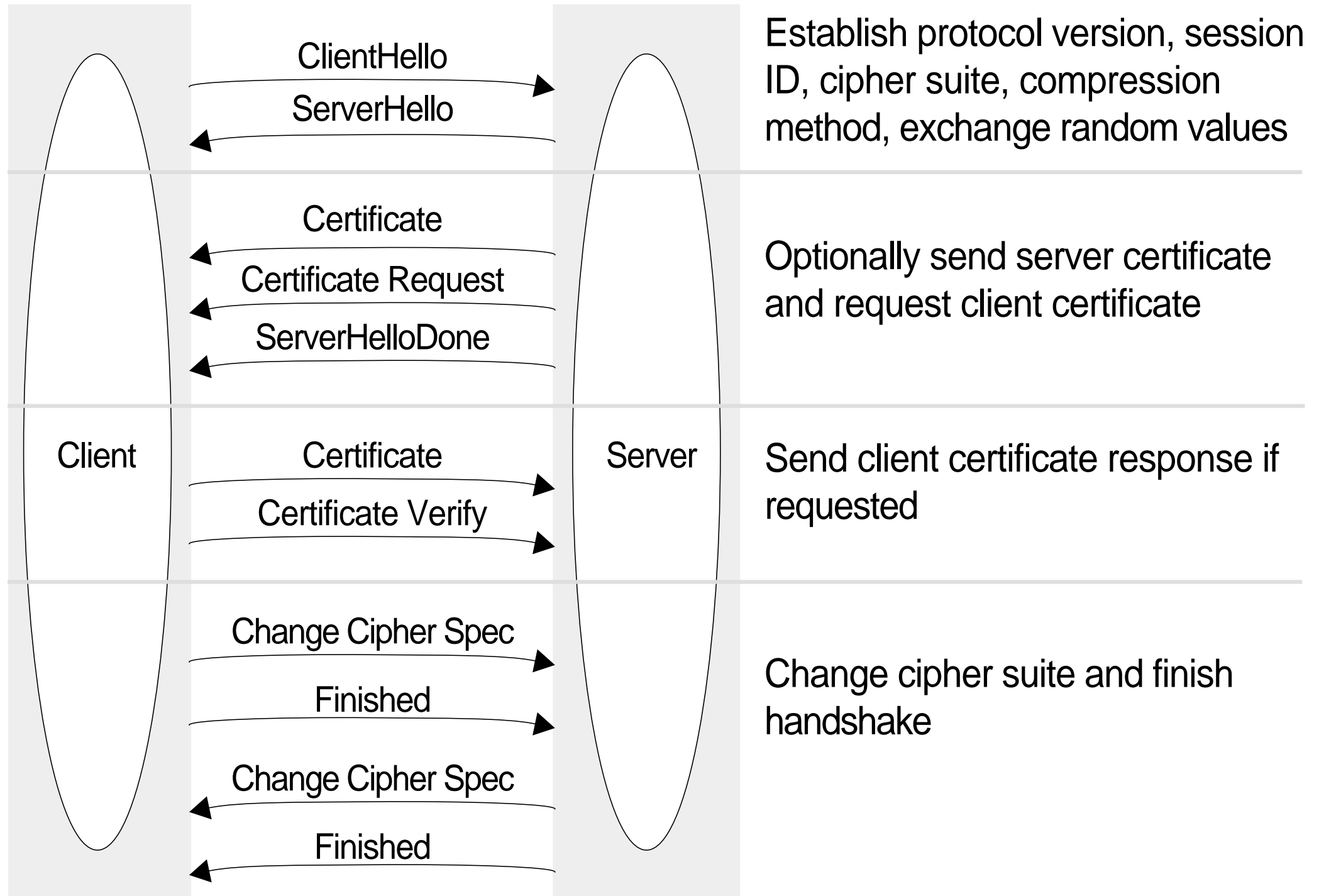
Secure Socket Layer (SSL)

- Avainten jako ja **turvatut kanavat** (alkujaan verkkokauppaan).
- Sisältää **neuvottelut jokaisessa vaiheessa** (salain, tunnistus, todistukset, jne)
 - Hybridi protokolla, julkisen avaimen algoritmit tunnistuksessa ja avaimenvaihdossa, symmetrinen liikenteessä.
 - Alkujaan Netscape Corporation (1994).
 - Laajennettu ja yleistetty Internet standardiksi nimellä **Transport Level Security, TLS** (RFC 2246).

SSL protokollapino [1]



SSL kättely [1]



- Avoin SSL/TLS toteutus.
- Sisältää **C-kirjaston** SSL-toiminnallisuuteen. Lisäksi salaus, tiiviste ja allekirjoitusalgoritmit sellaisenaan.
- Myös **komentorivityökalu** avainhallintaan, salaukseen, purkuun, jne.
- www.openssl.org
- `man openssl`
- esim.

```
openssl enc -bf -in selvä -out salattu
```

```
openssl enc -d -bf -in salattu -out selvä
```

```
openssl dgst -md5 tiedosto
```

- Kirjastot tukevat salaus-, tiiviste- ja varmennealgoritmeja, sekä mm. avaimenvaihto- ja neuvotteluprotokollia.
- Java API ei sinänsä edellytä mitään algoritmia, vaan **ajonaikainen kirjasto** voi tukea mitä haluaa.
- API viittaa käsitteisiin ja toimenpiteisiin.
- Tarkkoihin algoritmeihin ja niiden parametreihin viitataan **nimellä**.

```
Cipher c = Cipher.getInstance("AES");  
c.init(Cipher.ENCRYPT_MODE, avain);  
byte[] salaTeksti = c.doFinal(selvaTeksti);
```

1
2
3

- Erikseen voidaan vielä valita algoritmin tarjoaja (provider), JDK:n mukana tulevat SUN:n toteutukset.
- **SealedObject** -luokka pikasalauksiin:
 - Salaa minkä tahansa *Serializable*-objektin, on itse *Serializable*.
- Cipher{In|Out}putStream: tietovuon salaukseen.
- Kts esimerkki, harjoitukset.

Yhteenveto ja ohjeet turvallisuuteen

- Liittymät ovat **julkisia** (ainakin ennemmin tai myöhemmin).
- Verkot ovat **turvattomia**, osoitteita voidaan väärentää.
- Rajoita kunkin salaisuuden elinaika. Vain avaimet ovat salaisia.
- Hyökkääjät voivat käyttää sinun algoritmeja ja ohjelmia.
- Minimoi turvallisuuskriittinen osa järjestelmästä/ohjelmistosta.

Liittymät ovat alttiina kaikenlaisille hyökkäyksille

- Useimpien hajautettujen järjestelmien liittymät ovat aina avoinna.
- Hyökkääjä voi lähettää minkälaisen viestin tahansa.

Mahdollinen hyökkääjä on **taitavampi** kuin sinä!

- Älä suunnittele (äläkä toteuta) salausalgoritmia itse (ellet ole parempi kuin maailman parhaat ja suurimmat organisaatiot yhteensä).
- Käytä AES, Blowfish, IDEA, (tai (X)TEA).
- Sama pätee protokoliin.

- Resurssien, kommunikaatiokanavien ja liittymien turvaaminen on pakollinen osa hajautetun järjestelmän suunnittelua ja toteutusta.
- Keinoina pääsykontrolli ja turvatut kanavat.
- Julkisen- ja salaisen avaimen salakirjoitusjärjestelmät tarjoavat pohjan turvalliselle kommunikaatiolle.
- SSL/TLS (ja Kerberos) ovat laajasti käytettyjä ja standardoituja mekanismeja.

- Malleja käytetään näyttämään yksinkertaistettu ja abstrakti kuva hajautetusta järjestelmästä jotta voisimme tarkastella kulloinkin tärkeitä asioita.
- **Arkkitehtuurimallit** määrittelevät hajautetun järjestelmän komponentit ja niiden roolit.
- **Vuorovaikutusmallit** käsittelevät järjestelmän aikakäsitettä.
- **Vikamallit** määrittelevät millaisia vikoja voi tapahtua ja mitkä niiden vaikutukset ovat.
- **Turvallisuusmallit** määrittelevät potentiaaliset riskit ja viholliset hajautetussa järjestelmässä.

Luku 4

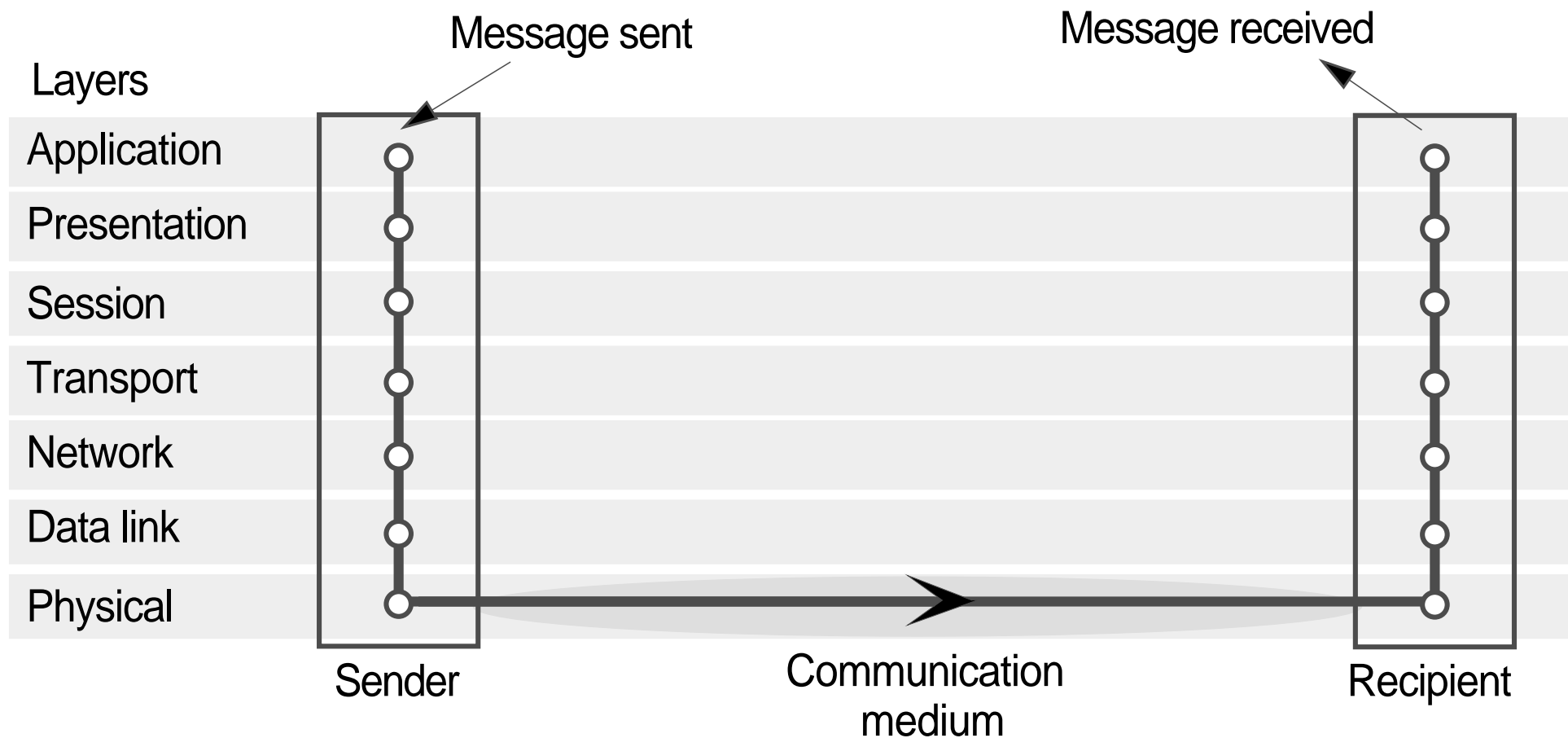
Kommunikaatio hajautetuissa järjestelmissä

- Kommunikaatioarkkitehtuuri: kerrostettu malli (layered) (s. 190)
- Internet -protokollat (IP) (s. 205)
- TCP (ja UDP) ohjelmointi soketeilla (s. 217)
- Etäproseduurikutsu (RPC) ja etämetodikutsu (RMI) (s. 240)

Kommunikaatioarkkitehtuuri: kerrostettu malli (layered)

ISO Open Systems Interconnection (OSI) protokollamalli

191

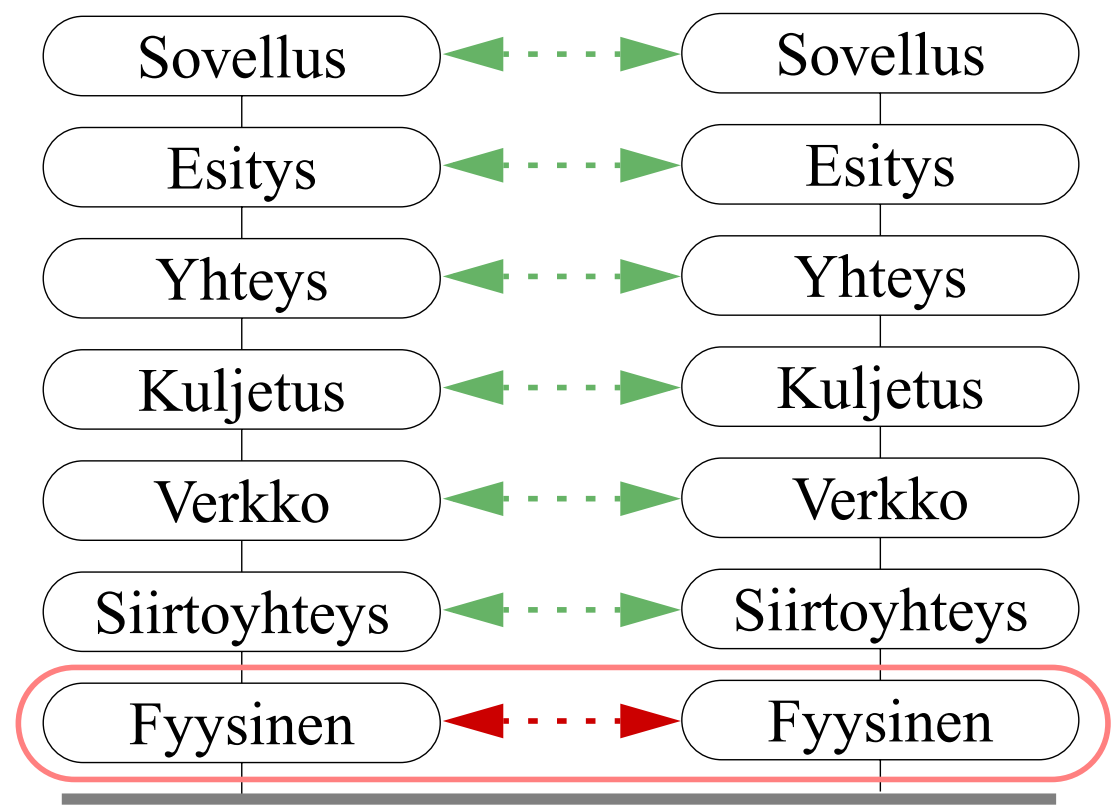


- Jokainen kerros **tarjoaa palveluja yläpuoliselle** kerrokselle.
- Jokainen kerros **laajentaa alemman kerroksen** tarjoamia palveluja.
- Loogisesti viestit/kommunikaatio kulkee tasoa myöten (vaakasuoraan).

Fyysinen kerros (physical)

⇒ Tarjoaa (epäluotettavan) bittiputken.

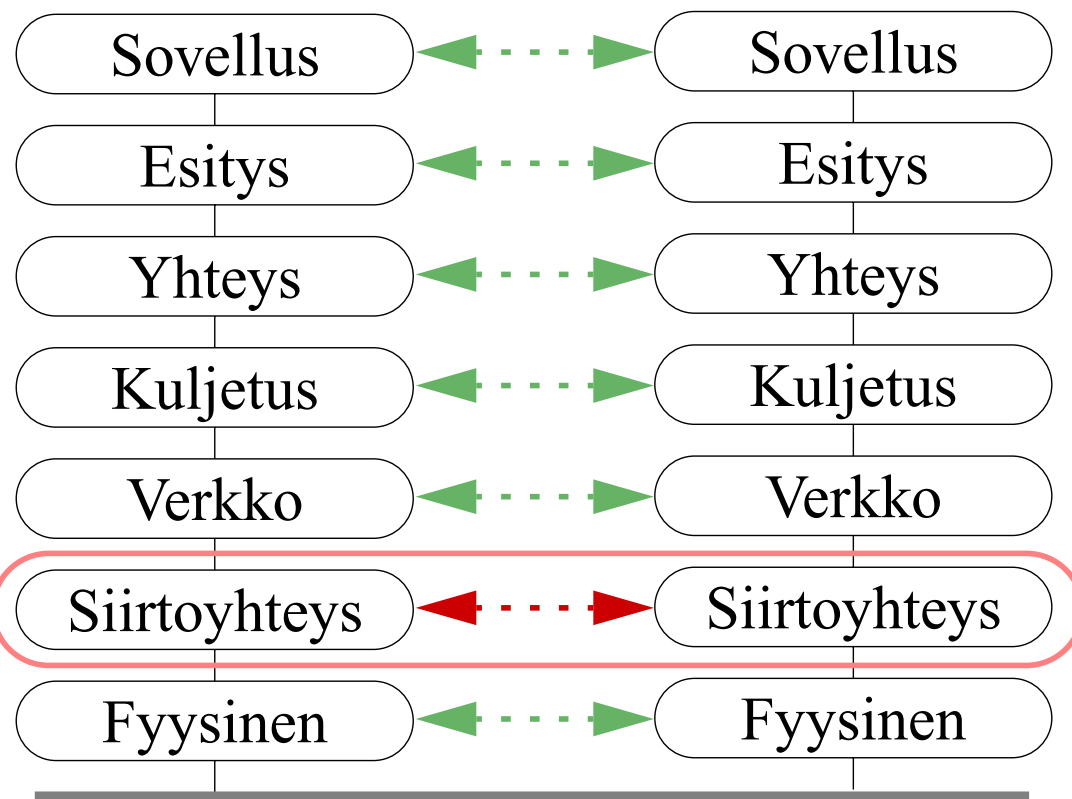
- Laitteelta laitteelle.
- Fyysinen kontakti laitteiden välillä, sähkön, valon, tms. modulointia johtime(i)ssa.
- Kts. tietoliikennetekniikka.
- Esim. Ethernet kantataajuus-signaali, ISDN.



Siirtoyhteyskerros (data link)

⇒ Tarjoaa suoran linkin luotettaville paketeille.

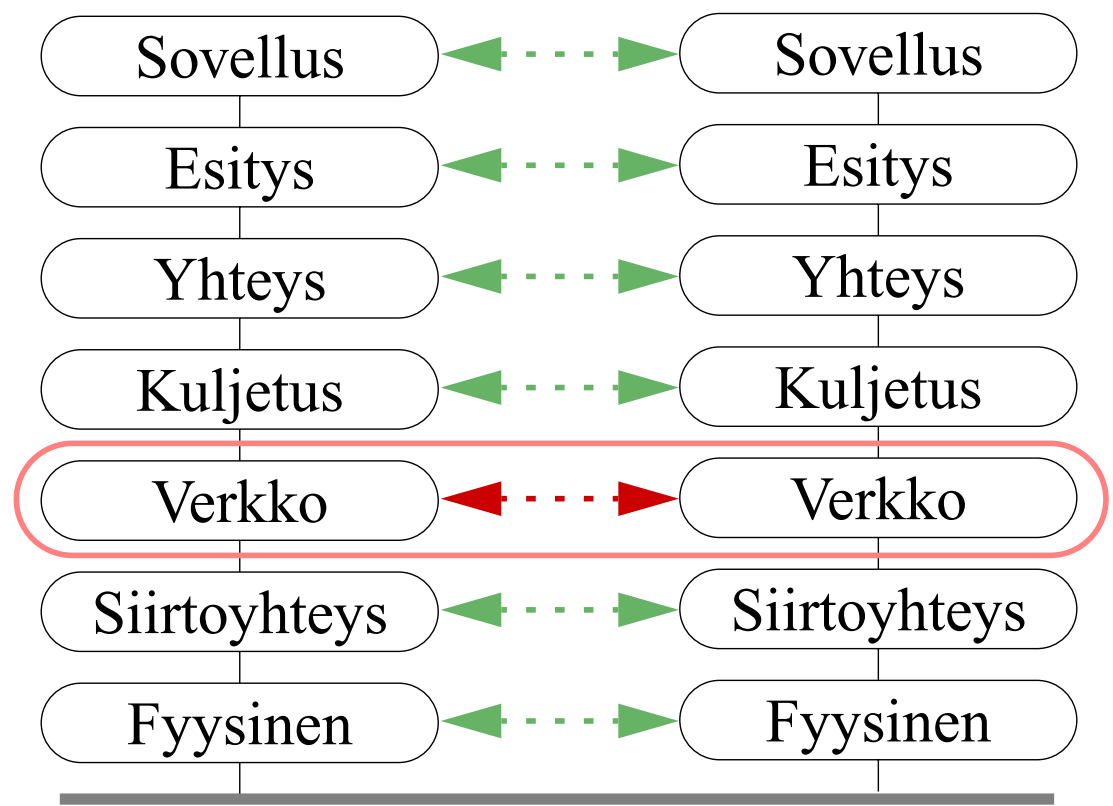
- Solmulta solmulle.
 - Ei vielä reititystä, keskitin/kytkin kytkee **rautaosoitteen** mukaan.
 - Paikallisverkossa tietokoneelta toiselle.
 - Laajassa verkossa tietokoneelta reitittimelle tai reitittimeltä reitittimelle.
- **Paketteja** bittien sijaan.
- Virheen (bittivirhe, törmäys, jne) havaitseminen ja korjaus.
- Esim. Ethernet MAC, ATM cell transfer, PPP.



Verkkokerros (network)

⇒ Tarjoaa päästä-päähän linkin luotettaville paketeille.

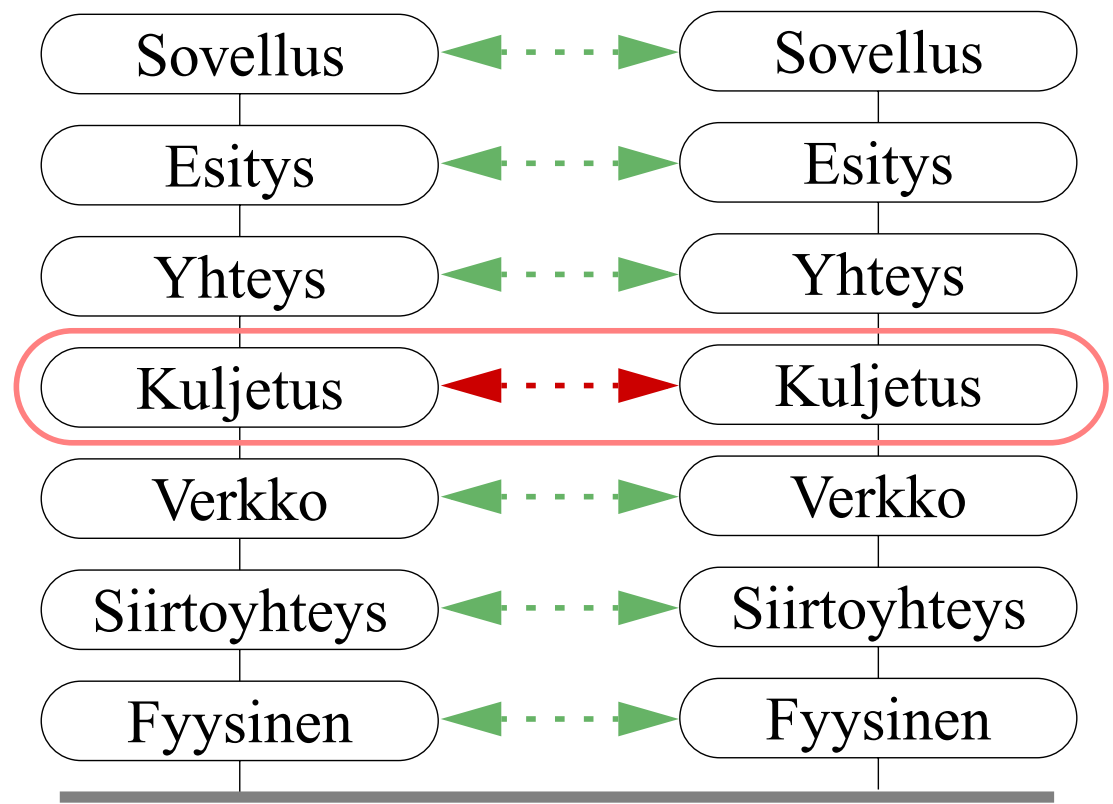
- Koko verkon alueella, tietokoneelta toiselle.
- Reititys solmujen kautta.
- Esim. IP, ATM virtuaalikytkennät.



Kuljetuskerros (transport)

⇒ Tarjoaa prosessilta-prosessille linkin viesteille.

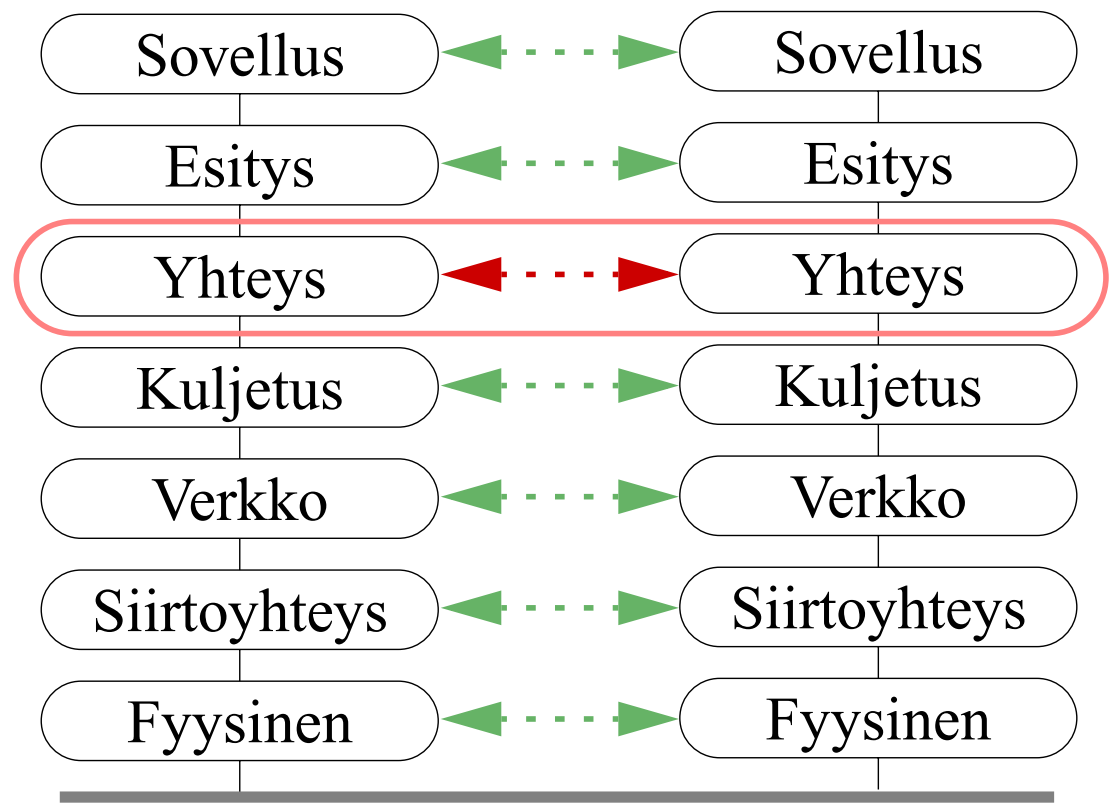
- Tarvittaessa pilkkoo viestit pienemmiksi verkkokerrosta varten.
- Protokollat voivat olla kytkentäisiä (TCP) tai kytkemättömiä (UDP).



Yhteyskerros (session)

⇒ Tarjoaa virtuaalisen yhteyden.

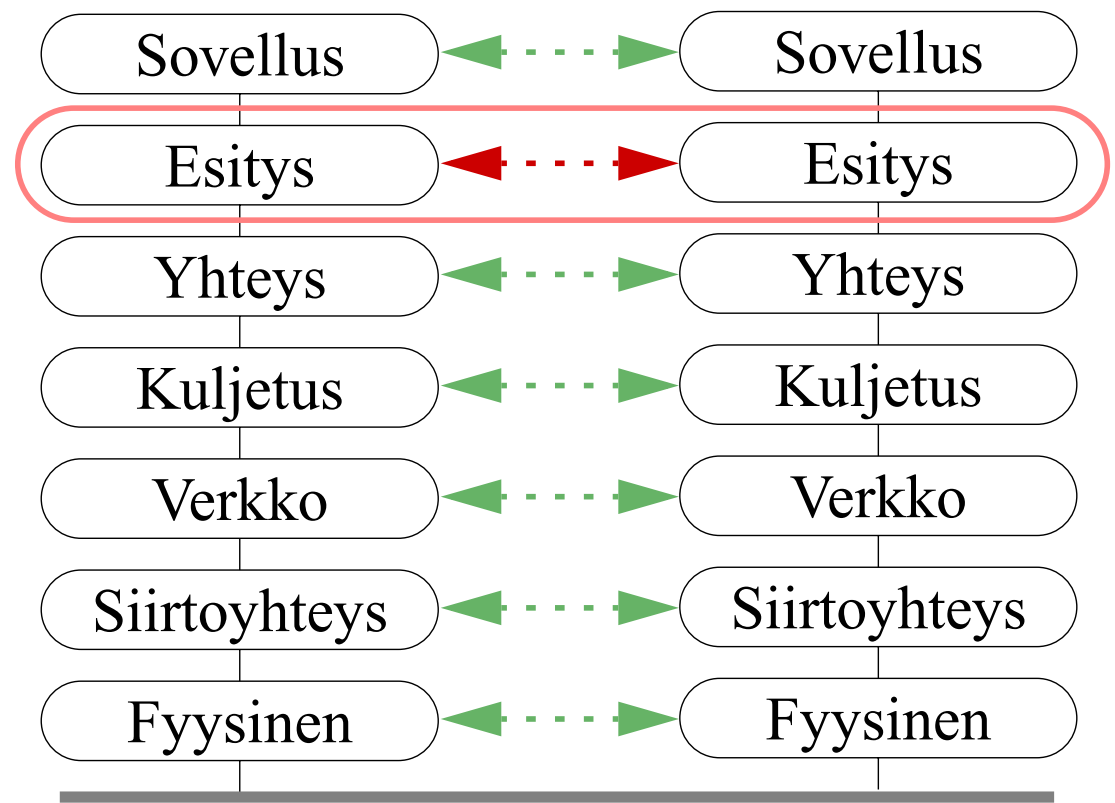
- Lisää luotettavuustekijöitä (esim. automaattinen uuden TCP-yhteyden ottaminen).
- Nimi- ja osoitepalvelut.
- Laskutus.
- Esim. SIP.



Esityskerros (presentation)

⇒ Tarjoaa virtuaaliverkon palveluita.

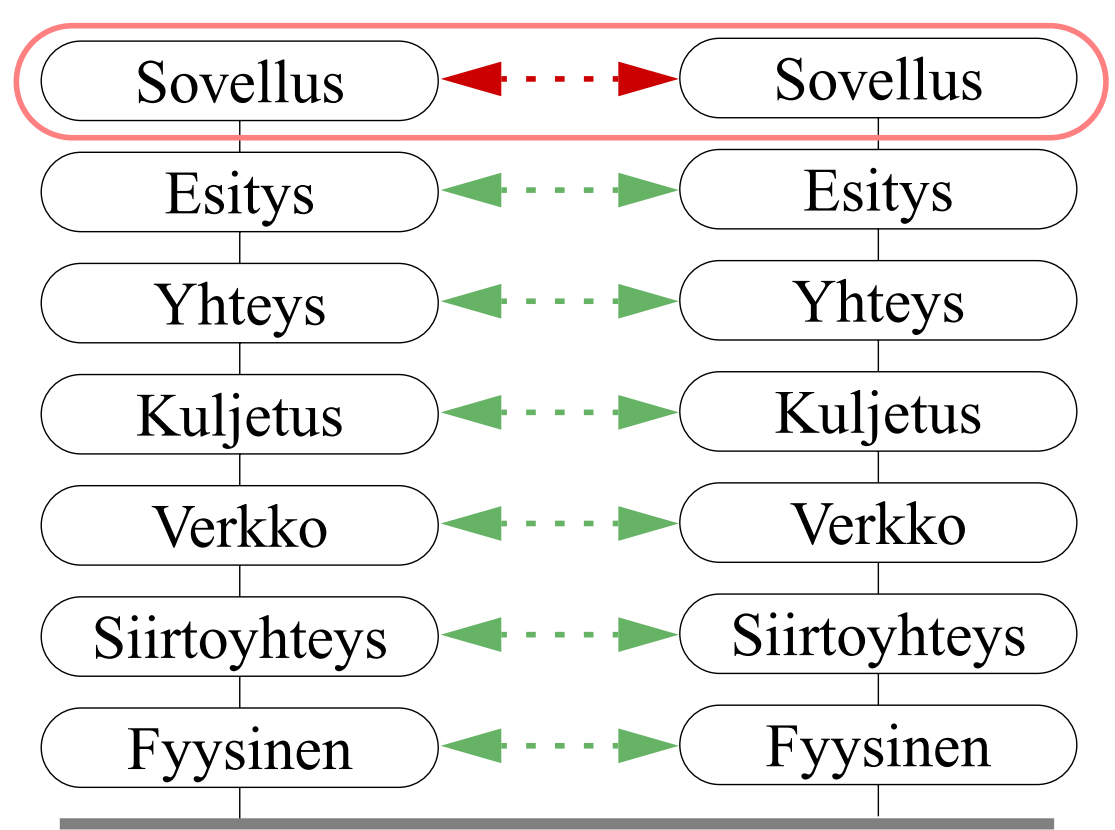
- Esitystapamuunnokset (liukuluvut, merkistöt).
- Tiivistys, salaus.
- Esim. SSL, CORBA Data Representation

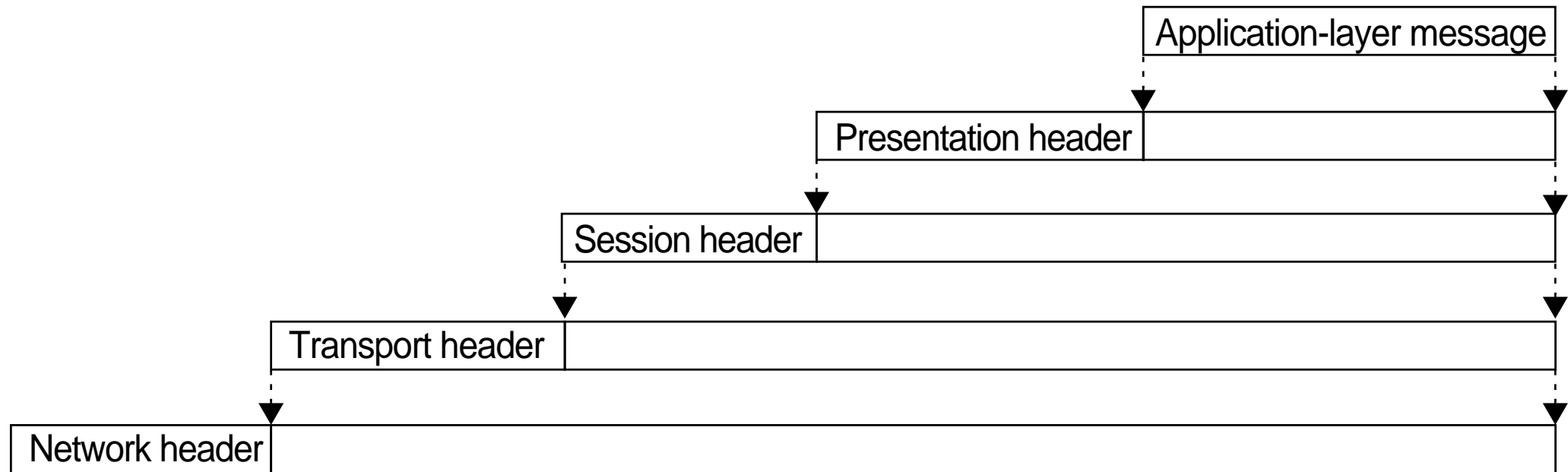


Sovelluskerros (application)

⇒ Tarjoaa sovellusprotokollia.

- **Palvelun** liittymä.
- Esim. HTTP, FTP, SMTP, CORBA IIOP.





- Sovellus lähettää paketin (viestin osan).
- Jokainen protokollakerros **lisää oman otsikkotietonsa** (header) pakettiin ja lähettää sen eteenpäin seuraavalle alemmalle kerrokselle. Joskus myös loppukaneettinsa (footer).
- Vastaavasti vastaanottavassa päässä pakettia "**kuoritaan**" ja tulkitaan kerros kerrokselta.
- Ylempien kerrosten osuuteen (tämän kerroksen kannalta hyötykuormaan) paketista ei normaalisti kosketa.

Kommunikoivat kokonaisuudet (yksiköt)

- Eri tasoilla erilaiset otukset kommunikoivat.
- Verkkolaitteet, tietokoneet, prosessit.

⇒ Meidän tarkoituksiimme, **prosessit** kommunikoivat.

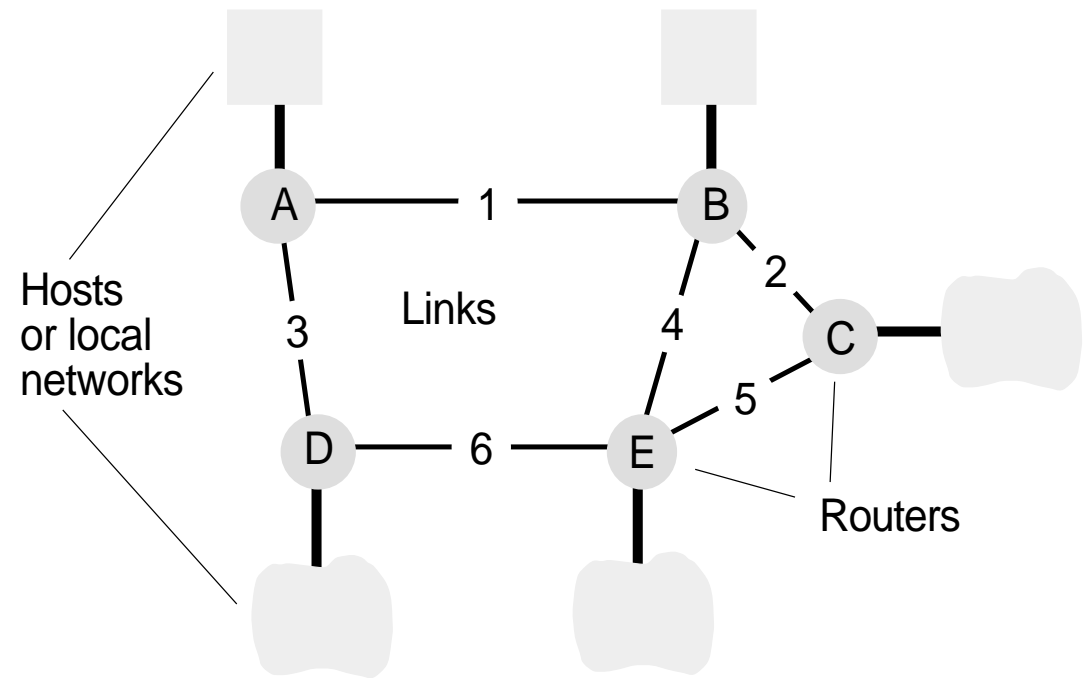
⇒ Osoitteisto on **tietokone:portti**.

- Useimmat tietokoneverkot ovat **pakettivälitteisiä**.
 - Vrt. esim. puhelinverkot ovat **piirikytkentäisiä**.
 - **Virtuaalipiirikytkentä**: yhteyttä muodostettaessa varataan jokaiselta välisolmulta/yhteydeltä kapasiteettia, kukin välisolmu muistaa yhteyden reitityksen (ATM).
 - Madonreikäreititys: viestit reititetään jo viestin (otsikon) ensimmäisten bittien perusteella – viive vähenee.
 - Esim. frame relay (ATM).
 - Jos jossain tulee estettä (toinen paketti), paketti pysähtyy.
 - Paketteja voidaan **puskuroida** väliaikaisesti välisolmuissa kunnes seuraava siirtotie vapautuu.
 - Reititys määrää **kunkin paketin reitin erikseen**.
 - Yksittäinen alemman tason paketti voi kadota, ylempi taso huolehtii virheiden havaitsemisesta ja korjauksesta.
 - Kommunikaatio on epäsynkronista.

Reititys (routing)

⇒ Reititystä tarvitaan kaikissa laajemmissa verkoissa.

- Jokaisen solmun pitää osata välittää mikä tahansa paketti **oikeaan suuntaan** tarkastelemalla paketin otsikkotietoja (kohdeosoitetta).
- Jollei suoraan lopulliseen osoitteeseen, niin oikeaan suuntaan.
- Adaptiivinen reititys mukautuu verkon tilaan (muutoksiin).
- Säännölliset paikalliset **reititystaulun** päivitykset.



- Sovellukset vaihtavat tietoa **viesteinä**.
 - Sovelluksen loogisen viestin pituus voi olla suuri (rajoittamaton).
- Tietoverkoissa **paketin koko on rajoitettu**, joskus jopa kiinteä.
 - Helpottaa välisolmujen toteutusta (**puskurit**, resurssien **aikajako**).
- Pitkät viestit **jaetaan osiin** alemmissa kerroksissa (fragmentation).
 - Lyhyt viesti välitetään yhdessä paketissa.

- Jotkin sovellukset vaativat tasaisen **tietovirran** lähteestä kohteeseen.
 - Esim. (video)puhelu, pääteyhteys.
- Tämä voidaan toteuttaa piirikytkennällä (varatulla kaistalla) tai sopivilla paketeilla ja ylemmän kerroksen protokollalla.
- Yleensä asynkronisilla paketeilla toteutettu vuo on riittävä, kts. harj 7.
- Eniten käytetty **TCP itse asiassa tarjoaa vuon** eikä paketteja.
 - Monet erillisiä viestejä käyttävät sovellukset ja protokollat ovat päinvastaisen ongelman edessä, kuinka pilkkoa **vuo paketeiksi!**
 - Pakettien lähettäminen vuohon on helppoa.
 - Vastaanottajan on **erotettava eri viestit toisistaan**.
 - Palataan tähän myöh., TCP Protokollaohjeita [1, 5] (s. 225).

Internet -protokollat (IP)

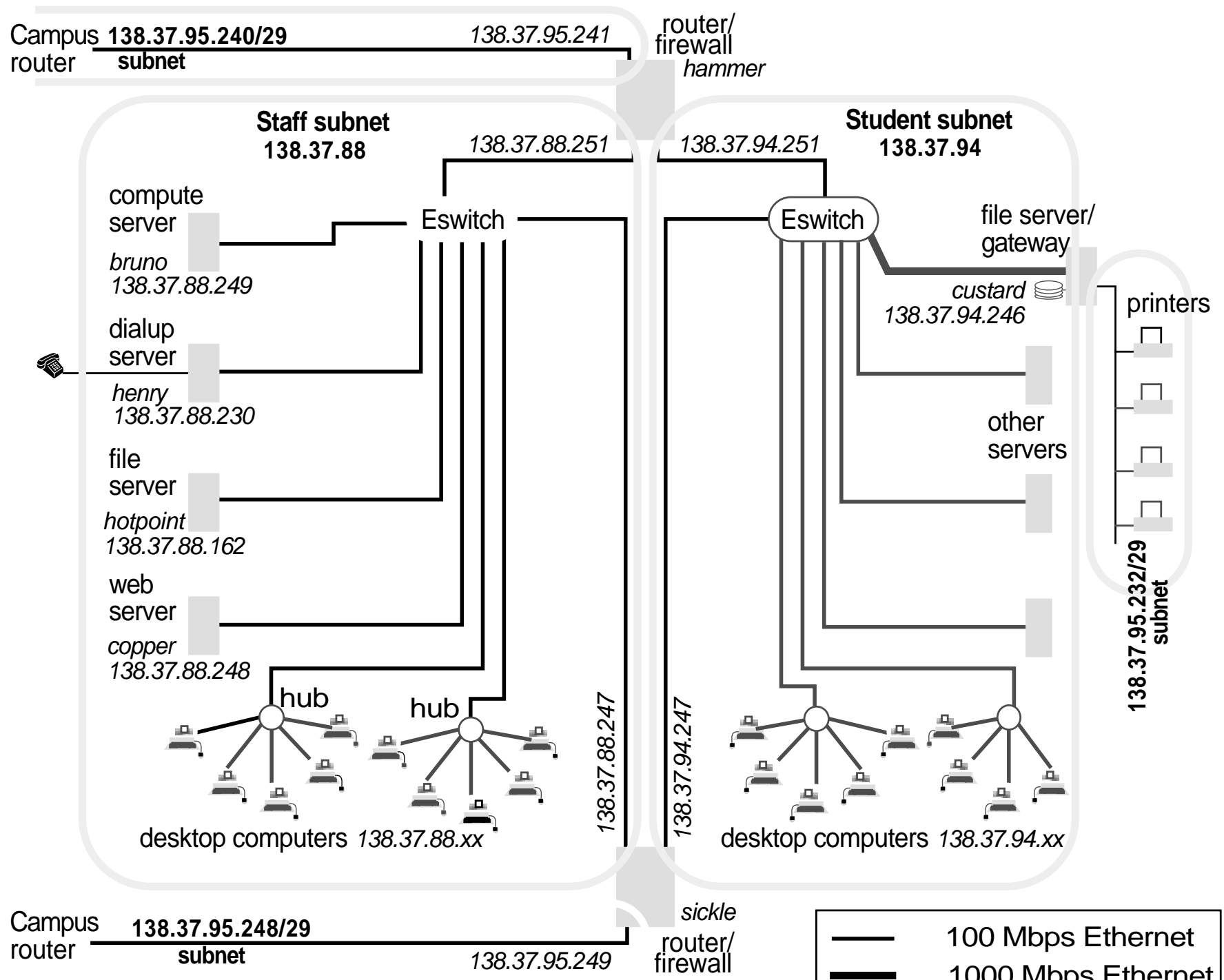
⇒ Pääosa maailman hajautetuista järjestelmistä rakennetaan joko suoraan käyttämään IP protokollia tai väliohjelmistoja (jotka tukevat myös IP:tä).

- Yhtenäinen **osoitejärjestelmä** (oli, ehkä joskus tulee olemaan).
- Yhtenäinen **reititys**.
- Yhtenäiset **sovellusprotokollat** (useimmille yleisille sovelluksille).

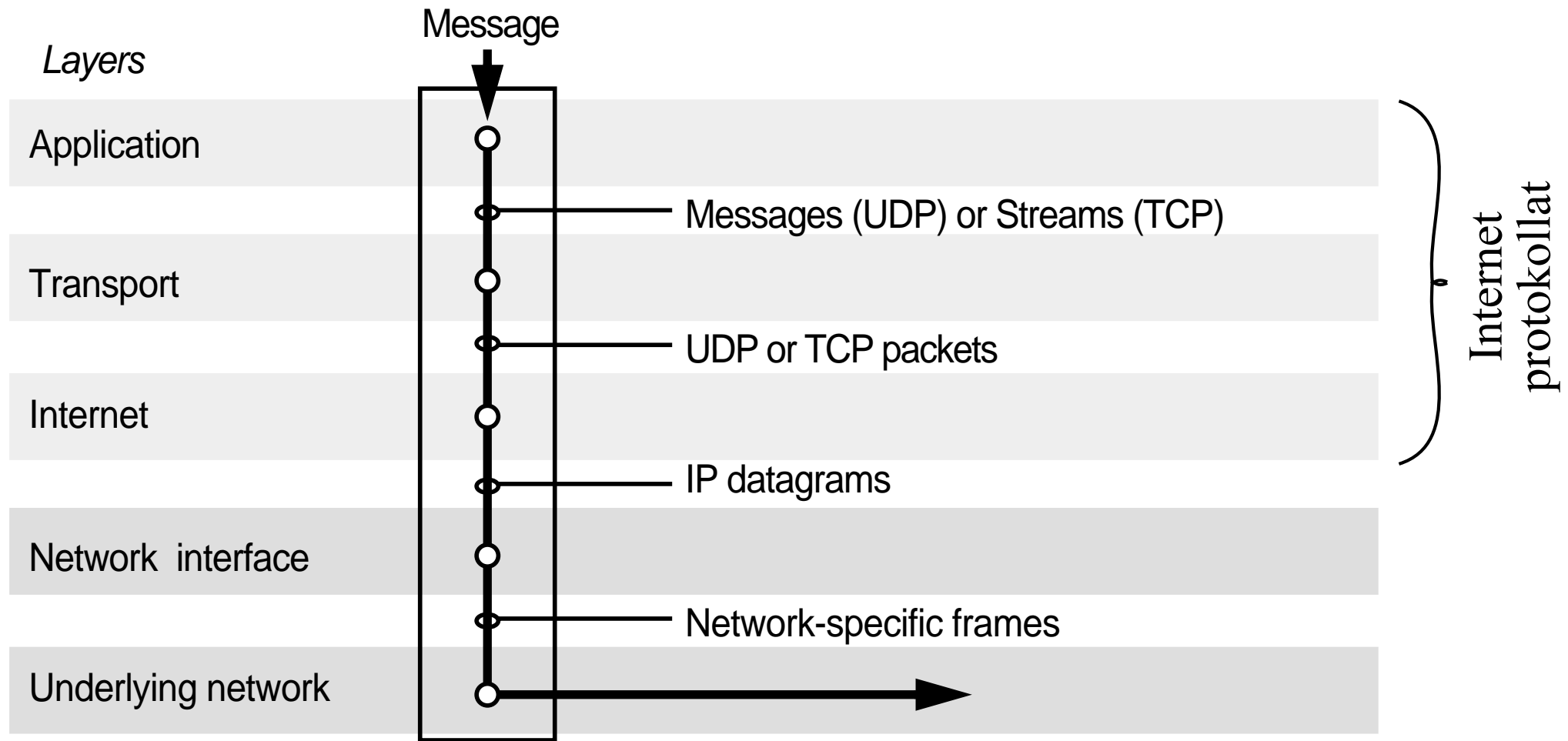
⇒ Erityisesti TCP/IP on ollut menestys!

Figure 3.10 Simplified view of the QMW Computer Science network

Internet -protokollat (IP)



- TCP (Transport Control Protocol) and UDP (User Datagram Protocol) ovat molemmat **kuljetuskerroksen** (transport) protokollia jotka on toteutettu **verkkotason** IP protokollan päälle.
- Internet -protokollat eivät täysin toteuta OSI-mallia.



- IP ei ota kantaa alempiin kerroksiin.
- **IP** tarjoaa **osoitteet** ja **reitityksen**.
 - Erilliset paketit tietokoneiden välillä.
 - Parhaansa tehden (best-effort) semantiikka.
 - Ei lupaus perillemenosta, ei uudelleenyrityksiä.
 - Alempien kerrosten protokollat (esim. Ethernet) toki voivat käyttää uudelleenlähetyksiä tai muita luotettavuustoimia.

	octet 1	octet 2	octet 3	
	<i>Network ID</i>		<i>Host ID</i>	
Class A:	1 to 127	0 to 255	0 to 255	0 to 255
	<i>Network ID</i>			<i>Host ID</i>
Class B:	128 to 191	0 to 255	0 to 255	0 to 255
	<i>Network ID</i>		<i>Host ID</i>	
Class C:	192 to 223	0 to 255	0 to 255	1 to 254
	<i>Multicast address</i>			
Class D (multicast):	224 to 239	0 to 255	0 to 255	1 to 254
Class E (reserved):	240 to 255	0 to 255	0 to 255	1 to 254
				<i>Range of addresses</i>
				1.0.0.0 to 127.255.255.255
				128.0.0.0 to 191.255.255.255
				192.0.0.0 to 223.255.255.255
				224.0.0.0 to 239.255.255.255
				240.0.0.0 to 255.255.255.255

- B-luokan osoitteiden loppumisen vuoksi nykyään käytössä paljolti "luokaton" osoitteisto (Classless interdomain routing (**CIDR**)):
 - Aliverkko määritellään **bittipeitteellä** (netmask), esim. 255.255.248.0 määrittää aliverkon jossa 11 alinta bittiä ovat käytössä, eli käytössä on 2046 osoitetta.
 - Merkitään esim. 193.162.248.0/21 (21-bittiä kiinnitetty, 11 vapaana)

- **Rekisteröimättämät** osoitteet ja osoitteenmuunnokset (NAT)
 - Erillisissä sisäverkossa voidaan käyttää samoja osoitteita.
 - Tähän käyttöön on varattu:
 - 10.0.0.0 - 10.255.255.255 (10/8)
 - 172.16.0.0 - 172.31.255.255 (172.16/12)
 - 192.168.0.0 - 192.168.255.255 (192.168/16)
 - Verkon **yhdyskäytävä** (gateway) **muuttaa ulosmenevien pakettien osoitteet** julkisen verkon mukaiseksi ja sisääntulevien pakettien osoitteet sisäverkon mukaisiksi.
 - Yhdyskäytävä -nimeä käytetään myös **ulkoverkkoon johtavasta reitittimestä** vaikkei osoitteita muunnettaisikaan,
 - Sisäverkon osoitteet jaetaan usein dynaamisesti (DHCP).
 - **DHCP** antaa myös muita verkon konfigurointitietoja (nimipalvelun osoite, oletusyhdyskäytävä, jne).

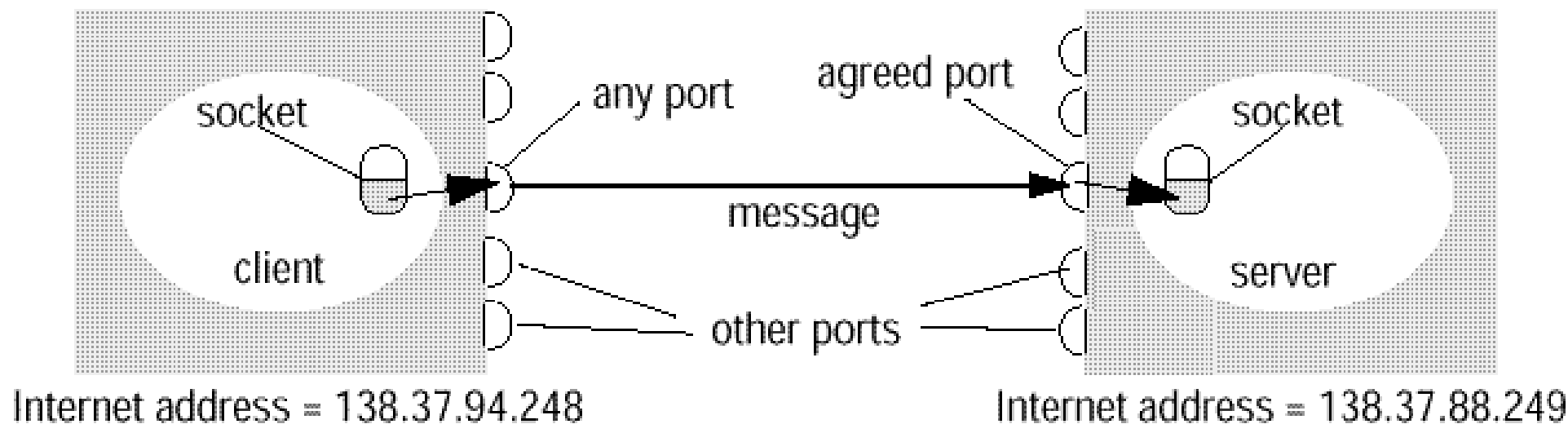
- **Nimipalvelu** (Domain Name Service, DNS) muuttaa tekstimuotoisen osoitteen numeromuotoiseksi.
 - Hierarkkinen järjestelmä, jos lähin palvelin ei tunne osoitetta, se kysyy ylempää.
 - Juuripalvelimet pyytävät tarkkaa osoitetta vuorostaan alemppaa.
- **2/3**-kerroksen reitityslaitteet
 - 2-tason laite yrittää oppia kaikki aliverkossa olevat laiteosoitteet ja reitittää niiden mukaan.
 - 3-tason laite reitittää IP-osoitteen mukaan.
 - Vrt. postinjakaja tuntee nimet vs. käyttää osoitteita.

- **Yhdyskäytävä** (gateway)
 - Kullekin tuntemattomalle osoitteelle voidaan määritellä mihin sinne kuuluva paketti lähetetään.
 - Yleensä paikallisverkossa on vain yksi oletusyhdyskäytävä jonne kaikki paikallisverkkoon kuulumattomat paketit lähetetään.
 - Käytetään **myös eri protokollien** välillä reitittävistä solmuista.
- **PPP** (point-to-point protocol)
 - Siirtää IP-paketteja muussa kahden pisteen välisessä verkossa.
 - Esim. paikallisverkon kytkentä puhelinmodeemilla toiseen verkkoon.
- **IPv6** – seuraava versio IP-protokollasta
 - 128-bittinen osoiteavaruus
 - IP-tasolla vähemmän prosessointia (ei tarkastussummia, eikä pakettien pilkkomista (fragmentation))
 - "Priorisointi" / pakettien luokitus.
 - Vuonmerkintä (flow label) mahdollistaa sovellukselle varatun kaistan.
 - IPv6 paketin otsikkoa (header) voidaan laajentaa (*next header*).

- Tarjoavat **kommunikaatioprimitiivit sovelluksille** (ja väliohjelmistoille).
- **Prosessilta prosessille** kommunikaatio (porttien kautta).
- Portti on 16-bittinen **osoite tietokoneen sisällä**.
 - Esim. Tiedepuistolla on monta postilaatikkoa.
 - Suomen Posti tuo postia infoon.
 - Info jakaa postin eri laatikoihin.
- Tietokoneella on (yleensä) yksi osoite.
 - Tietokoneessa voi olla monta postilaatikkoa (porttia).

- Kuljetuskerros välittää viestin oikeaan porttiin paketin otsikkotietojen ohjaamana.

- Jokainen käytössä oleva **portti on kytketty johonkin prosessiin**.
- Prosessi näkee portit kahvoina (pistokkeina, soketteina, socket) jotka ovat kuin **tiedostokuvaajia**.



TCP (Transport Control Protocol)

⇒ TCP takaa vastaanottavan prosessin saavan kaiken lähetävän prosessin lähettämän tiedon samassa järjestyksessä.

- Ennen tiedon välittämistä on (asiakkaan) otettava **TCP-yhteys**.

Requesting a connection

Listening and accepting a connection

```
s = socket(AF_INET, SOCK_STREAM, 0)
•
•
connect(s, ServerAddress)
•
•
write(s, "message", length)
```

```
s = socket(AF_INET, SOCK_STREAM, 0)
•
bind(s, ServerAddress);
listen(s, 5);
•
sNew = accept(s, ClientAddress);
•
n = read(sNew, buffer, amount)
```

- Yhteys voi luonnollisesti katketa (tai sitä ei saada edes kytkettyä).
- Luotettavuuden saavuttamiseksi TCP mm. **numeroi** yhteyden paketit, **kuittaa** säännöllisesti vastaanotetut paketit, lähettää tarvittaessa paketteja **uudestaan**, **puskuroi** vielä kuittaamattomat paketit, säätää **lähetysnopeutta** ja käyttää **tarkistussummia**.

UDP (User Datagram Protocol)

- UDP lähettää yksittäisen viestin toiseen osoitteeseen.
- Käytännössä vain kuljetuskerroksen versio IP:stä, mahdollisimman pienet lisäkustannukset (eikä juuri lisätoiminnallisuutta).
- Tehokas, muttei luotettava.
- Erityisesti ylikuormitustilanteessa ero TCP:hen näkyy:
 - Ylikuormituksessa satunnaiset IP-paketit pudotetaan.
 - TCP lähettää pudonneet paketit uudelleen ja hidastaa IP pakettien lähettämistä, vuo säilyy ehjänä.
 - UDP paketteja putoaa, lähettäjä ei tiedä siitä mitään.
- Jos luotettavuutta halutaan parantaa, se voidaan tehdä sovellustasolla.
- IP-monilähetys toimii kuten UDP.

TCP (ja UDP) ohjelmointi soketeilla

- Soketti = pistoke = kahva = socket.
- Samat mekanismit kaikilla kielillä ja alustoilla, vain syntaksit vaihtelevat (ja tarvittavien detaljien taso).
- Kommunikaatio yhteensopivaa ristiin kaikille alustoille.
 - Kunhan tietosisällön yhteensopivuudesta huolehditaan.
- Pikakurssi Javalla, lähinnä muokataan valmiita esimerkkejä.
- Kts myös Java tutorial, custom networking.
- Esimerkkejä myös C, Perl, Python.

⇒ Avattu TCP soketti on kaksisuuntainen yhteys toiseen prosessiin.

- Erilliset luokat asiakkaalle ja palvelimelle:
 `java.net.ServerSocket` ;
 `java.net.Socket` ;
- Kytkettäessä tarvitaan osoite.
 - Palvelimella **paikallinen portti** johon se odottaa uusia yhteyksiä.
 - Asiakkaalla **palvelimen osoite ja portti** johon kytkeydytään.
- (Kytketyn) soketin tärkeimmät ominaisuudet ovat **tietovirta molempiin suuntiin** (*InputStream* ja *OutputStream*).
- Lisäksi voidaan tarkastella tilaa ja sulkea yhteys (molemmat suunnat erikseen).

Muutamia vaihtoehtoja käyttää näitä

- **Kytkeminen luotaessa** tai myöhemmin.
- Osoitteena merkkijono-osoite tai *InetAddress*.
- *InputStream*:ia ja *OutputStream*:ia voidaan käyttää monella tavalla (apuluokalla).
 - *PrintWriter*, *BufferedReader*, *BufferedInputStream*, *Scanner*, *ObjectOutputStream*, jne.

ServerSocket.accept()

- Palvelimen *accept()* metodi **odottaa uutta asiakkaan yhteydenottoa** ja sellaisen tultua palauttaa **uuden soketin**.
- Vanha soketti jää **odottamaan uusia yhteydenottoja** (jotka saadaan taas käyttöön *accept():llä*).
- **Uudesta soketista** otetaan käyttöön *Input/OutputStream:t* asiakkaan kanssa keskustelua varten.
- Yhteyden lopuksi uusi soketti suljetaan.
- Jotta palvelin olisi samanaikainen, uusi soketti annetaan uudelle säikeelle tai prosessille.

Perusmalli (tekstipohjaiselle protokollalle)

- Asiakas
 - Yhteydenotto:
`Socket kahva = new Socket(palvelinosoite, portti);`
 - Tiedon lähettäminen:

```

PrintWriter ulosvirta =
    new PrintWriter(kahva.getOutputStream(), true);
ulosvirta.println(lähetettäväTekstiRivi);

```

- Vastaava palvelin

- Palvelimen avaus:

```

palvelinkahva = new ServerSocket(portti);

```

- Uuden asiakkaan yhteydenoton hyväksyminen:

```

Socket asiakaskahva = palvelinkahva.accept();

```

- Tiedon lukeminen asiakkaalta:

```

syote = new BufferedReader(new InputStreamReader(
    asiakaskahva.getInputStream()));
String viestiRivi = syote.readLine();

```

- Kts. *AsiakasEsimerkki*, *PalvelinEsimerkki*, *PalvelinSaieEsimerkki*.

- *Buffered{In|Out}putStream* lukee/kirjoittaa tavuja.

```
out = new BufferedOutputStream(kahva.getInputStream());
in = new BufferedInputStream(kahva.getInputStream());
out.write(byte[]); // koko taulukko
int c = in.read(); // tavu kerrallaan
in.read(byte[] b, 0, min(b.length, in.available()));
```

- *out.flush()* varmistaa kirjoituksen.
- Kts. MD5 esimerkit, Java API.
- Protokolla määriteltävä tarkasti, kts. Binääri (tavuvirta) (s. 228).
- Tavujen käyttö Javalla hieman työläämpää kuin esim. C:llä sillä sisäistä esitysmuotoa ei haluta näyttää.
 - Paitsi luokilla joilla on *toByteArray()* metodi, mutta silloinkin esitysmuoto voi olla melkein mitä vain.
 - *java.nio.ByteBuffer* tarjoaa käyttökelpoisen pohjan.

Objektiprotokollalle

- Sarjallistuva (*Serializable*) olio voidaan kirjoittaa näppärästi *ObjectOutputStream*:iin (*writeObject()*).

- Vastaavasti lukeminen *readObject()*.

```
ObjectOutputStream oOut =
    new ObjectOutputStream(kahva.getOutputStream());
oOut.writeObject(olio);
```

```
ObjectInputStream oIn =
    new ObjectInputStream(kahva.getInputStream());
Object o = oIn.readObject();
```

- Edellyttää molempiin päihin **samaa Javan versiota**.
 - Kts *SalaOlioXxxxEsimerkki.java*

Vuon salaus

- TCP-virta voidaan **salata** helposti käyttäen *Cipher{In|Out}putStream*.
 - Käytettävä vuosalainta (lohkosalainta vuo -tilassa) ja 8-bittisiä lohkoja, kts. harjoitukset.

⇒ Aina ei haluta suorituksen (edes säikeen) pysähtymään ("ikuisesti") odottamaan yhteyttä (*accept()*) tai tietoa (*read()*).

- Yleensä pysähtyvät säikeet ovat ok.
- *Selector()* -luokalla voidaan tarkastaa mitkä siihen kytketyt kanavat (esim. soketit) ovat luku/kirjoitusvalmiita (mm. *accept()*).
 - Käyttö hieman työläämpää kuin C:n *select()* (josta esimerkki [www-sivulla](#)).
- *BufferedReader.ready()* voidaan käyttää testaamaan josko jotain olisi valmiina (*.read()*)
- Soketille voidaan antaa aikakatkaisu millisekunteina *setSoTimeout()* metodilla, jolloin *accept()* ja *read()* antavat poikkeuksen (*SocketTimeoutException*) ellei mitään kuulu. Kts. esimerkki *PalvelinEsimerkkiAikaKatkaisu.java*

Protokollan tehtävät

- Viestin **koodaaminen** siirtoa varten (encoding)
- Viestien **erottaminen** toisistaan (framing)
- **Raportointi** (reporting)
- Useiden viestien asynkronisuus (**liukuhihnaus**, pipelining)
- Asiakkaan ja palvelimen **varmennus** (authentication)
- **Yksityisyys** (privacy)

Lisäksi muistettava

- **Skaalattavuus**
- **Tehokkuus**
- **Yksinkertaisuus** (K.I.S.S.)
- **Laajennettavuus** (versioiden yhteensopivuus)
- **Vakaus** (virheistä toipuminen)

- Etsi **olemassaoleva protokolla** joka tekee (suunnilleen) mitä tarvitset.
- Määrittele protokolla **HTTP** tai **SMTP** (toiminnallisuuden) **päälle**.
- Määrittele sopiva uusi protokolla **alusta alkaen**.

HTTP:n käyttö

- Tuttu.
- Voit käyttää HTTP-palvelinta ja skriptejä.
- Työkalut (kirjastot) useimmilla kielillä ja alustoilla, myös JME.
- Voit jopa käyttää porttia 80 (jonka palomuurit päästävät usein läpi).
 - Älä kuitenkaan kierrä tietoturvasääntöjä (edes vahingossa).
- Sopii hyvin pieni pyyntö-suuri vastaus -malleihin.
- Kts. XMLRPC (<http://www.xmlrpc.com/>) (s. 256).

Tiedon esitysmuoto

- **Teksti** (merkkivirta)
 - **Sovittu merkistö** (7-bit ASCII, UTF-8, jne), sovittu tapa esittää luvut.
 - MIME (tai uuencode) koodaa muu kuin ASCII-data.
 - Vie enemmän tilaa.
 - **Selaaminen** hitaampaa.
 - Helpompi tutkia (debugata).
 - **Rakenteen** koodaamismahdollisuuksia:
 - Sovittu kenttien järjestys (ei kovin laajennettava)
 - **Komento**/otsikkopohjainen
 - **Merkintäkieli** (XML)
 - Sovitut erottimet (1+ väliä, rivinvaihto: CR LF (\r\n) (\015\012))
 - Kts RFC 822.

- **Binääri** (tavuvirta)
 - Sovittu arvojen **binääriesitys**
 - Merkitsevin tavu viimeisenä (big endian, **network byte order**)
 - Käytä unsigned, jollei etumerkkejä tarvita.
 - Vältä liukulukuja (tai käytä IEEE Standard 754)
 - Kenttien pituus määritellään tavuina (tai bitteinä).
 - **Viestin pituus** ensimmäisenä kenttänä (muista mainita kuuluuko se pituuteen vai ei).
 - **Ei erottimia** tai tägejä.
 - **Nopea** selata.
 - **Tiivis**.
 - Vaikea virheenetsintä.
 - Tasaa sanat sanarajalle (ja puoli ja kaksois) (jollei tila ole kallista).
 - Älä oleta minkään alustan tietueen sisäisestä tallennustavasta mitään (älä siis lähetä sitä verkkoon).

⇒ Kolme vaihtoehtoa

- Viestin **loppumerkki**
 - Viesti on selattava vastaanoton aikana.
 - Varauduttava pitkiin viesteihin.
 - Jos loppumerkki esiintyy viestissä, on se koodattava.
 - Esim, SMTP: yksinäinen piste (.) rivillä.
- Viestin **pituus** alussa (otsikkotiedoissa)
 - Koko viestin (pituuden) on oltava valmiina otsikoita lähetettäessä.
 - Voidaan välttää paloittelulla (segmentation)
 - Vastaanottaja voi varata muistia sopivasti.
 - Esim, HTTP 1.1, IMAP
- **Yhteyden sulkeminen** jokaisen viestin jälkeen.
 - Esim., HTTP 1.0, FTP
 - Helppo.
 - Tehoton jos useita lyhyitä viestejä.

Raportointi

- Raportoi tuloksesta aina (ok, tila, varoitus, virhe).
- 3-numeroinen koodaus (RFC 821 SMTP: E: **Theory of Reply Codes**):
- Lisää tietoa voi seurata koodin perässä (yleensä selväkielisenä).

Liukuhihnaus

- Sallii asiakkaan tehdä useita pyyntöjä palvelimelle peräkkäin odottamatta vastauksia.
- Pyyntöt silti käsitellään ja raportoidaan **järjestyksessä**.
- Vähentää verkkoviiveen vaikutusta, kasvattaa pakettikokoa.
- Esim., HTTP1.1 pysyvät (persistent) yhteydet, komentojen liukuhihnaus (pipelining) SMTP:ssä.
- **TCP** varmistaa oikean järjestyksen.

Esim: SMTP (RFC 821)

```
C: <client connects to service port 25>
C: HELO snark.thyrsus.com           sending host identifies self
S: 250 OK Hello snark, glad to meet you receiver acknowledges
C: MAIL FROM: <esr@thyrsus.com>     identify sending user
S: 250 <esr@thyrsus.com>... Sender ok receiver acknowledges
C: RCPT TO: cor@cpmy.com            identify target user
S: 250 root... Recipient ok        receiver acknowledges
C: DATA
S: 354 Enter mail, end with "." on a line by itself
C: Scratch called. He wants to share
C: a room with us at Balticon.
C: .                                 end of multiline send
S: 250 WAA01865 Message accepted for delivery
C: QUIT                             sender signs off
S: 221 cpmy.com closing connection  receiver disconnects
C: <client hangs up>
```

```
S: <wait for open connection>
C: <open connection to server>
S: 220 innosoft.com SMTP service ready
C: HELO dbc.mtview.ca.us
S: 250-innosoft.com
S: 250 PIPELINING
C: MAIL FROM:<mrose@dbc.mtview.ca.us>
C: RCPT TO:<ned@innosoft.com>
C: RCPT TO:<nsb@thumper.bellcore.com>
C: RCPT TO:<kvc@innosoft.com>
C: DATA
S: 250 sender <mrose@dbc.mtview.ca.us> OK
S: 250 recipient <ned@innosoft.com> OK
S: 550 remote mail to <nsb@thumper.bellore.com> not allowed
S: 250 recipient <kvc@innosoft.com> OK
S: 354 enter mail, end with line containing only "."
    . . .
C: .
C: QUIT
S: 250 message sent
S: 221 goodbye
```

- TCP yhteys

HTTP 1.0

- Asiakas antaa komentoja (GET, POST)
- Pyyntö loppuu tyhjään riviin

```
GET /hakemisto/tiedosto HTTP/1.0
[Optio: arvo]
```
- Palvelin vastaa statuksella (optioilla) ja dokumentilla
- otsikot ja dokumentti **erotettuna tyhjällä rivillä**

```
HTTP/1.0 200 OK
[Optio: arvo]

<html>
<body>
<h1> ...
```
- Dokumentin lopuksi palvelin sulkee yhteyden.

- Pyynnössä on oltava **Host:** parametri (tulevaisuudessa (jopa jo nyt) myös koko URL GET:n perässä)
- Vastauksen otsikoissa on **Content-Length:**
- Viesti voidaan paloitella.
- Yhteyttä ei suljeta joka dokumentin jälkeen.
- Kts.esim. <http://www.jmarshall.com/easy/http/>

- ⇒ Kukin UDP paketti on yksilö, lähetetään erikseen, vastaanotetaan (tai jätetään vastaanottamatta) erikseen.
- Kukin paketti sisältää täydelliset osoitetiedot lähettäjistä ja vastaanottajasta.

DatagramSocket, DatagramPacket

- *DatagramSocket* oliota käytetään niin pakettien vastaanottoon kuin lähetykseenkin.

```
pistoke = new DatagramSocket(54321);
```

 - Soketti sidotaan vain paikalliseen porttiin
 - Palvelimella sovittu portti, asiakkaan päässä ei väliä.
- *DatagramPacket* sisältää sekä vastapuolen **osoitteen** (*InetAddress* + portti, tai *SocketAddress*), että **datan** (*byte[]*).
 - Voidaan antaa konstruktorissa, tai myöhemmin.
 - Samaa pakettia voidaan kierrättää (esim. muuttaa osoite).

- Paketin vastaanotto

```
byte[] buf = new byte[1024];   paketit max koko
DatagramPacket paketti =
    new DatagramPacket(buf, buf.length);
pistoke.receive(paketti);      // odottaa pakettia
```

- Paketti on siis luotava (tila varattava) etukäteen, vastaanotto tapahtuu "päälle".
- Vastauksen todellinen koko: *paketti.getLength()*

- Paketin lähetys

```
paketti = new DatagramPacket(buf, buf.length,
                               osoite, portti);
pistoke.send(paketti);
```

- Vastaanotto-vastaus:

- Paketin lähettäjä on yleensä vastauksen vastaanottaja:

```
InetAddress osoite = paketti.getAddress();
int portti = paketti.getPort();
vastauspaketti = new DatagramPacket(sbuf,
                                     sbuf.length, osoite, portti);
```

- Tai osoite ja portti nipussa (*get|set*)*SocketAddress()* (kts. esimerkki).

- Monilähetys: *MulticastSocket*.

- Hyvä C-ohjelmointiopas: **Beej's guide to Network Programming: Using Internet Sockets.**
 - <http://beej.us/guide/bgnet/>

Miten?

- Tarvitaan muutama struct, osoittimia, tyyppinmuunnoksia.
 - Nämä voi aina ottaa esimerkeistä.
- Asiakas: *socket()*, *connect()*, *send()*, *recv()*
- Palvelin: *socket()*, *bind()*, *listen()*, *accept()*, *recv()*, *send()*
 - Usean asiakkaan palvelu: joko *select()* tai *fork()*
- Kts esimerkit.

- Ennen versiota 5.004 (Socket): kuten C:llä
- 5.004- (IO::Socket)

- Kts. esimerkit
- Asiakas

```
use IO::Socket;
```

```
$remote = IO::Socket::INET->new(  
    Proto      => "tcp",  
    PeerAddr   => "localhost",  
    PeerPort   => "daytime(13)",  
    ) or die "cannot connect";  
while ( <$remote> ) { print; ... }
```

- Palvelin:

```
$server = IO::Socket::INET->new(  
    Proto      => 'tcp',  
    LocalPort  => $PORT,  
    Listen     => SOMAXCONN,  
    Reuse      => 1) or die "Sock";  
while ($client = $server->accept()) { ...
```

- Asiakas

```
import socket
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((palvelin, portti))      # osoite on pari
s.send(viesti)
vastaus = s.recv(koko)
```

- Palvelin

```
...
s.bind(osoite)
s.listen(jono)
uusisoketti, osoite = s.accept()
pyynto = uusisoketti.recv(koko)
...
```

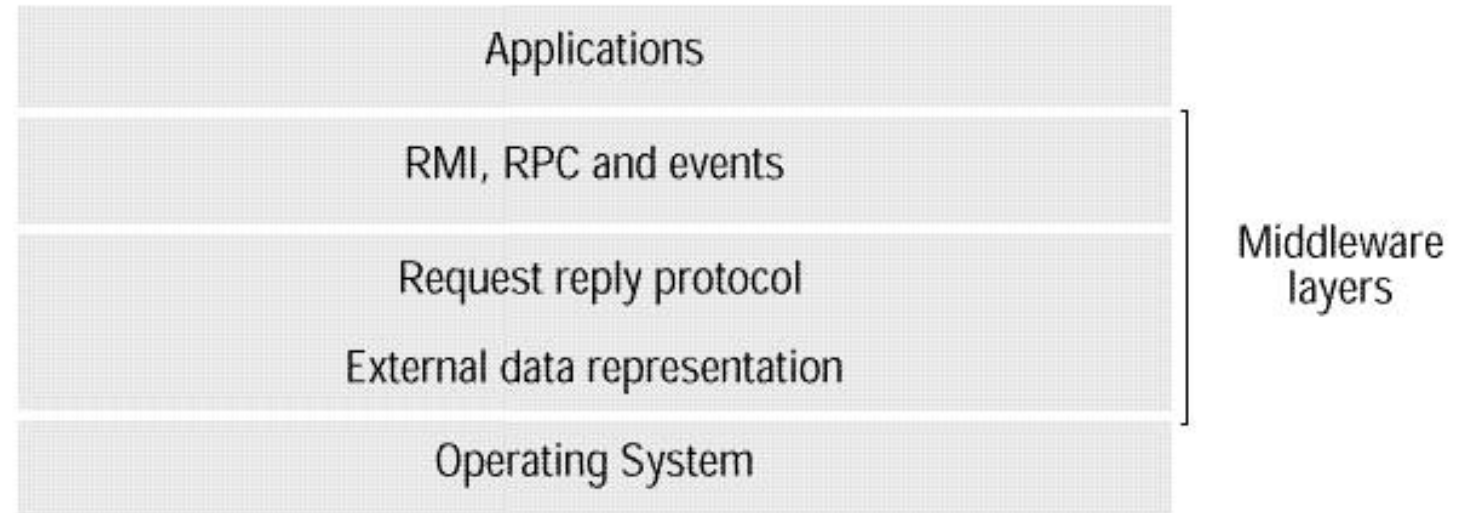
- Lisäksi poikkeusten käsittely (joka ei ole pakollista).
- Kts esimerkki.

Etäproseduurikutsu (RPC) ja etämetodikutsu (RMI)

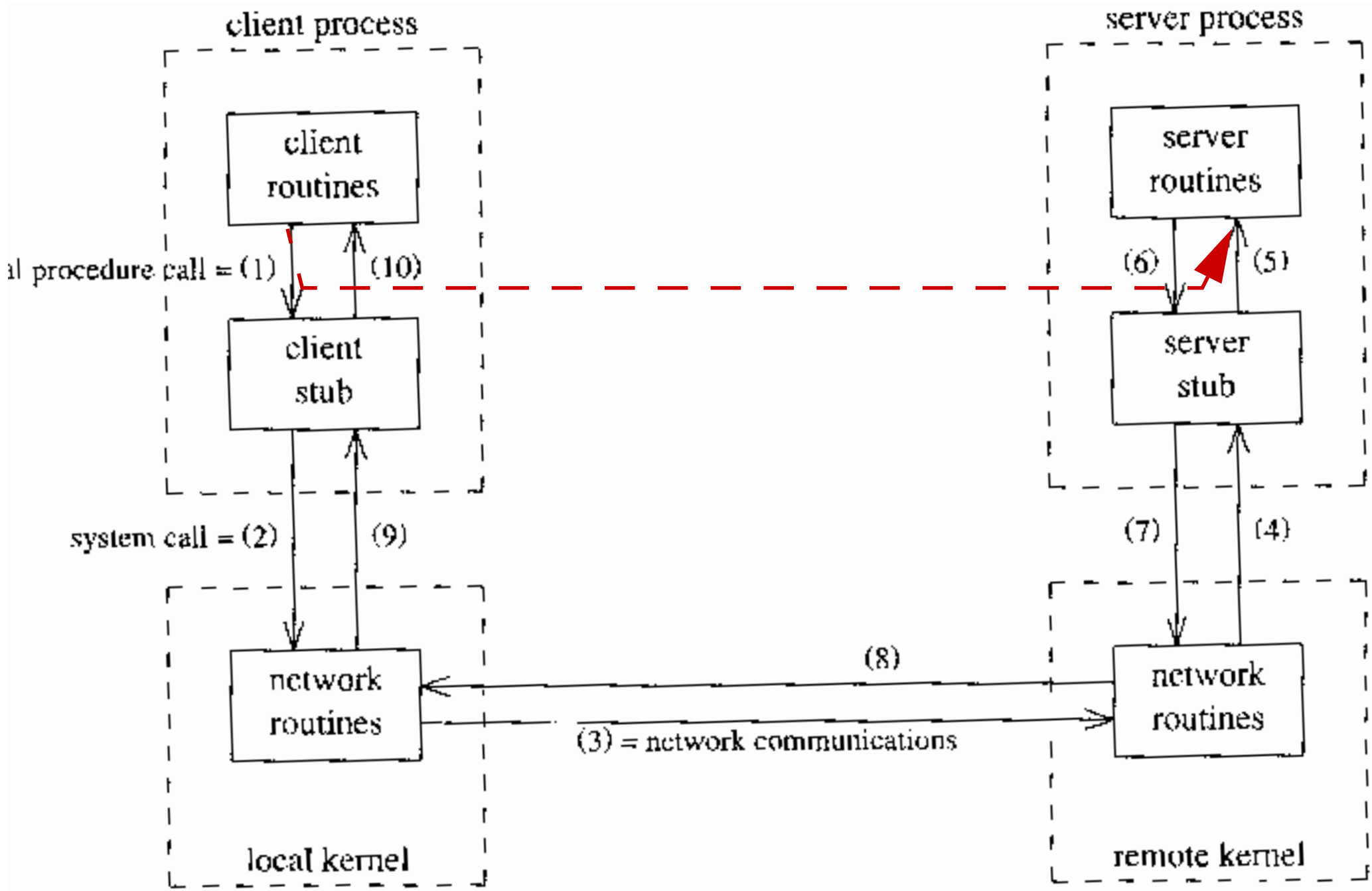
⇒ Tavoite: hajautuksen tuntumattomuus ohjelmoijalle.

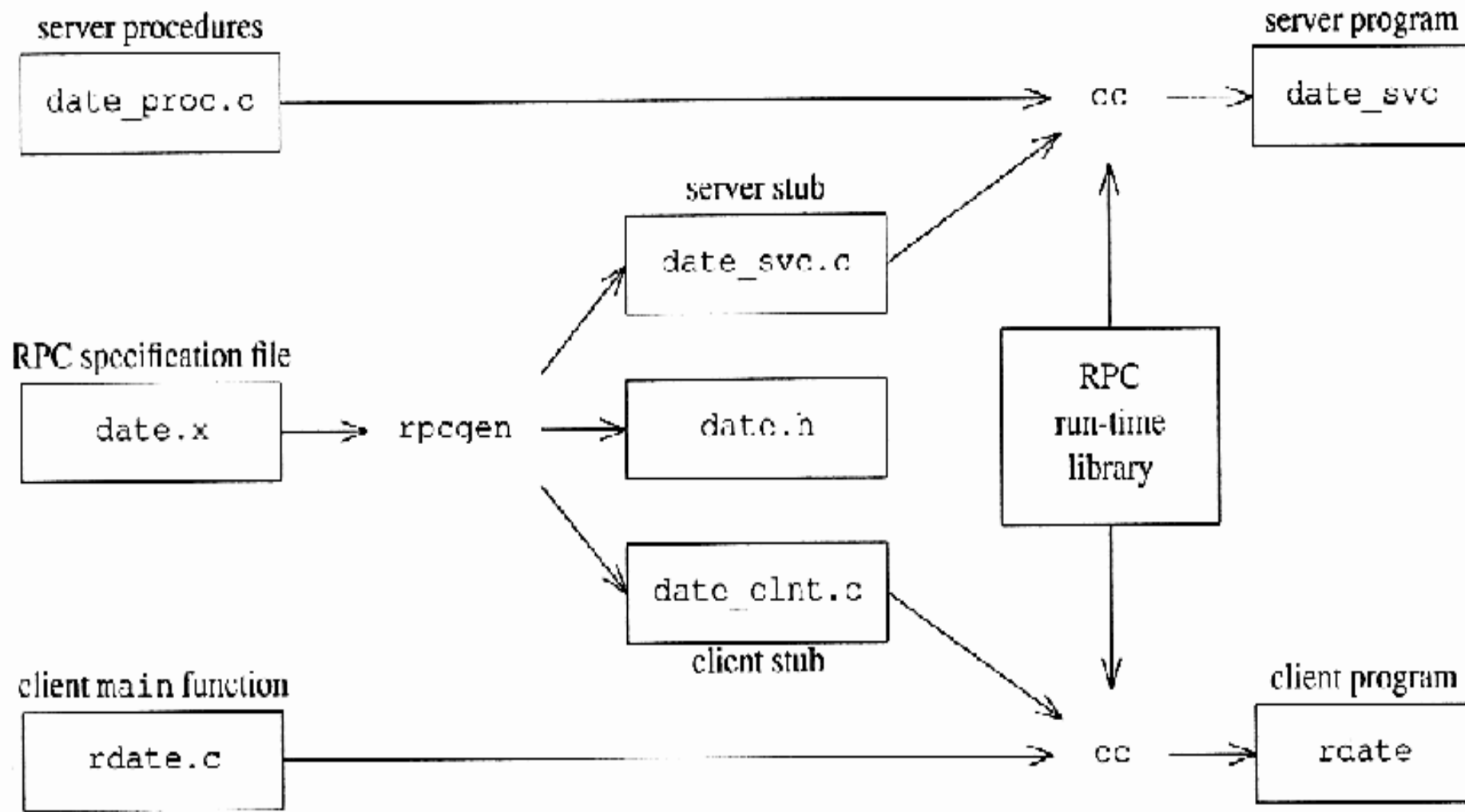
Ratkaisu:

- Palvelua pyydetään suorittamalla **proseduuri/metodikutsu**.
- Jotta kutsu voitaisiin suorittaa, tarvitaan jonkunlainen **viite** etäprosessin/objektiin.
- Kun meillä on viite, kutsu on **kuten paikallinen kutsu**.
- Kutsun varsinaisen tekemisen toiseen koneeseen hoitaa **väliohjelmisto**.



Remote Procedure Call (RPC) (John White 1976) [2]



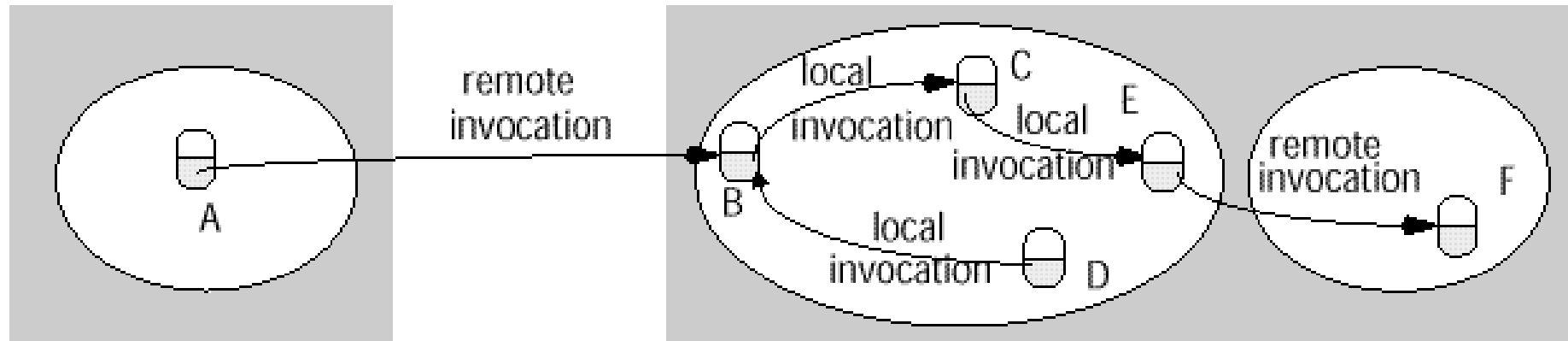


- RPC määrittelytiedosto sisältää proseduurin kuvauksen, **rpcgen** generoi automaattisesti asiakkaan ja palvelimen kannat (stub.c) ja otsikotiedoston.

Palvelinprosessin löytäminen/rekisteröinti

- Kun palvelinprosessi käynnistyy ja sitoo soketin paikalliseen porttiin, se rekisteröi ohjelman (numeron) paikalliselle portmapper demonille.
- Asiakas ottaa yhteyttä ensin portmapper:iin.
- Portmapper palauttaa oikean portin.

⇒ Prosessi/olio lähettää kutsuviestin (mahdollisesti) muualla sijaitsevalle oliolle.



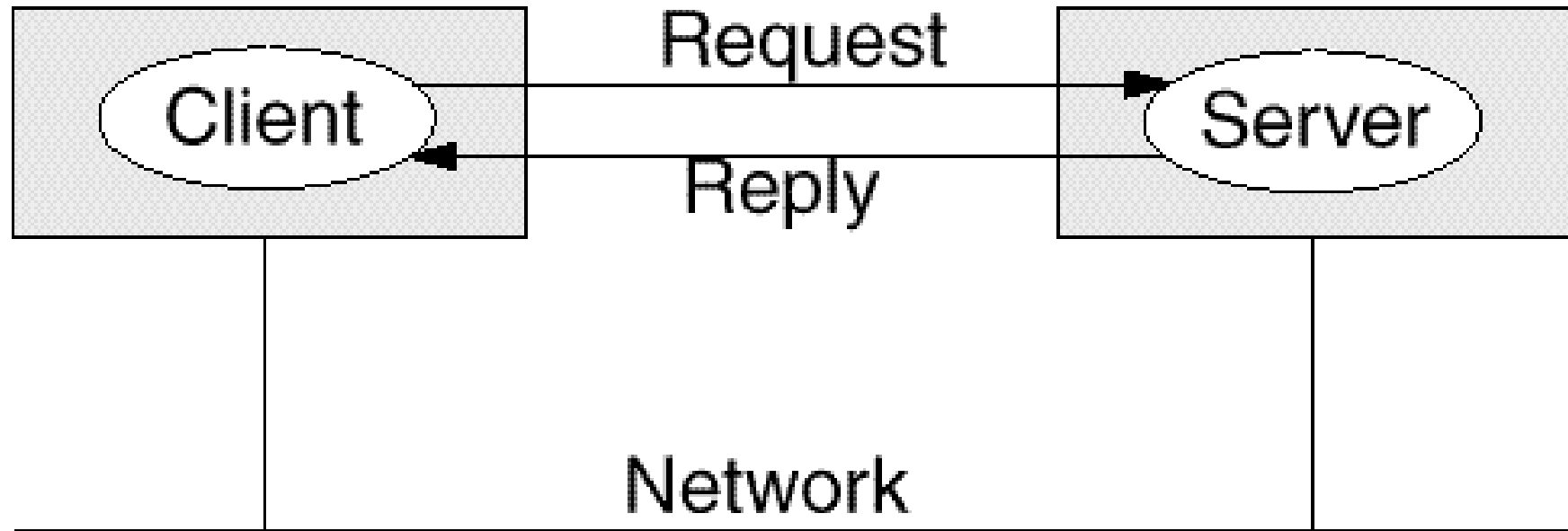
- Jos/kun kutsuttavaan olioon saadaan yhteys, **tuntumattomuus** saavutetaan kauniisti.

Miten hyvin oliomalli (RMI) soveltuu hajautettuun järjestelmään?

- **Metodikutsu on itseasiassa viesti** toiselle oliolle.
- Olioviite
 - Hajautetuissa järjestelmissä tämä tarkoittaisi, että jokaisella oliolla olisi **globaali nimi/osoite**.
 - Periaatteessa mahdollista, mutta hieman kallista toteuttaa.
 - Riittää, että kullakin oliolla **voi olla globaali nimi**.
- **Parametrina** voi olla primitiiviarvo, olio (kopio) tai olioviite.
 - Aina se ei voi olla viite, joskus tiedonkin on siirryttävä.
- Oliot vastaavat "**itsenäisesti**" suorittamalla metodin.
 - Tämä sopii mainiosti hajautettuihin järjestelmään.
- Oliot näyttävät ulospäin **liittymän**.
 - Samoin tämä sopii mainiosti hajautettuihin järjestelmiin.
- Olion tila on **koteloitu**.
 - Hajautetussa järjestelmässä olion tila todella on kapseloitu, ei esimerkiksi pakettisuojattuja muuttujia.

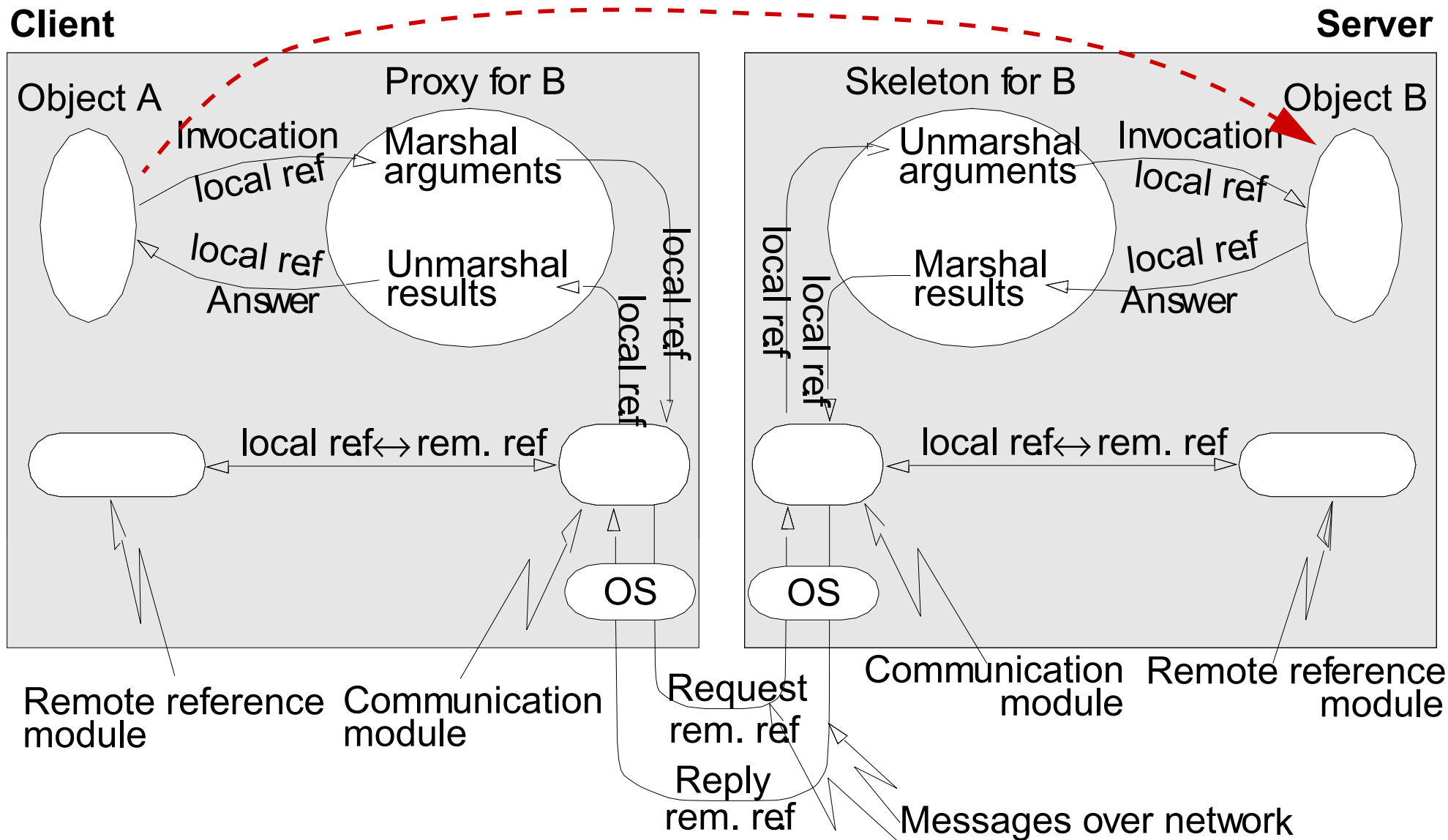
- Olio voidaan jakaa liittymänsä kautta.
- Olion kutsut voivat olla **samanaikaisia**.
- Vikaantumismahdollisuudet otettava huomioon.
- Olioita ei (välttämättä) erityisesti vapauteta.
 - Globaali roskienkeruu on vaikeaa (mutta muisti halpaa).

Etämetodikutsu (RMI) (yleisesti ei vielä Java RMI)



- Asiakas kutsuu:
`palvelin.palvelu(parametrit, palautusarvot);`
- Palvelimella on metodi:
`public palvelu(in tyyppi parametri, out tyyppi tulos);`
- Ohjelmoijan ei tarvitse tietää mitä verkkoviestejä toteuttamiseen tarvitaan.

Mitä oikeasti tapahtuu?



- Asiakas kutsuu paikallista **välitysoliota** (stub, proxy) joka **edustaa etäoliota asiakasprosessissa**.
- Palvelinoliota kutsuu **palvelinrunko** (skeleton) (**edustaa asiakasta palvelinprosessissa**).

Välitysolio (asiakaskanta, proxy)

- Jos asiakkaalla A on viite etäoliioon B, sillä on paikallinen viite välitysoliioon.
- Välitysolio **luodaan kun etäviite muodostetaan**.
- Välitysoliolla on samat metodit kuin itse palvelinoliolla B.
- Kutsuttaessa välitysolio **pakkaa** (marshal) kutsun **parametrit** ja rakentaa **viestin** lähetettäväksi etäprosessille.
- Paluuarvon saadessaan välitysolio purkaa viestin ja palauttaa arvon kutsuneelle oliolle.

Palvelinrunko (skeleton)

- Palvelinrunko tekee vastaavat toimenpiteet (vastaanotto, purku, paikalliskutsu, pakkaus, lähetys) palvelimen päässä.
- Palvelimen päässä on oltava myös mekanismi ohjata vastaanotettu viesti **oikealle oliolle** (ja sitä ennen oikealle prosessille).
- Kommunikaatiomoduulit vastaavat viestien varsinaisesta lähettämisestä.

- **Etäviitemoduuli** (remote reference module) muuntaa paikalliset viitteet etäviitteiksi ja päinvastoin.
- Tyypinmuunnokset
 - Mitä jos/kun (on mahdollista että) asiakas ja palvelin käyttävät erilaisia tiedonesitysmuotoja (luvut, merkkijonot)?
 - Yleensä käytetään **standardoituja esitystapoja** verkkoliikenteessä.
 - Välitysolio ja palvelinrunko tekevät **tyypinmuunnokset** parametrien pakkaus/purkuvaiheessa.
- Kuka tekee välitysolion ja palvelinrungon?
 - Yleensä ne **generoidaan automaattisesti** palvelimen **liittymän** määrittämisestä.

- Vain Java
- Ei järjestelmänlaajuista nimiavaruutta.
- **Ei täysin tuntumaton**, asiakkaan ja palvelimen on huomioitava toisen osapuolen olevan etäällä (yhteydenotto, poikkeukset).
- Kutsun parametrit välitetään **arvoparametreina** (call by value).
 - Parametrien oltava sarjallistuvia (Serializable).
 - **Etäviitteitä** voidaan välittää parametrina.
- Liittymät määritellään normaalisti (**Interface**), mutta ne laajentavat **Remote** -liittymää, operaatiot heittävät **RemoteException** -poikkeuksen.
 - **rmic** generoi asiakkaan **välitysoliion** ja **palvelinrunгон** liittymästä.
- Palvelin avaa palvelun sitomalla sen tietokoneen **rmiregistry** -prosessiin.
- Asiakas paikallistaa etäolioita etätietokoneen rmiregistry:ä (**osoite** (, **portti**)) ja **palvelun nimeä** käyttäen.

Etäliittymä (remote interface)

```
import java.rmi.*;
public interface AikaLiittyma extends Remote {
    public boolean kaykoAika(Integer aika)
        throws RemoteException;
    public boolean kaykoAjat(Set<Integer> ajat)
        throws RemoteException;
}
```

Etäliittymän toteutus

```
import java.rmi.*;
import java.rmi.server.*;
public class AikaOlio extends UnicastRemoteObject
    implements AikaLiittyma {
    public boolean kaykoAika(Integer aika)
        throws RemoteException{
        ...
        return true;
    }
    public boolean kaykoAjat(Set<Integer> ajat)
        throws RemoteException {
        ... } }
}
```

- Molemmat käännetään ensin **javac** -kääntäjällä.
- Liittymä käännetään lisäksi **rmic** -kääntäjällä (< JSE 1.5):
 - `rmic AikaOlio` # ei siis `.java`
 - `rmic` generoi asiakkaan välittimen ja palvelimen asiakaskannan.
 - Java 1.5 ja myöhemmät tekevät tämän automaattisesti.

Asiakas

```
public static void main (String[] args) {
    try {
        AikaLiittyma palvelin = (AikaLiittyma)
            Naming.lookup( "//pal.do.fi:1234/Aikapalvelu" )
        boolean ok = palvelin.kaykoAika(42);
    } catch (RemoteException e) { ... }
}
```

- Voi toki olla muuallakin kuin `main()`:ssa.

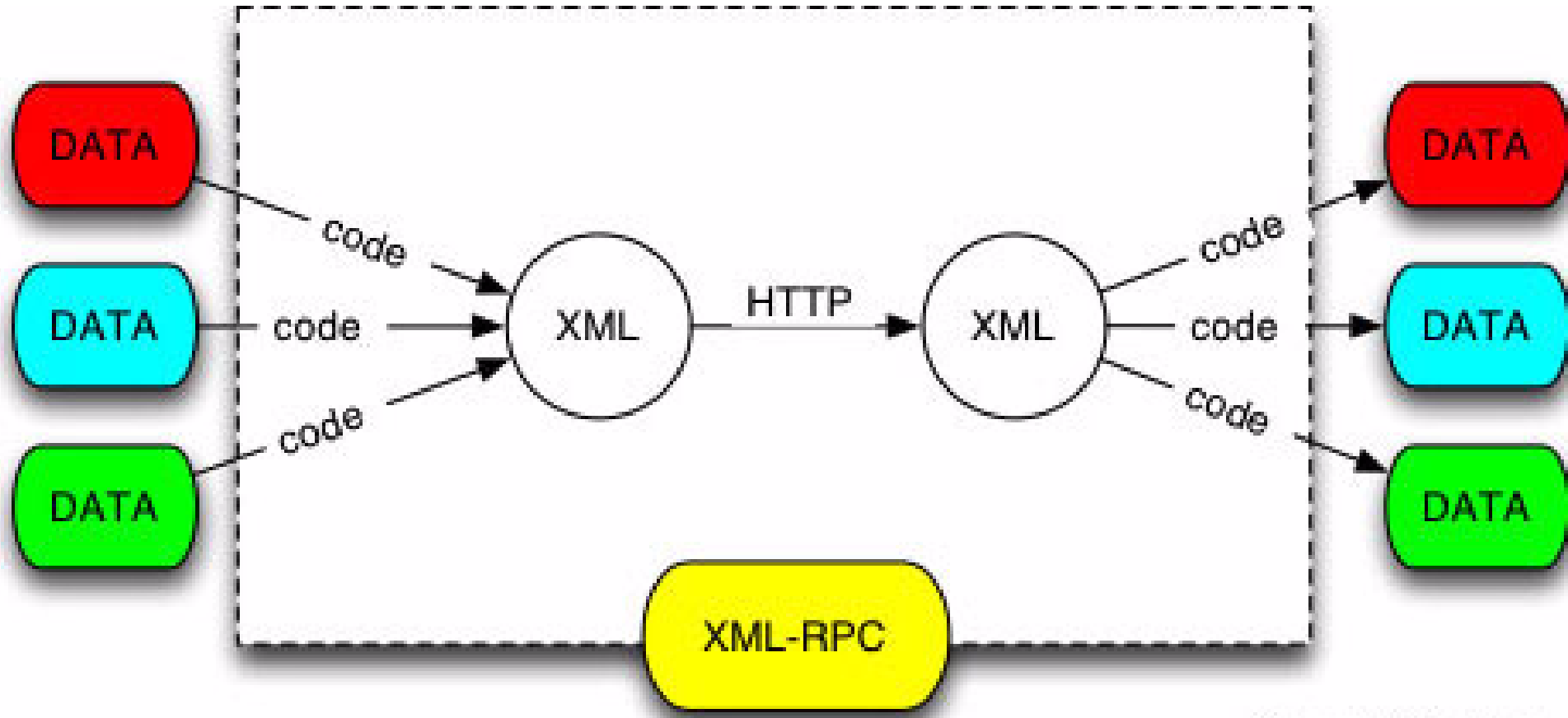
```
public static void main (String[] args) {  
    try {  
        Naming.rebind("Aikapalvelu", new AikaOlio());  
    } catch (Exception e) { }  
}
```

- Luo palvelevan olion.
- Sitoo sen jollakin **nimellä** rmiregistry:yn.
- Heittää poikkeuksen jos rmiregistry:ä ei löydy tai nimi on jo varattu.

Käynnistäminen

- rmiregistry 1234 &
 - Portti voi olla varattu, muista **pysäyttää rmiregistry** lopuksi.
- java Palvelin &
- java Asiakas

- Kts Java Tutorial, RMI
- Kts. esimerkit.
- Uudemmat versiot tukevat myös luokkien kuvausten ja toiminnallisuuden välittämistä (**Dynamic Code Loading**).
 - Huomioi tietoturvariski!



Source: JY Stervinou

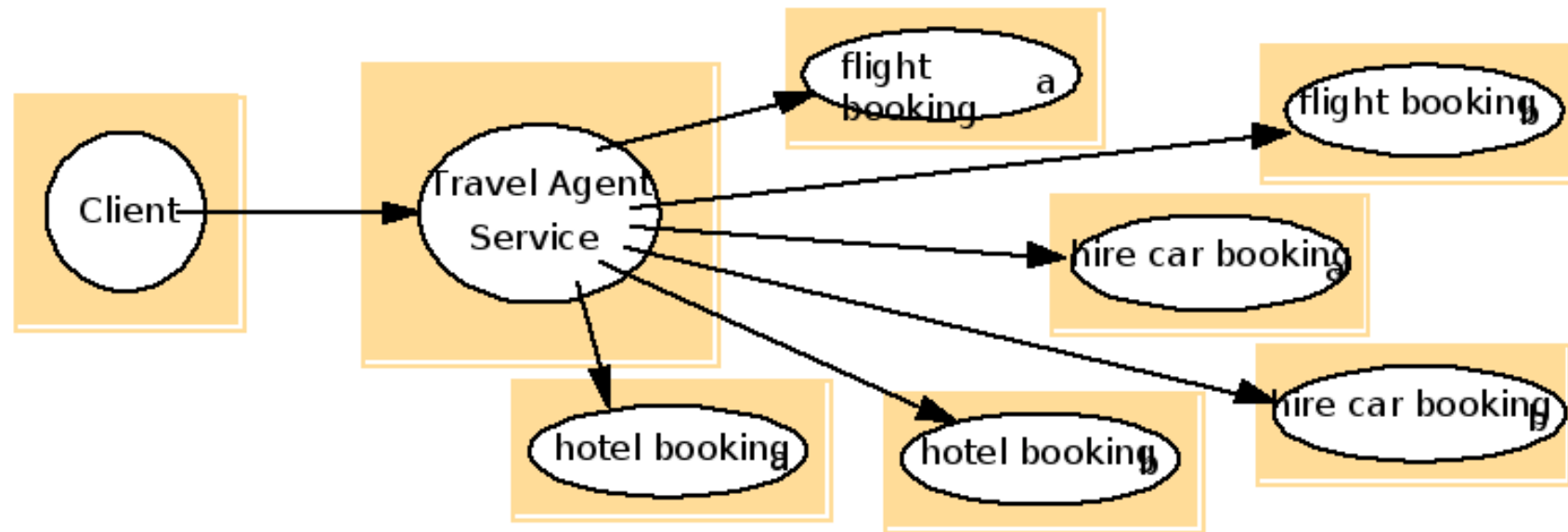
- Asiakas: Koodaa proseduurikutsun (resurssi, parametrit) XML:ksi ja lähettää viestin HTTP:llä, vastaanottaa tuloksen XML:nä.
- Palvelin: joko itsenäinen sovelluspalvelin joka toimii HTTP "palvelimena", tai CGI (tms) skripti oikealla WWW-palvelimella.
- Kirjastot hoitavat XML:n ja HTTP:n, ohjelmoijalle varsin yksinkertainen, kts. esimerkit. Kieliriippumaton, kirjastoja useimmille kielille.

SOAP (Simple Object Access Protocol)

- Kts alla.

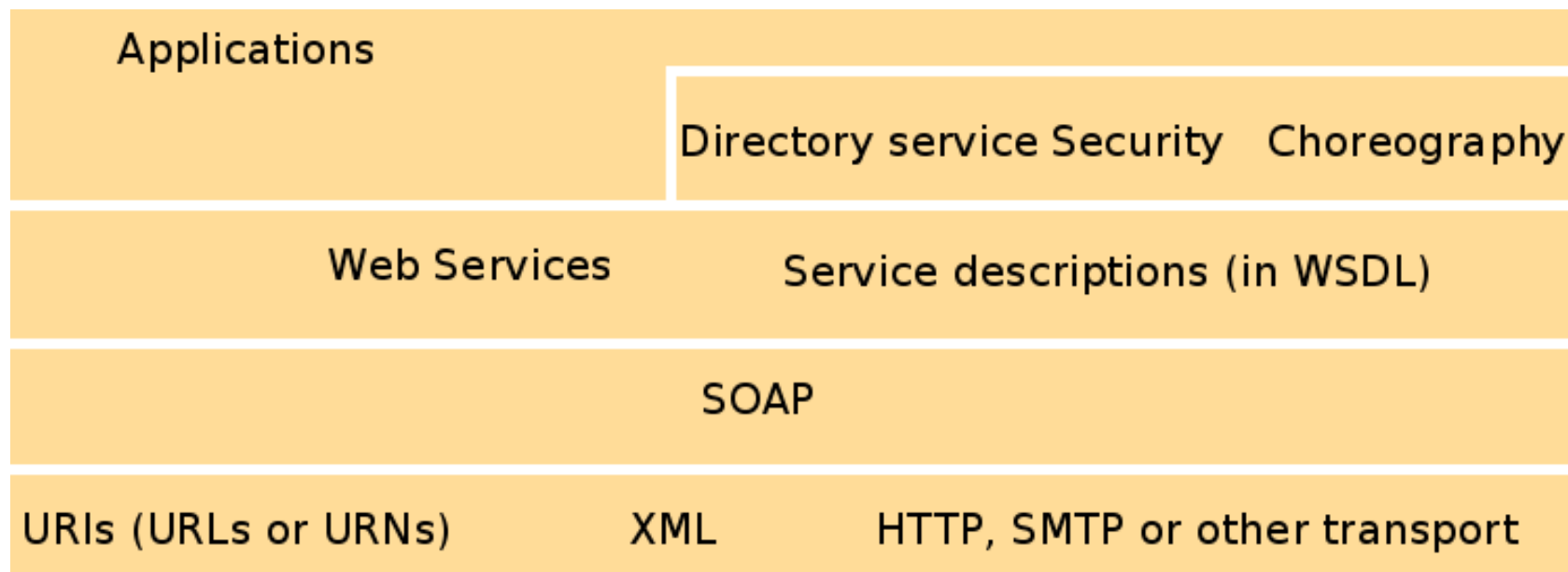
- ⇒ Palveluita joita asiakasohjelmistot käyttävät web (-sukuisia) protokollia käyttäen.
- Web-palvelu != web-palvelin.
 - Protokollana yleensä HTTP (jonka päällä SOAP).
 - Palvelu yksilöidään **URI**:lla (Unified Resource Identifier).
 - URL (... Location) tai **URN** (... Name)
 - Asiakasohjelmistolla ei tässä tapauksessa (yleensä/välttämättä) ole ihmistä käyttäjänä, vaan vain **järjestelmät keskustelevat**.
 - Verrattuna esim. CORBA:an, web-palveluiden käyttöönottokynnys on yleensä matalampi.
 - **Tehokkuus** ei yleensä ole kovin hyvä (mutta se ei ole tavoitteenakaan).

- Usein web-palvelut käyttävät ja yhdistelevät muita web-palveluita [1]. 259



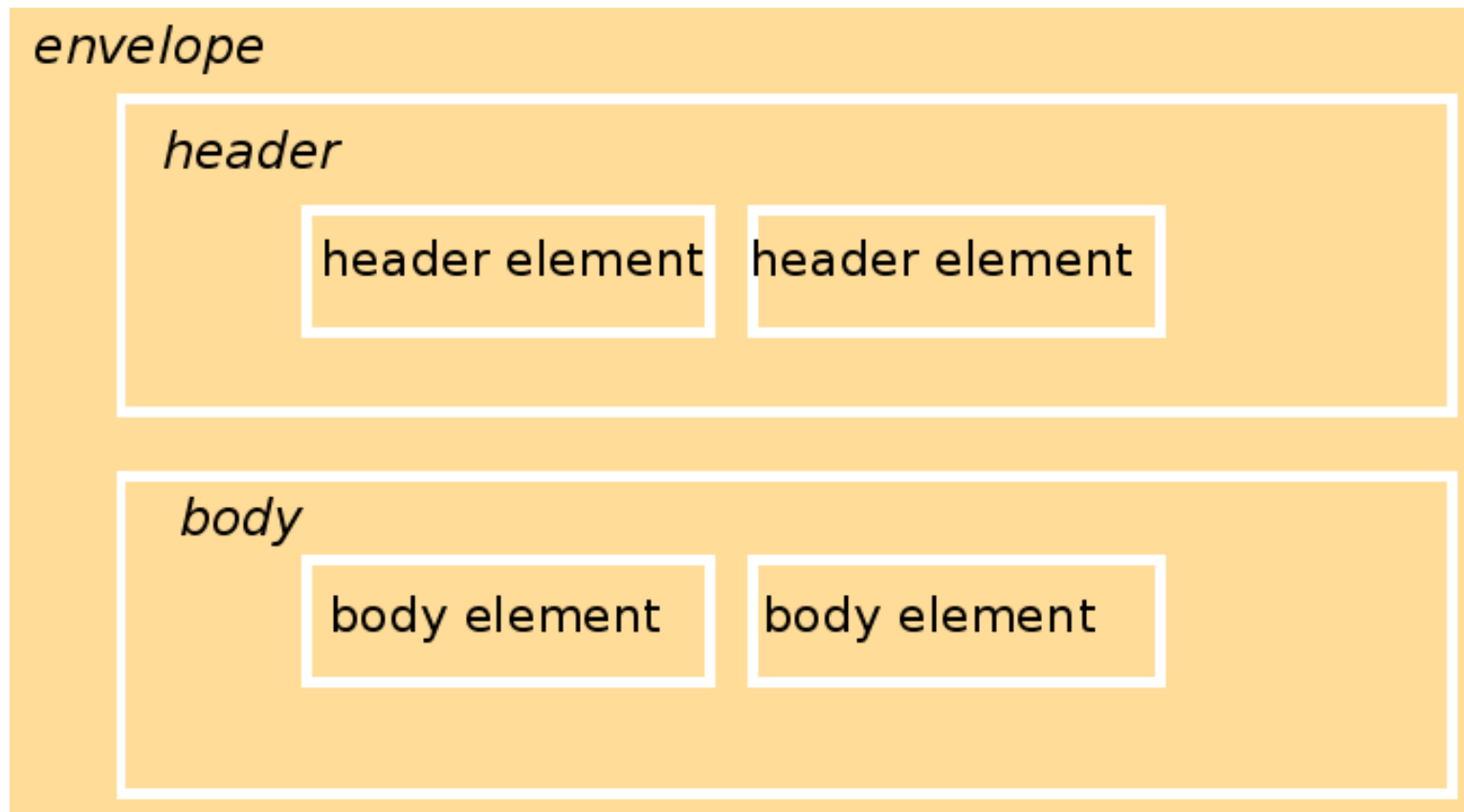
- Tietokoneille tarkoitettu web-palvelu on usein ihmisille tarkoitettun liittymän rinnalla (tai takana).
- Lähtöjään web-palvelut on ad-hoc tekniikka, mutta nykyään SOAP on varsin rakenteinen ja palveluiden rajapinnat voidaan kuvata ja rakentaa WSDL-kielillä (Web Services Description Language).

- Monitasoinen: [1]



SOAP (Simple Object Access Protocol)

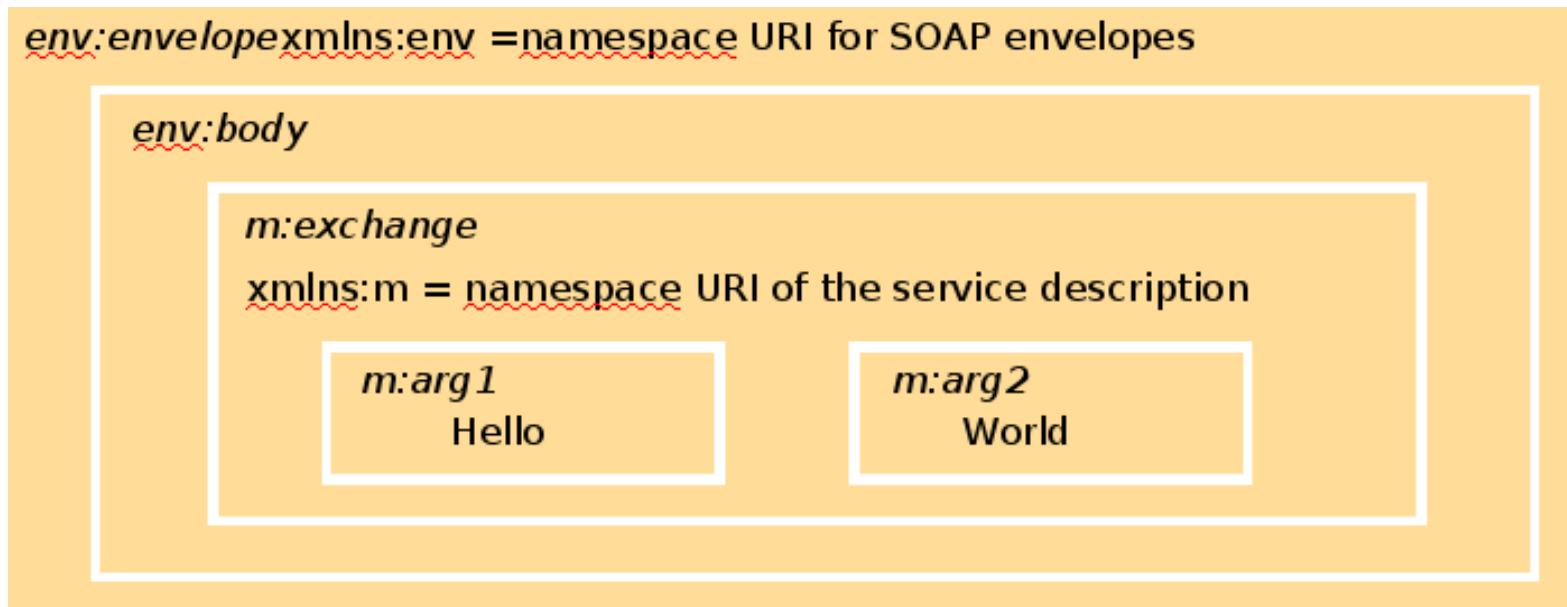
- <http://www.w3.org/2000/xp/Group/>
- Etämetodikutsu XML:llä (kuten XML-RPC).
- Tukee monipuolisempia palveluja (nimi-, välitys, jne).
- Pyyntö ja vastaus koodataan XML:llä. Viesti kuljetetaan yleensä HTTP:llä (myös mm. SMTP ja pelkkä TCP/UDP)
- Viestit kulkevat "kirjekuorissa" (**envelope**) [1].



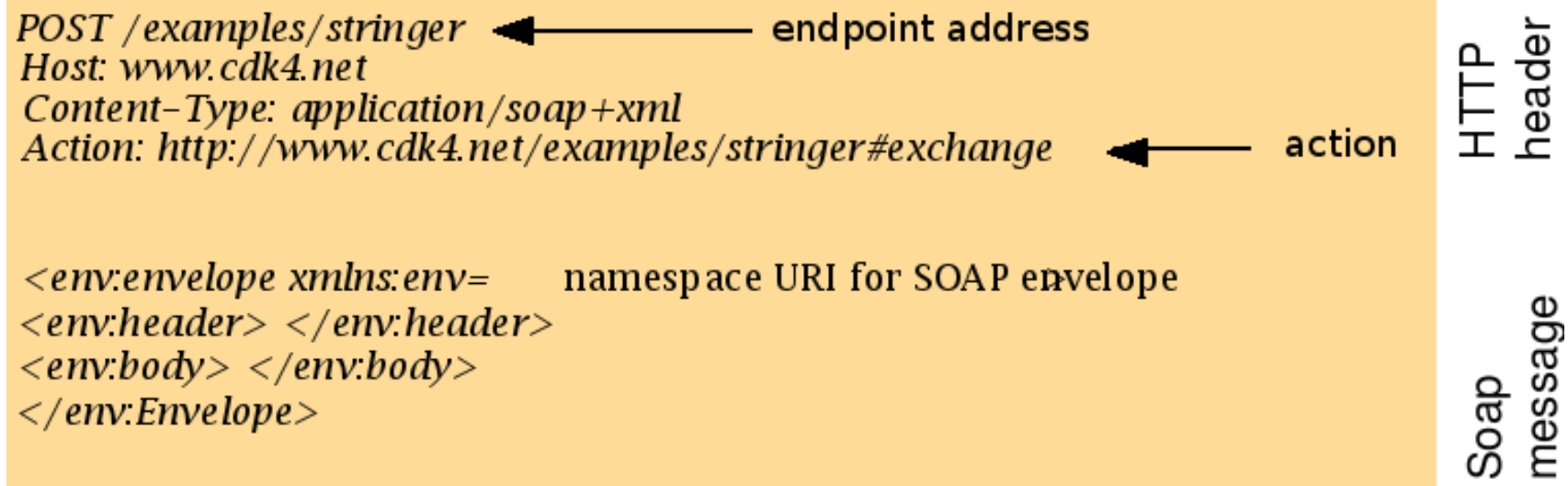
- Kunkin viestinosan otsikkotieto (header) kuvaa pyynnön ja määrittelee kenelle se on tarkoitettu.
 - Välisolmut voivat tarkastella (ja muuttaa) otsikkotietoja.
- Palvelin voi olla joko erityispalvelin jossa on integroitu HTTP-palvelin, tai HTTP-palvelimen käynnistämä ohjelma.

⇒ SOAP -viesti on (suhteellisen) monimutkainen.

- Peruspyyntö ilman headereita [1]:

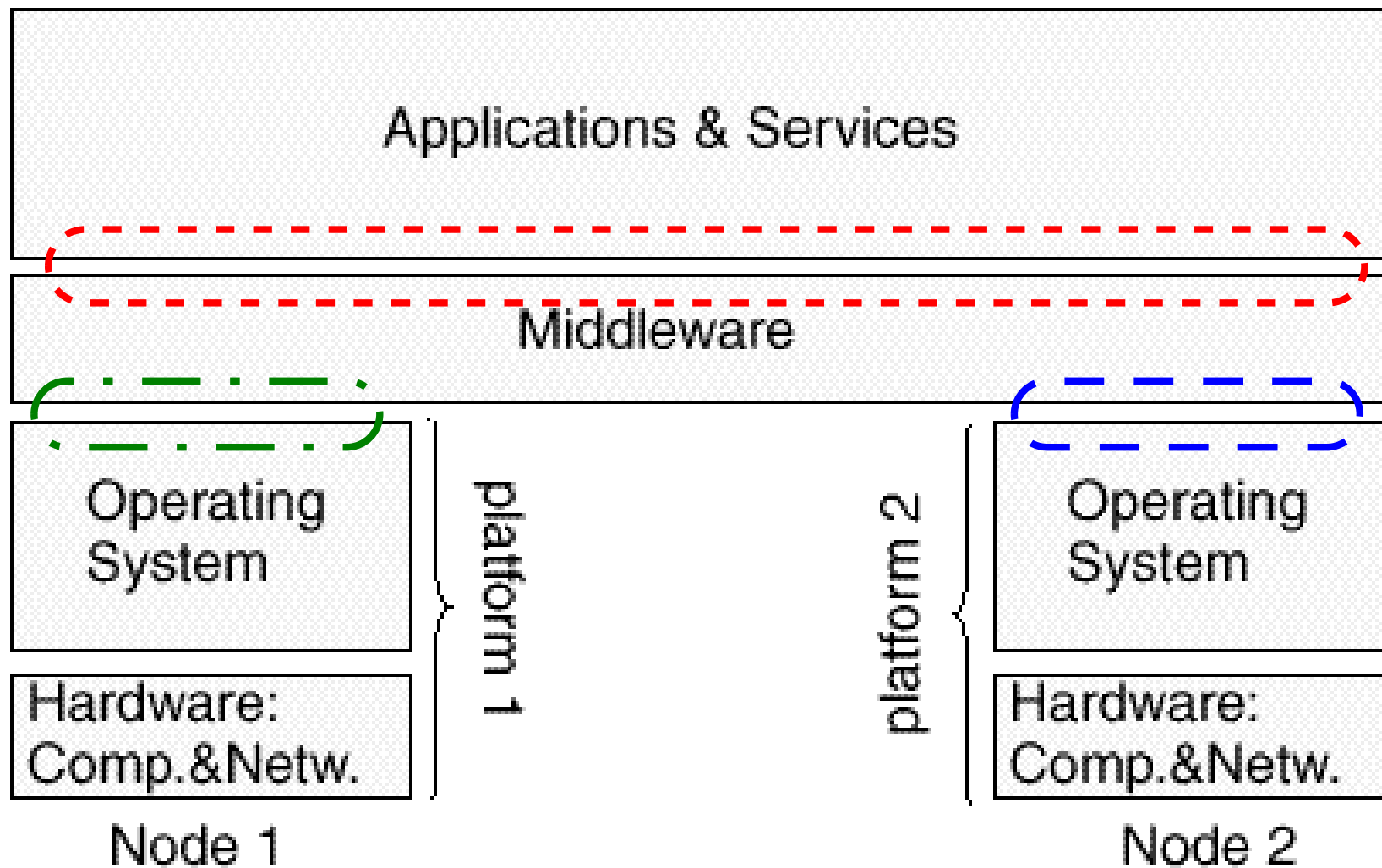


- XML-koodattu pyyntö välitetään HTTP:llä [1]:



- Onneksi eri **ohjelmointirajapinnat** (API:t) eri kielissä (Java, Perl, Python, jne) piilottavat tämän monimutkaisuuden.
- Jos tekee vain asiakkaita, ei myöskään WSDL:ää yleensä tarvitse osata (kunnolla).

Väliohjelmisto (sivu 49)

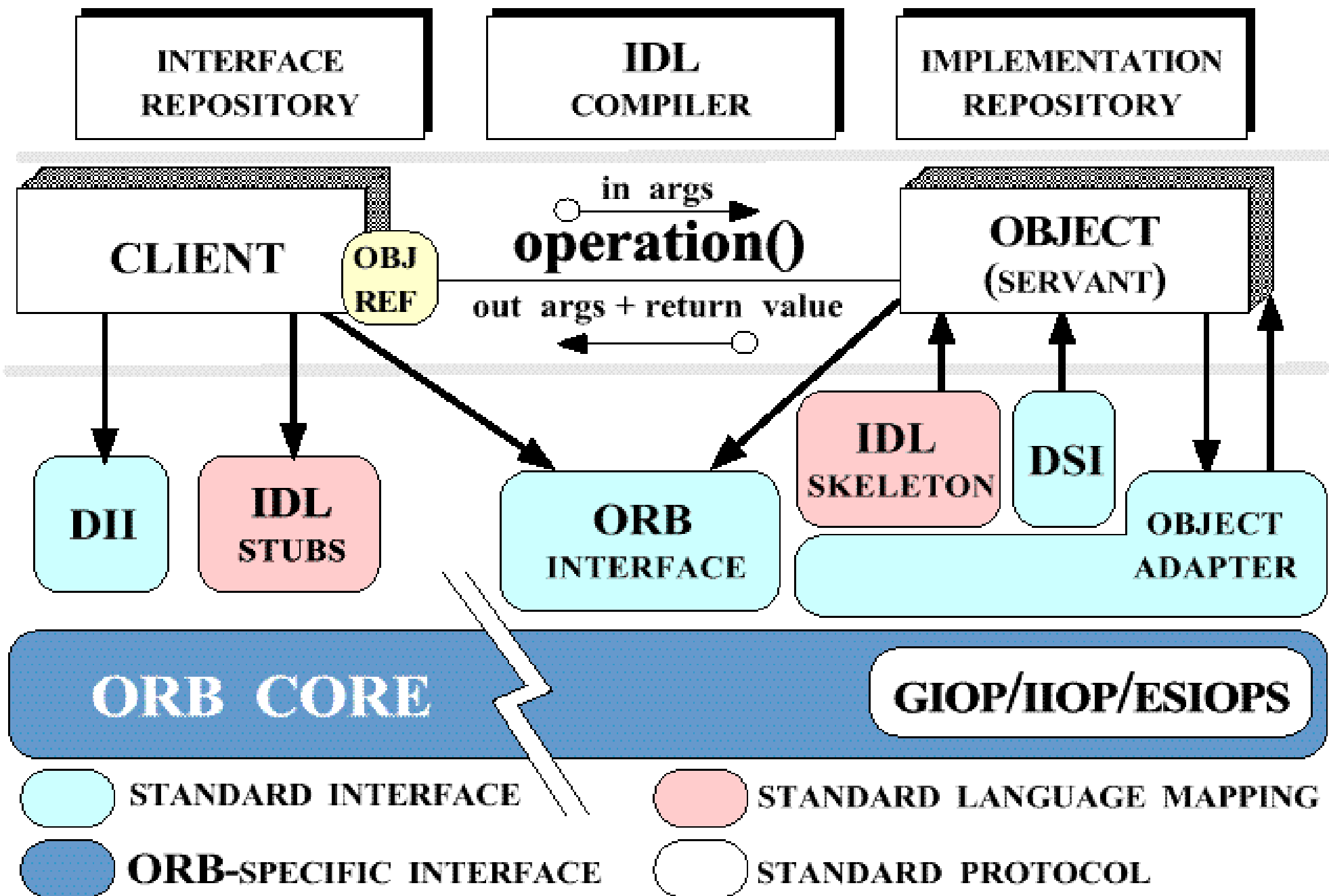


CORBA (Common Object Request Broker Architecture)

www.omg.org, www.corba.org

⇒ Object Management Group (OMG): teollisuuskonsortio

- CORBA määrittää liittymien kuvaukset ja **oliokutsuvälittimen** (ORB) toiminnan.
- Eri valmistajien ORB:t toimivat yhteen.
- **Liittymänkuvauskielellä** kuvatut palvelut voidaan toteuttaa eri kielillä (ja vastaavasti kutsua).
- Palveluja ja liittymiä voidaan ottaa mukaan staattisesti (käännösaikana) tai dynaamisesti ajonaikana (joskin se on aika monimutkaista).



⇒ Kuten C++ esittelyjä

- interface eikä class, esim,

```
typedef float Hinta;
```

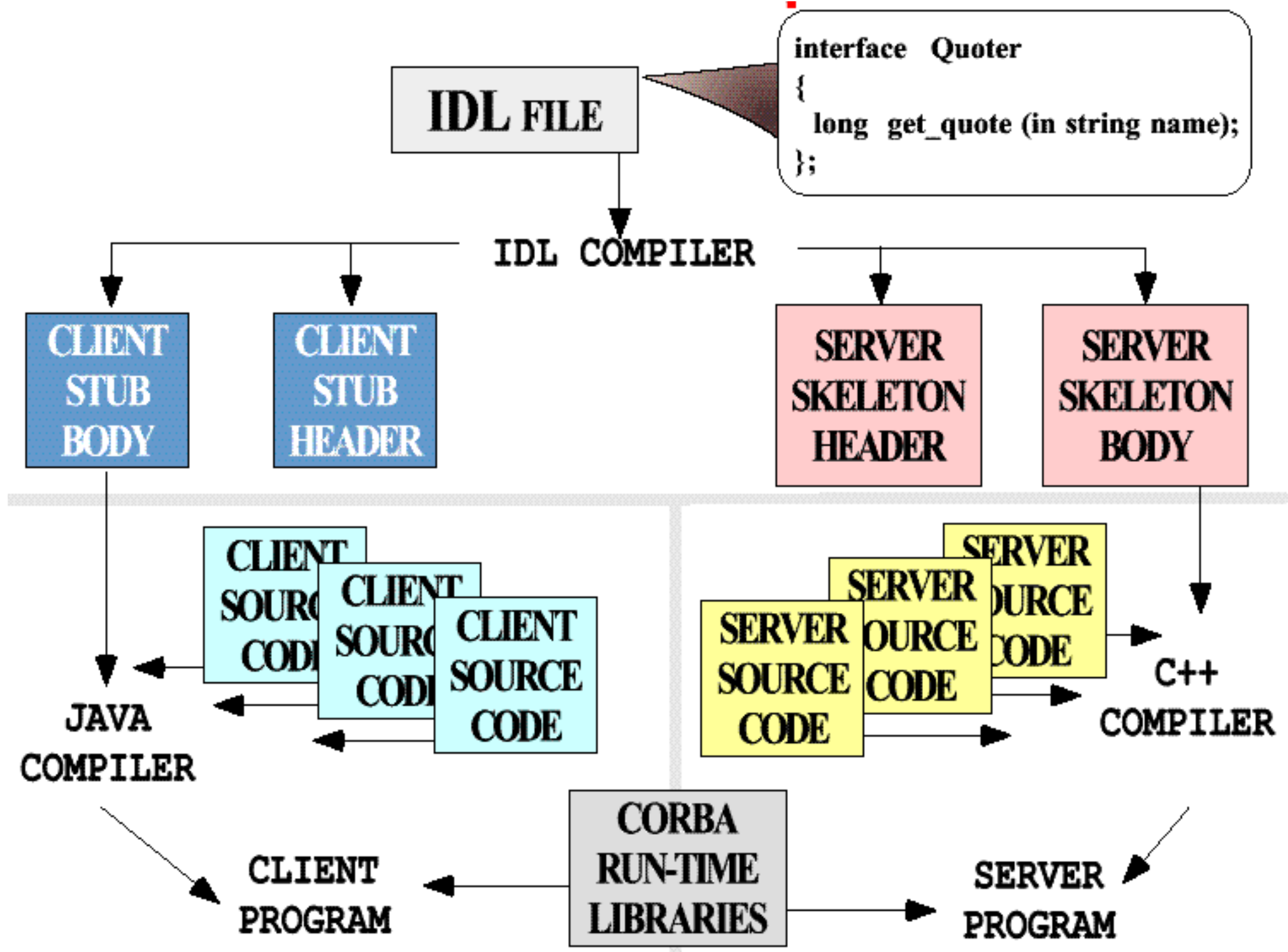
```
struct Esine {  
    string nimi;  
    Hinta minimi;  
};
```

```
typedef sequence<Item> Kaikki;
```

```
interface Huutokauppa
```

```
{  
    exception TuntematonEsine {};  
    boolean Tarjoa(in Esine i, inout Hinta p)  
        raises TuntematonEsine;  
    Kaikki AnnaKaikki();  
    // ...  
}
```

- Ei tietojäseniä, toteutusta. (Moni)perimys, moduulit, attribuutit.
- Sidonnat ja kääntäjät useille kielille.



```

interface Quoter
{
  long get_quote (in string name);
};
  
```

Luku 5

Rinnakkaislaskenta

Mitä on rinnakkaislaskenta? (s. 270)

Miksi rinnakkaislaskentaa tarvitaan? (s. 273)

Nykyiset (kaupalliset) rinnakkaiskoneet (s. 285)

Miten rinnakkaiskoneita ohjelmoidaan?

Esimerkki rinnakkaisalgoritmista.

Mitä on rinnakkaislaskenta?

⇒ "Käytetään montaa tietokonetta yhden asian ratkaisuun!"

- Kaksi on parempi kuin yksi, tuhat on parempi kuin kaksi (vrt. ihmistyö)
- Tehtävä pitää osata **jakaa moneen** osaan!
 - Jotkin tehtävät on helppo jakaa, toisia ei.
- Jotta monta tietokonetta/ihmistä voisi tehdä yhteistyötä, on niiden pysyttävä **kommunikoimaan**.
 - Syötteensä yksi tehtävä, tuloksena yksi ratkaisu.
 - Kommunikaatiovälineitä on monenlaisia (ja monentasoisia)
- Tietokoneen ei tarvitse olla kokonainen ...
 - Emme tarvitse tuhatta monitoria, näppäimistöä, koteloa, verkkokorttia, jne

Esimerkki: lajittelu

- Syöte: nippu A4 arkkeja, kussakin mm. henkilön nimi
- Tehtävä: lajitella arkit aakkosjärjestykseen.
- 10, 100, 1000, tai 10000 A4 arkkia. (1 mm, 1 cm, 10 cm, 1 m)

Yksin lajittelu: [TRA1]

- 10 arkkia:
 - kunhan sutaisee, 30 s [3 s/arkki]
- 100 arkkia:
 - jako 10 pinoon, pinojen lajittelu, yhdistäminen, 8 min [5 s/arkki]
- 1000 arkkia:
 - jako 10 pinoon, pinojen lajittelu kuten edellä, 2 h [7 s/arkki]
- 10000 arkkia:
 - jako 10 pinoon, pinojen lajittelu kuten edellä,
 - 25 h [9 s/arkki], **kaipaa apua ...**

Rinnakkaislajittelu:

- 10, 100, 1000, 10.000 **apulaista!**
- Työn organisointi tärkeää!

Suuri kysymys: **onko apulaisista hyötyä, ja miten paljon?**

- Nopeuttaako 10 apulaista työtä 10 kertaisesti???
- 10 arkin tehtävässä **ei**
- 10.000 arkin tehtävässä **kyllä** (ainakin melkein)
- Nopeuttaako **10.000 apulaista** työtä 10.000 kertaisesti???
- 10 arkin tehtävässä **ei**
- 10.000 arkin tehtävässä **ei**, mutta **aika paljon**.
- 100.000.000 arkin tehtävässä: **kyllä**
- Mikä on **optimi/maksimi määrä** apulaisia kullekin tehtäväkoolle???
- Minkä mukaan optimoidaan:
 - Absoluuttisen seinäkelloajan?
 - Kustannustehokkuuden? (henkilötyösekuntia/arkki)

Miksi rinnakkaislaskentaa tarvitaan?

⇒ Miksi tietokoneet ovat hyödyllisiä?

- Koska tietokoneet ovat nopeita LASKEMAAN ja niissä on suuri ja nopea MUISTI.

Eikö P4 3.8 GHz riitä?

- Huomaa, 3,8 GHz markkinoilla jo 2004!
- Laskentateho tuplaantuu alle kahdessa vuodessa! [Moore]
- AMD/Intel 2-3 GHz on sitäpaitsi edullinen.
- 20 vuotta sitten jotkut olisivat maksaneet miljoonia nykyisestä kotitietokoneesta.

Mitä vielä tarvitsemme?

- (Ihmiset ovat ahneita ja hätäisiä)

⇒ Jotkin tehtävät ovat liian **vaativia** ja **kiireellisiä** yhden prosessorin laskettavaksi.

⇒ Jotkin laskentatulokset ovat sitä **tarkempia** (arvokkaampia) mitä enemmän laskentaa niiden tekemiseen voidaan käyttää.

- Tekstinkäsittely?
- WWW-selailu?
- Pankkijärjestelmä?
- eBusiness?
- Pelaaminen?
- **Oikean maailman simulointi!**
 - Maailma koostuu tosi pienistä palasista!
 - Kaikkia alkeishiukkasia ei voida nykytietokonein simuloida muuta kuin mikroskooppisen pienenä kappaleesta!
 - Mitä pienempiä palasia voimme simuloida, sitä tarkempia tuloksia saamme!
 - Pienemmät palat -> lisää paloja -> lisää laskettavaa!
 - Rajattomasti laskettavaa!

Miksi haluamme simuloida reaali maailmaa?

276

- Laitteen **testaaminen** rakentamatta sitä.
- **Optimointi**.
- Keinotekkoisten asioiden "näkeminen".
- **Luonnonilmiöiden ennustaminen**.

Esimerkki: sään ennustaminen:

- Historiatietoa, jatkuvat mittaukset, satelliittitieto.
- **Simuloidaan** ilmakehän tulevaa tilaa **fysiikan** lakien ja säämallien mukaan:
 - ilman lämpötila, paine, kosteus, liike, maan muodot, aurinko, ...
- Molekyylejä on paljon!
- Laskentaa enemmän kuin mitä voidaan mitenkään laskea.
- Tyydytään **pienempään resoluutioon**: mallinnetaan jonkin kokoista ilmalohkoa (3D) yhtenä kokonaisuutena.
 - Tarkkuutta menetetään, mutta laskenta tulee mahdolliseksi.
- Ennustus mahdollisimman pitkälle tulevaisuuteen.
 - Valitettavasti virheet kertautuvat.

⇒ **Sääennusteet ovat arvokkaita!**

- **Hitaasti** valmistunut "menneisyyden ennuste" on täysin **arvoton!**

- Haluamme mahdollisimman **tehokkaan tietokoneen!**
 - Olemme jopa valmiit **maksamaan** siitä!
 - Valitettavasti IA256 @ 100 GHz vasta 2020+
 - Ei edes suurella suurella rahalla

⇒ Käytämme siis **useaa prosessoria lisätehon saamiseksi**

- Ilmatieteen laitos 2008 tilanne esimerkiksi
 - 304 Intel Itanium2 prosessoria ja 304 gigatavun keskusmuisti + ~128 × 1.1GHz Power4 (CSC)
 - 44×44×0,5 km (×3 min) Kanadasta Uralille, 3-10 päivälle
 - 15×15×0,5 km (×2 min) Islannista Moskovaan, 3-10 päivälle
- Rinnalla tarkempi:
 - 7,5×7,5×0,? km (×2 min) Islannista Moskovaan, 3-10 päivälle
 - 304 Intel Itanium2 prosessoria ja 304 gigatavun keskusmuisti.
- Koekäytössä 2,5 km

- Tietokantahaut
- Digitaalinen signaalinkäsittely, erityisesti etsintä, tunnistus
- Monimutkaiset käyttöliittymät (virtuaalitodellisuus, pelit)
- DNA mallinnus ja haut
- Molekyylimallinnus (esim. entsyymit, lääkeaineet)
- Säteilyn mallinnus
- Fuusioplasman mallinnus
- Ympäristön mallinnus (saasteet, maanjäristykset, merivirtaukset, ilmaston lämpeneminen pitkällä aikavälillä)
- Optimointi (aero-/hydrodynamiikka)
- Kryptoanalyysi.
- Hahmontunnistus
- Tiedon louhinta/indeksointi/luokittelu
- Keino"äly"

Esimerkki: talon rakentaminen

- **Yksi** kirvesmies voi rakentaa talon **vuodessa**.
- **Kaksi** kirvesmiestä noin puolessa **vuodessa**.
- **12** kirvesmiestä yhdessä **kuukaudessa vaikeaa**, vaatii ainakin hyvää suunnittelua ja erikoistekniikoita.
- **365** kirvesmiestä yhdessä päivässä: **todennäköisesti mahdotonta**.
- **Miljoona** kirvesmiestä **10 sekunnissa**: varmasti mahdotonta.

Miten rakentaa talo viikossa?

- Osaavat tekijät.
- Työn synkronointi.
- Osittain **erilliset komponentit**.
- Useampi kuin yksi työnjohtaja.
- Hyvät suunnitelmat ja ohjeet, useampi kopio.
- Ei hitaita komponentteja.
- Ei osien välisiä riippuvuuksia.

Siis:

- Rinnakkaistusmahdollisuudet riippuvat ongelmasta (kaivo/oja).
- Kommunikointi ja koordinointi ovat elinehtoja.

⇒ Nopeutus, (lisä)työ, tehokkuus.

Työvoima	Kalenteriaika	Nopeutus	Työ	Työkulut	Tehokkuus
1	1 vuosi	1,00	1,00 htv	36.000 e	1,00
2	7 kk	1,71	1,17 htv	42.000 e	0,86
4	4,5 kk	2,67	1,50 htv	54.000 e	0,66
365	5 päivää	73,00	5,00 htv	180.000 e	0,20

- Optimaalinen peräkkäisalg yhdellä prosessorilla, suoritus aika $T_s(N)$
- Rinnakkaisalgoritmi P :llä prosessorilla, suoritus aika $T_p(N, P)$
- **Nopeutus** $= T_s / T_p$
 - Nopeutus $T_s / T_p = O(P)$
 - Ylilineaarinen nopeutus ei ole mahdollinen koska muuten olisimme samalla keksineet uuden nopeamman peräkkäisalgorimin.
- **Työ** (kulutettu resurssi) $= T_p \times P$.
- Jos työ $T_p \times P = O(T_s)$, kyse on **työoptimaalisesta** (work optimal) algoritmista

⇒ Voimmeiko nopeuttaa suoritusta rajatta aina vain prosessoreja lisäämällä?

- Emme rajatta, ongelmilla on jokin raja, jota nopeammin niitä ei voida ratkaista, useimmiten sen on logaritminen.
- Käytännössä rinnakkaistuksen rajana on **raha**.
- **Työläät ongelmat ovat suuria** (datat ovat suuria) (pl. NP vaikeat).
- Pienet ongelmat ratkeavat nopeasti vähemmälläkin prosessoreilla.
 - Korttitalo – leikkimökki – omakotitalo – rivitalo – kerrostalo – pilvenpiirtäjä.
- Teoriassa rinnakkaistuksen rajana ovat avaruuden 3-ulotteisuus ja valon nopeus.

Tavoitteet

- mahdollisimman **suuri nopeus**
 - ei väliä kuinka monta prosessoria tarvitaan
 - useisiin ongelmiin löytyy logaritmisessa ajassa toimiva algoritmi
- mahdollisimman hyvä **hyötysuhde**
 - valitettavasti peräkkäisalgoritmilla on aina paras hyötysuhde
- jotain näiden välillä, tai
 - **annetussa ajassa** mahdollisimman vähillä (ja halvoilla) prosessoreilla (ja muulla laitteistolla), tai
 - **annetulla prosessorimäärällä** (laitteistolla) mahdollisimman nopeasti

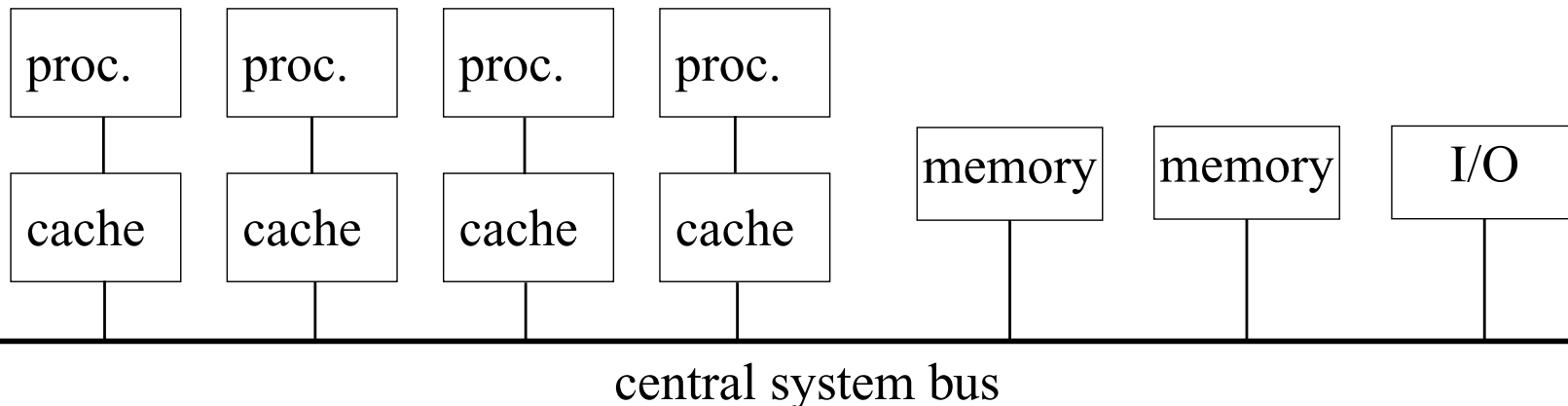
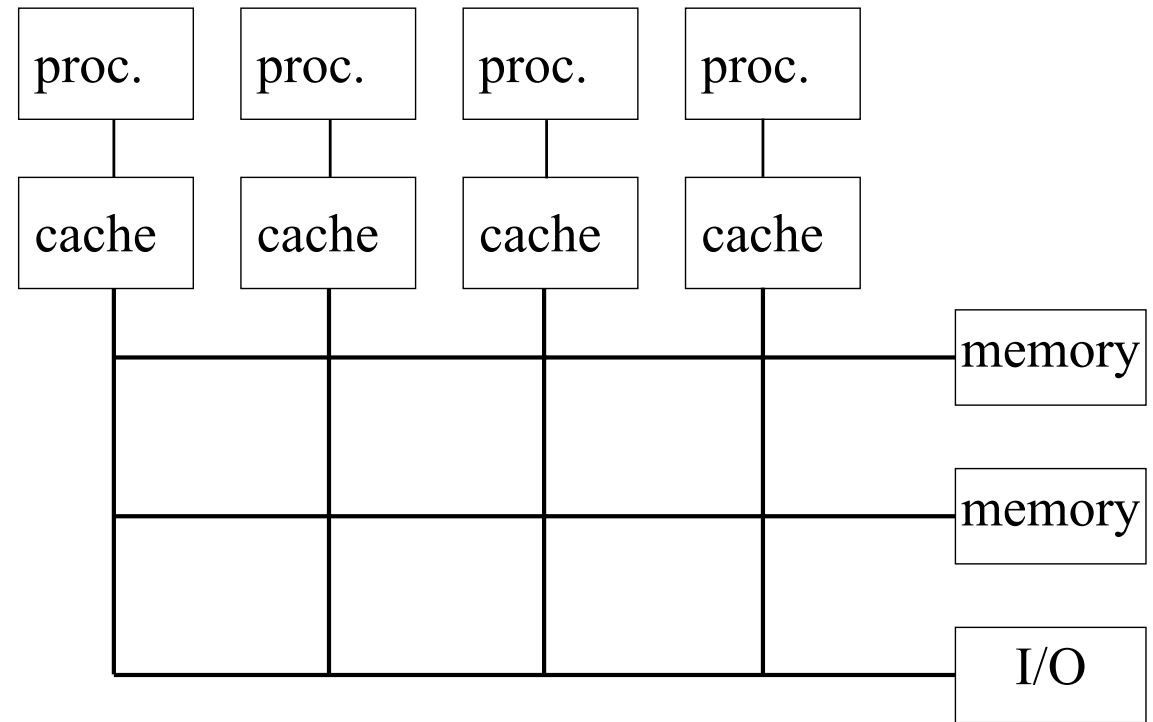
Brentin lauseen helpotus

- Jos algoritmimme toimii P prosessorilla ajassa T , voimme suorittaa sen $P' < P$ prosessorilla ajassa $T \times \lceil P/P' \rceil$.
- ⇒ Voimme aina huoletta suunnitella algoritmimme niin usealle prosessorille kuin pystymme. Algoritmi toimii myös vähemmällä prosessoreilla, jopa yhdellä.

Nykyiset (kaupalliset) rinnakkaiskoneet

SMP (Symmetric MultiProcessor)

- 2-16 (-64) prosessoria **samassa muisti**väylässä, (tai muussa kytkennässä)
- usea muistipankki, omat välimuistit
- skaalautuvuus huono .. huonohko
- huom: prosessorit (käyttäjän ohjelmat) eivät viesti suoraan keskenään, vaan käyttävät muistia kommunikatiovälineenä



Moniydin, monisäie prosessorit

- Hyödynnetään kasvavaa **transistorimäärää**.
- **Muistiväylä** pullonkaulana.
- Tulee olemaan jokapäiväistä:
 - Intel: 90% 2007 loppuun mennessä.
 - Xbox360: 3 ydintä, 2 säiettä kummassakin.
 - PS3: 9 ydintä.
 - Sun T1 (2005): 8 ydintä, 4 säiettä kussakin, T2 (2007): 16×4.
 - nVidia G8-sarja: 8-128 ydintä, jopa 64 säiettä kussakin.
- Miksi **monisäiesuoritusta**?
 - Prosessorin toiminnalliset yksiköt suorittavat kutakin h säiettä h -kertaa maksimia hitaammin.
 - Säikeen peräkkäisten käskyjen keskinäiset riippuvuudet eivät huononna prosessorin (ALUjen) käyttöastetta.
 - Kunkin säikeen muistiviittaukset hoituvat "nopeammin" (käskyinä).

- 1-32 huipputehokasta prosessoria
- Kukin jopa 20 GFLOPS
- Jokaisella kellojaksolla jopa 16 liukulukukäskyä
- Esim. pistetulo
- Vaatii pitkän vektorin (taulukko)
- Jokaisella kellojaksolla jopa 16 sanaa keskusmuistista
- Ei välimuisteja, vaan raudalla toteutettu **etukäteishaku ja leveät muistiväylät** (sekä SRAM muisti)
- Cray, Hitachi, Fujitsu, **NEC**.
- Erittäin kalliita, jopa tehoonsa nähden.
- Sukupuutto uhkaa.

MPP (Massively Parallel Processing)

- kymmeniä .. tuhansia prosessoreita (jopa satoja tuhansia)
- **laskentasolmussa** 1-4 prosessoria (SMP), muisti
 - prosessorit massatavaraa (edullisia) (Opteron, Xeon)
- erillisiä I/O solmuja tarpeen mukaan
- laskentasolmut kytketty toisiinsa kommunikaatioverkolla, topologia vaihtelee
- usein rauta tukee virtuaalista yhteistä muistia
- **skaalautuu** riittävästi
- **kommunikaatioverkko** on kallis (jopa puolet koneen hinnasta)
- Erikoiskäyttöisissä koneissa kommunikaatiokapasiteetti, topologia, muisti, I/O, jne voidaan mitoittaa juuri sopivaksi.
- Yleiskäyttöisissä koneissa kaikkea olisi oltava riittävästi (hintaan nähden).
- Esim Cray XT5, SGI Altix, IBM eServer, jne

NOW (Network of Workstations)

- Henkilökohtaiset työasemat ovat 99% jouten (yöt, editorin käyttö)
 - Joutava CPU-aika hyötykäyttöön: **nice laske**.
- **"Ilmaista" tehoa!!!**
- Tavallisia (UNIX) työasemia, TCP/IP
- Yksi keskitin/kytkin ... LAN ... WAN ... Internet
- Joskus myös tätä tarkoitusta varten kasattu ryväs (cluster)
 - 100[0] Mb Ethernet, ei näyttöjä, jne
- Kommunikaation hitaus rajoittaa algoritmivalikoimaa
- JoY: 2000 konetta = \approx 1 TFLOPS.
- Suomi: 1,5M konetta = \approx 500 TFLOPS
 - Vrt. louhi.csc.fi 10TFLOPS (\rightarrow 70 TFLOPS)

\Rightarrow Halvin FLOPS massatuotannon ansiosta.

Erilaiset rinnakkaiskoneet tuntuvat **lähestyvän toisiaan.**

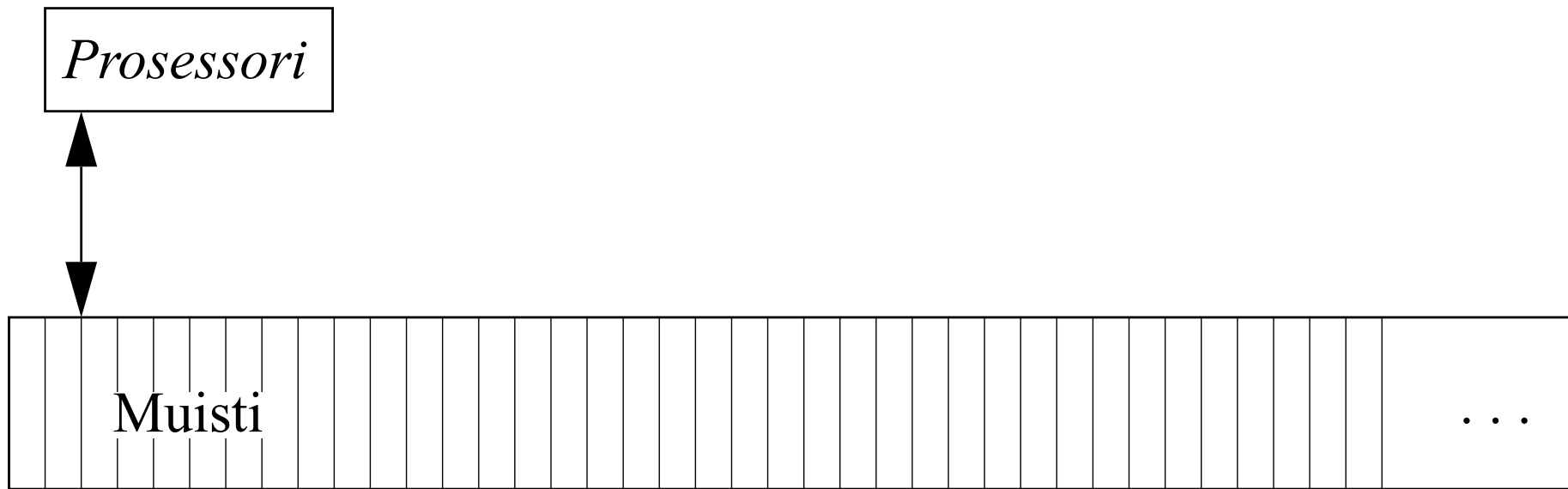
- SMP-koneissa väylä **korvataan verkolla.**
- Vektorikoneissa **proessorimäärä kasvaa**, ja niitä voidaan ryvästää.
- Vektorikoneissa käytetään CMOS tekniikkaa ja DRAM:a.
- MPP-koneissa (ja PC:ssä) otetaan käyttöön **vektorointitekniikoita** ja virtuaalinen yhteinen muisti (ja SMP solmut).
- MPP koneita rakennetaan **työasemakomponenteista.**
- NOW ryväksiä rakennetaan rinnakkaislaskentaan.
- **Korttipalvelinräkit** (Blade) näyttävät suurkoneelta.
- Näytönohjainten grafiikkaprosessoreita (jopa 128 ydintä/lastu) käytetään rinnakkaislaskentaan.
- Ilmastonmuutosaikana tärkeä yksikkö on MFLOPS/W.
 - Tehoprosessori ottaa 100W, virransyöttö, muistit, jne vievät omansa.
 - Tietokoneen käyttämä **watti tuplautuu rakennuksen jäähdytyksessä.**
 - Yksi prosessori voi kuluttaa sähköä 2000e edestä!
 - $250W \times 24h \times 365pv \times 5v \times 0,2e/kWh.$

⇒ "Yksinkertainen rinnakkaisuuden malli"

- Käytetään jottei tarvitsisi puuttua hankaloihin yksityiskohtiin

Tuttu tapa ohjelmoida:

- RAM (Random Access Machine)



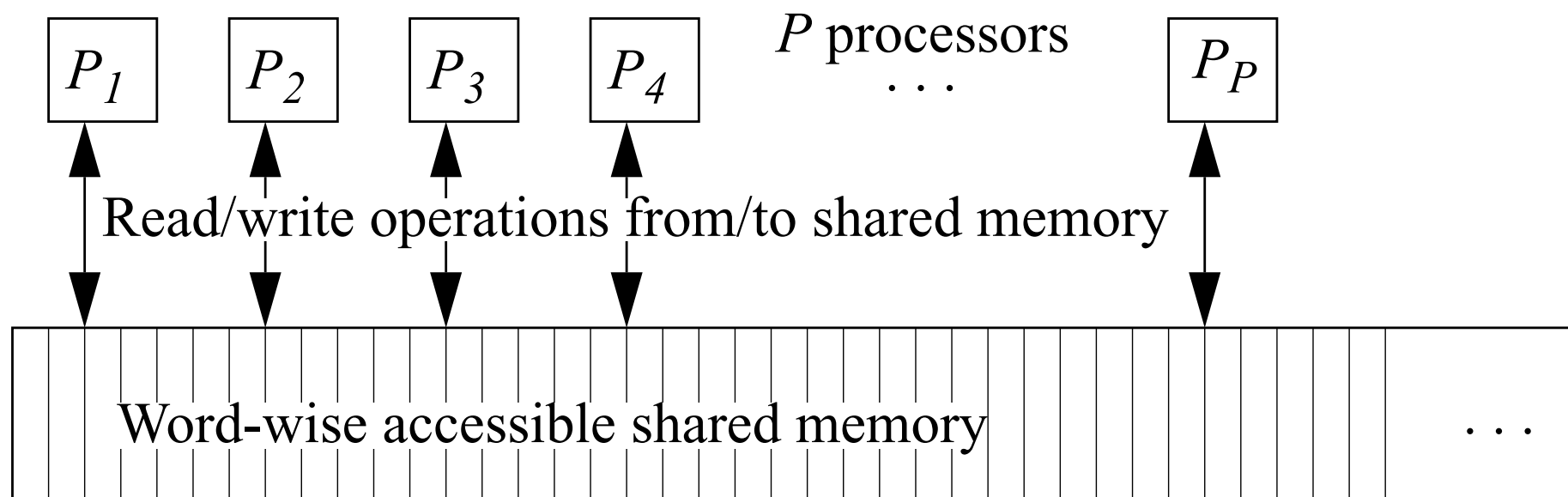
- Proseduraalinen ohjelmointi, erityisesti muuttujat

Luonnollinen laajennus:

- PRAM (Parallel Random Access Machine)
- Fortune and Wyllie 1978, monet muut

⇒ Lisätään prosessoreiden määrää.

- Kaikilla prosessoreilla **pääsy yhteiseen muistiin**



- Proseduraalinen ohjelmointi, yhteiset muuttujat
- Kaikkia prosessoreja on ohjelmoitava

Hyvää:

- **Yksinkertainen**
- Vahva malli
- Jos rinnakkaisalgoritmi yleensäkin voidaan tehdä, se voidaan tehdä PRAM:lle
- **Muistuttaa hieman oikeita** tietokoneita (kuten RAM:kin)
- Muunneltava tarpeen mukaan
- PRAM:sta on olemassa kymmeniä versioita
- **Yleisesti käytetty**
- Lähes kaikki rinnakkaisalgoritmit on suunniteltu PRAM:lle
- Valmiiden hyvien rinnakkaisalgoritmien joukko on suuri

Huonoa:

- P -porttista yhteistä muistia ei pystytä rakentamaan (suoraan, edullisesti).
- Todellisen tietokoneen **viiveet jätetään huomiotta**
- Ei ota huomioon komponenttien kustannuksia
- **Ei kannusta resurssien säästöön**

Kuitenkin

- Kätevä tutkimus- ja opetusmalli (väline)
- **Algoritmit useimmiten muunnettavissa** todellisille koneille

⇒ Jotta prosessorit voisivat tehdä työtä tehokkaasti, on kullakin niistä oltava mahdollisuus käyttää yhteistä muistia jatkuvasti (millä tahansa kellojaksolla).

- Kaikki prosessorit **viittaavat yhteiseen muistiin samalla hetkellä**, voidaan tällaista määritellä/toteuttaa?
 - Määritellä : kyllä.
 - Toteuttaa: tietyin ehdoin, kunhan muistipankkeja on riittävästi, esim. $M = 16P$.
- Mallissa muisti on yhtenäinen, oletamme, että viittaukset pystytään käsittelemään **vakioajassa**.
 - Kukin konekäsky, mukaan lukien muistiviittaukset, vievät yhden kellojakson.
 - Tämä on mahdoton suoralla toteutuksella yli 10MHz nopeudella.
 - Erittäin syvällä liukuhihnauksella (ylikuormituksella) toteutus on mahdollinen, joskin (virtuaalisten) prosessoreiden määrä kasvaa huomattavasti.

- Entä jos **muistiviittaukset osuvat samaan muistipankkiin** tai jopa samaan **osoitteeseen**?
 - Samaan pankkiin, eri osoitteeseen
 - Mallissa ei väliä, todellisella koneella ongelmia
 - Samaan osoitteeseen
 - **Viittaukset voidaan ehkä yhdistää**. Kirjoittaminen: jotain kirjoitetaan, lukeminen: kopioidaan tulos.
 - Yhdistämisen toteuttaminen vaatii älyä muistinhallintayksiköltä (kutakin muistipankkia kohti)
- **Mallissa** samassa muistipaikassa samalla kellojaksolla tapahtuvat asiat on erikseen määriteltävä:
 - Samalla kellojaksolla lukeminen on vahva operaatio, mutta helppo määritellä
 - Kirjoittaminen ja lukeminen(-sia) samasta muistipaikasta voidaan määritellä vaikkapa siten, että kirjoitus tapahtuu ennen lukemista.
 - Useampi kirjoitus samaan paikkaan sensijaan on hankalampi määritellä.
 - Kussakin muistipaikassa on kerrallaan vain yksi arvo!

- EREW (Exclusive Read, Exclusive Write)
 - Sekä useampi lukeminen, että useampi kirjoittaminen samanaikaisesti on kiellettyä.
- CREW (Concurrent Read, Exclusive Write)
 - Usea prosessori saa lukea samanaikaisesti, mutta kirjoittaa vain yksi kerrallaan
- CRCW (Concurrent Read, Concurrent Write)
 - Sekä lukeminen, että kirjoittaminen ovat sallittuja samanaikaisesti.
 - Kirjoitusten samanaikaisuus on ratkaistava jotenkin!
- CROW (Concurrent Read, Owner Write)
 - Kukin muistipaikka on jonkin prosessorin "omistuksessa", muut saavat vain lukea.

- WEAK
 - vain samanaikainen nollien kirjoitus sallitaan
- COMMON
 - vain samanaikainen saman arvon kirjoitus sallitaan
- TOLERANT
 - samanaikainen mitään ei tapahdu jos usea yrittää kirjoittaa yhtäaikaan
- COLLISION
 - erityinen törmäyssymboli kirjoitetaan jos usea yrittää kirjoittaa yhtäaikaan
- COLLISION+
 - erityinen törmäyssymboli kirjoitetaan jos usea yrittää kirjoittaa yhtäaikaan eri arvoa (vrt. COMMON)
- ARBITRARY
 - jokin (satunnainen) arvoista jää jäljelle jos usea yrittää kirjoittaa yhtäaikaan

- **PRIORITY**
 - prioriteetiltaan (PID) paras prosessori onnistuu, muut eivät
- **STRONG**
 - Yhdistelmä operaatioista suoritetaan
 - ADD&WRITE tms
 - Eri variaatioita

Esimerkkejä tehoeroista:

- Yhden tiedon levitys kaikille
 - CREW: kaikki lukevat saman muistipaikan: $O(1)$
 - EREW: arvoa kahdennetaan kunnes kaikki ovat lukeneet sen: $O(\log P)$
- Maksimin haku vektorista
 - CREW: $O(\log N)$
 - WEAK CRCW: $O(1)$
- Lajittelu
 - EREW: $O(\log N)$
 - STRONG CRCW: $O(1)$

- Kuten peräkkäisohjelmoinnissakin, käytetään useaa abstraktiotasoa
 - Kuvataan algoritmi suomeksi
 - Kuvataan algoritmi algoritminotaatiolla
 - Kirjoitetaan algoritmi ohjelmointikielellä
 - Käännetään ohjelma konekielelle

Rinnakkainen algoritminotaatio

- Kuten peräkkäinenkin, lisäksi

```

for  $i \in 1..N$  pardo           // tai esim. for each element      1
    lause;                       // esim. if A[i] = 0 then A[i] := ...  2

```

- *lause* suoritetaan kerran kullekin i :n arvolle $1..N$.
- suorituskerrat suoritetaan rinnakkain (mahdollisuuksien mukaan)
- aika: $T_{lause} + O(1)$ jos prosessoreita riittävästi
- $\lceil T_{lause}/P \rceil + O(1)$ kun **huomioimme** P :n.
- suorituskerrat eivät saisi häiritä toisiaan

```

for  $i \in 1..N$  pardo           1
    A[A[i]] = A[i];           // tulos epäselvä !?      2

```

- Jos tarvitsemme kullekin prosessorille paikallisia muuttujia (muistia), voimme käyttää avainsanoja *private* ja *shared* tarpeen mukaan selventämään tilannetta.

⇒ Pyritään maksimaaliseen nopeutukseen (ja rinnakkaisuuteen) työoptimaalisuuden puitteissa.

Peräkkäisalgoritmin riippumattomien osien **rinnakkaistus**

- *for-do* silmukoiden (tai muuten peräkkäisten osien) analysointi
- jos peräkkäiset osat ovat riippumattomia, voidaan peräkkäisyys muuttaa rinnakkaisuudeksi
- joskus sisemmät, joskus ulommat *par-do* -silmukat ovat rinnakkaisia
- silmukoiden uudelleenjärjestely saattaa auttaa

- Jaetaan syöte kahtia ja **ratkaistaan osatehtävät rinnakkain**, toistetaan rekursiivisesti
 - tuttu tekniikka jo peräkkäisohjelmoinnista
 - rekursio lopetetaan joko kun syöte on triviaali (kuten peräkkäisohjelmoinnissa), tai kun käytettävien prosessoreiden määrä on 1, jolloin voimme vaihtaa mahdollisesti tehokkaampaan peräkkäisalgoritmiin
- Osatehtävien tulokset yhdistetään "suuremmiksi" tuloksiksi rekursiosta palattaessa kuten peräkkäisohjelmoinnissakin.
- Osatulosten **yhdistäminen on syytä tehdä myös rinnakkaisesti**.
 - yhdistämisen rinnakkaistaminen on useimmiten vaikeampaa kuin jakaminen

- Jätetään välistä syötteen rekursiivinen jakaminen osiin ja aloitetaan suoraan yhden kokoisista paloista.
- Verrataan syötteitä keskenään pareittain ($N/2$ paria), voittanut siirtyy seuraavalle kierrokselle.
- "Voittanut" sanan määritelmä riippuu sovelluksesta, joskus se voi olla myös kahden syötteen yhdiste.
- $N/2$ prosessorilla vakioajassa/kierros.

- Mahdollisimman nopeasti (melkeinpä: "mahdollisimman monella prosessorilla").
- Pyritään laskemaan **kaikki mahdollisuudet kerralla**.
- Esimerkiksi: verrataan kaikkia pareja yhtäaikaan, $O(N^2)$ vertailua $O(1)$ ajassa $O(N^2)$ prosessorilla.
- N alkion syötteestä syntyy N^2 osatulosta.
- Tulosten yhdistäminen helpointa CRCW-mallissa.
- Pyrkimys vakioaikaiseen tai logaritmiseen aikavaativuuteen.

- Edellä esitetyt keinot käyttävät prosessoreita usein epätasaisesti.
 - esimerkiksi turnauksen alussa käytetään $N/2$ prosessoria, mutta jokaisella kierroksella tarvittavien prosessoreiden määrä vähenee
 - näin ei yleensä päästä työoptimaalisuuteen
- Rajoitetaan rinnakkaisuutta sopivasti työoptimaalisuuteen pääsemiseksi
- Idea:
 - Käytetään hieman vähemmän prosessoreita,
 - aloitetaan isommista lohkoista,
 - käsitellään ne "rinnakkain peräkkäin", ja
 - aloitetaan nopea rinnakkaisalgoritmi vasta kun kullakin prosessorilla on enää yksi syöte.

- Esimerkiksi turnauksessa:
 - Käytetään vain $N/\log N$ prosessoria,
 - kukin etsii ensi peräkkäin omasta $\log N$ lohkostaan voittajan $O(\log N)$ ajassa, ja
 - jäljellejääneet $N/\log N$ alkiota käsitellään rinnakkaisella turnauksella $O(\log N)$ ajassa $N/\log N$ prosessorilla.
 - Koko algoritmi $O(\log N)$ ajassa $N/\log N$ prosessorilla, $O(N)$ työ.

Muut keinot

- Usein edellisiä keinoja yhdistellään, erityisesti lohkomista muihin.
- Jotkin rinnakkaisalgoritmit ovat vain "uskomatonta, mutta totta" ideoita, jotka käyttävät aivan omaa tekniikkaansa.
- Joitain perusalgoritmeja, erityisesti alkusummaa, voidaan käyttää laajempien algoritmien osina.
- Satunnaistus (säännöllisyyden rikkominen).
- Näytteistys: otetaan syötteestä edustava otos, analysoidaan se (raa'alla voimalla), jaetaan koko syöte osiin otoksen osoittamalla tavalla.
 - Syöte saadaan jakautumaan tasaisemmin prosessoreille.

- ⇒ Yhteinen muisti on mukava abstraktio, mutta ei kovin tehokkaasti/edullisesti toteutettavissa.
- Jos yhteistä muistia ei ole eikä sitä kannata simuloida, prosessit joutuvat kommunikoimaan muuten.
 - Viestinvälitysmallissa prosessit (prossessorit) kommunikoivat suoraan keskenään vaihtamalla kahdenkeskisiä viestejä.

Sovellusala

- Viestinvälitystekniikoita käytetään hyvin monella tasolla emolevyjen väylistä sähköpostijärjestelmiin.
- Seuraavassa puhutaan lähettäjistä ja vastaanottajasta abstraktimmin. Muistetaan, että toimija voi olla vaikkapa prosessori tai prosessi.
- Eri järjestelmät mahdollistavat erilaisia viestinvälitysoperaatioita jotka eroavat toisistaan paljonkin.

Viestinvälitysoperaatiot send ja receive

- Prosessori/ssi lähettää/vastaanottaa sanoman toiselle/lta.

`send(dest_pid, start_addr, length, mode);`

`receive(buffer_addr);`

- Variaatioita on paljon, useimmat järjestelmät antavat vaihtoehtoja.

Määrätty lähettäjä vastaanotossa vai ei?

- Lähettäjän on epäilemättä tavalla tai toisella kerrottava minne viesti lähetetään.
- Vastaanottaja sen sijaan mahdollisesti voi joko odottaa viestiä juuri tietyltä lähettäjältä, tai kelpuuttaa viesti miltä tahansa lähettäjältä. Tai jotenkin rajatulta lähettäjäjoukolta.

Lähettäjän/vastaanottajan identifiointitapa

- Käytetäänkö **absoluuttista prosessori/ssi numeroa**, vai jotain abstraktimpaa ohjelman käynnös/käynnistys/ajon aikana annettua osoitetta?
- Voiko uusia kommunikaatiokohteen numeroita luoda ohjelman suorituksen aikana **välittää tällaisia loogisia osoitteita** esim. aliohjelmien parametreina tai peräti viestien mukana?

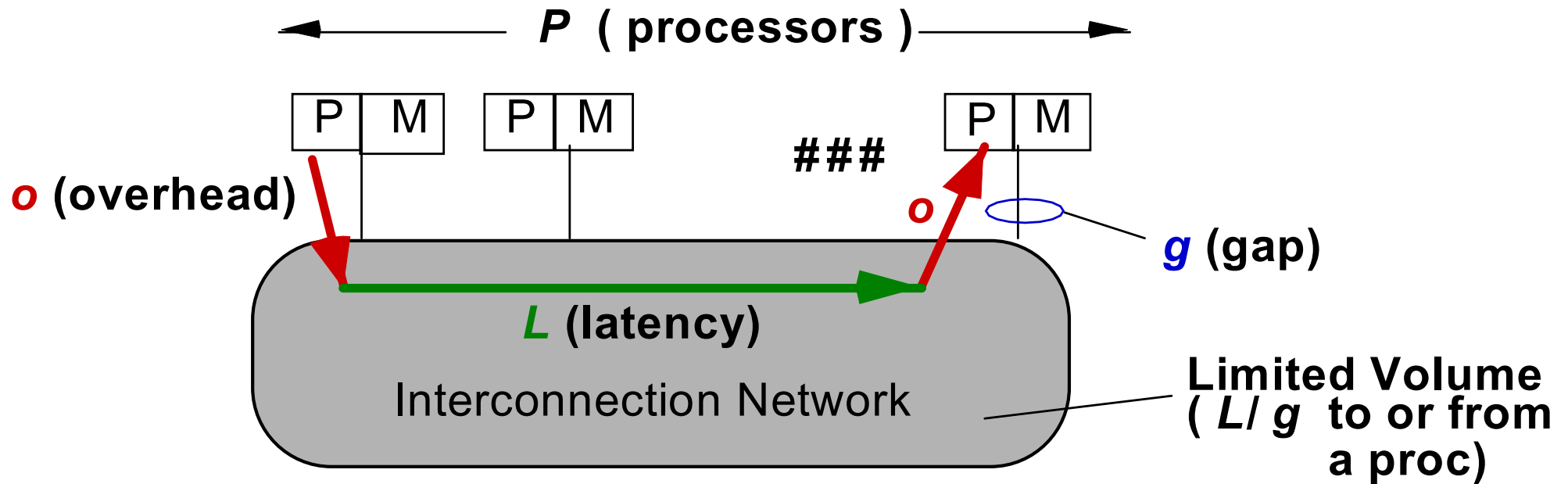
Kommunikaatiotopologia

- Voiko mikä tahansa prosessori suoraan lähettää viestin **mille tahansa** prosessorille?
- Vai onko käytössä vain **rajattu määrä kommunikaatiokanavia** koko järjestelmässä (tai prosessoria kohti)?
- Mihin muotoon (topologiaan) kommunikaatiokanavat on järjestetty? Voidaanko sitä muuttaa?
- Käytetäänkö rajoitetussa topologiassa edelleen globaaleita prosessorinumeroita vai paikallisia määreitä (esim. *N*, *E*, *S*, *W*)?

Puskurointi, pysähtymiset, kysely

- Mitä tapahtuu, jolleivät lähettäjä ja vastaanottaja suorita operaatioita samanaikaisesti?
- Jos lähettäjä on edellä, **jääkö lähettäjä odottamaan** vastaanottajan vastaanottovalmiutta (blocking send), vai **puskuroituuko** tieto jonnekin (non-blocking send)?
- Puskuroituuko viesti **lähettäjän** lähetyspuskuriin **vai vastaanottajan** vastaanottopuskuriin?
- Entä jos **puskurit täyttyvät** tai verkossa on ruuhkaa, jääkö lähettäjä sitten odottamaan, vai häviääkö viesti?
- Samat vaihtoehdot valittavana, jos viesti on pitkä ja kommunikaatioväylä hidas.
- Jos **vastaanottaja on edellä**, jääkö se odottamaan lähetystä, vai voiko se tehdä jotain muuta välillä?
- Miten **vastaanottaja saa tiedon** lopulta tulevasta viestistä (tai siitä onko viestiä yleensäkin tulossa)?
- Voiko vastaanottaja asettaa puskurin "kunhan se viesti lopulta tulee, niin antaa tulla tähän"?

- Hieman vaikeampi arvioida kuin yhteiselle muistille, esim LogP:.



Viestinvälityksen ja yhteisen muistin välimuotoja

- **remote memory access** (RMA): kutsu lukea/kirjoittaa toisen prosessorin paikalliseen muistiin
- **remote procedure call** (RPC): kutsu käynnistää toisessa prosessorissa aliohjelma (parametrit kutsun mukana)

- *for .. pardo* soveltuu paremmin tietorinnakkaiseen ohjelmointiin.
- Yleensä **proessorit/ssit käynnistetään kerralla** (esim. *mpi_init()*), jolloin määrätään kunkin prosessin prosessi-id ja sijaintiprosessori.
 - Tällöin uusia prosessoreita ei voi ottaa dynaamisesti käyttöön.
- Joissakin järjestelmissä uusia prosesseja voi perustaa *fork()*:lla tai jopa jollakin yksinkertaisemmalla tavalla.
- Rinnakkaisuus on usein huomattavasti **karkeajakoisempaa** kuin PRAM-ohjelmoinnissa.
- Useimmiten käynnistyy aluksi vain yksi prosessi (master) joka käynnistää muut prosessit (orjat, slave) suorittamaan samaa ohjelmaa.
 - Ohjelman alussa tarkastetaan ollaanko masterissa (käynnistetään muut) vai ollaanko orjana, jolloin muita ei käynnistetä.
 - Myös rekursiivinen puumainen prosessien käynnistäminen on mahdollista joissakin järjestelmissä, mutta harvemmin käytössä.

- **Viestinvälitys itsessään on synkronointia**, joten erillisiä synkronointioperaatioita ei tarvita.
- Lukkiumien (deadlock) estäminen on viestinvälitysohjelmoinnin tärkein asia.

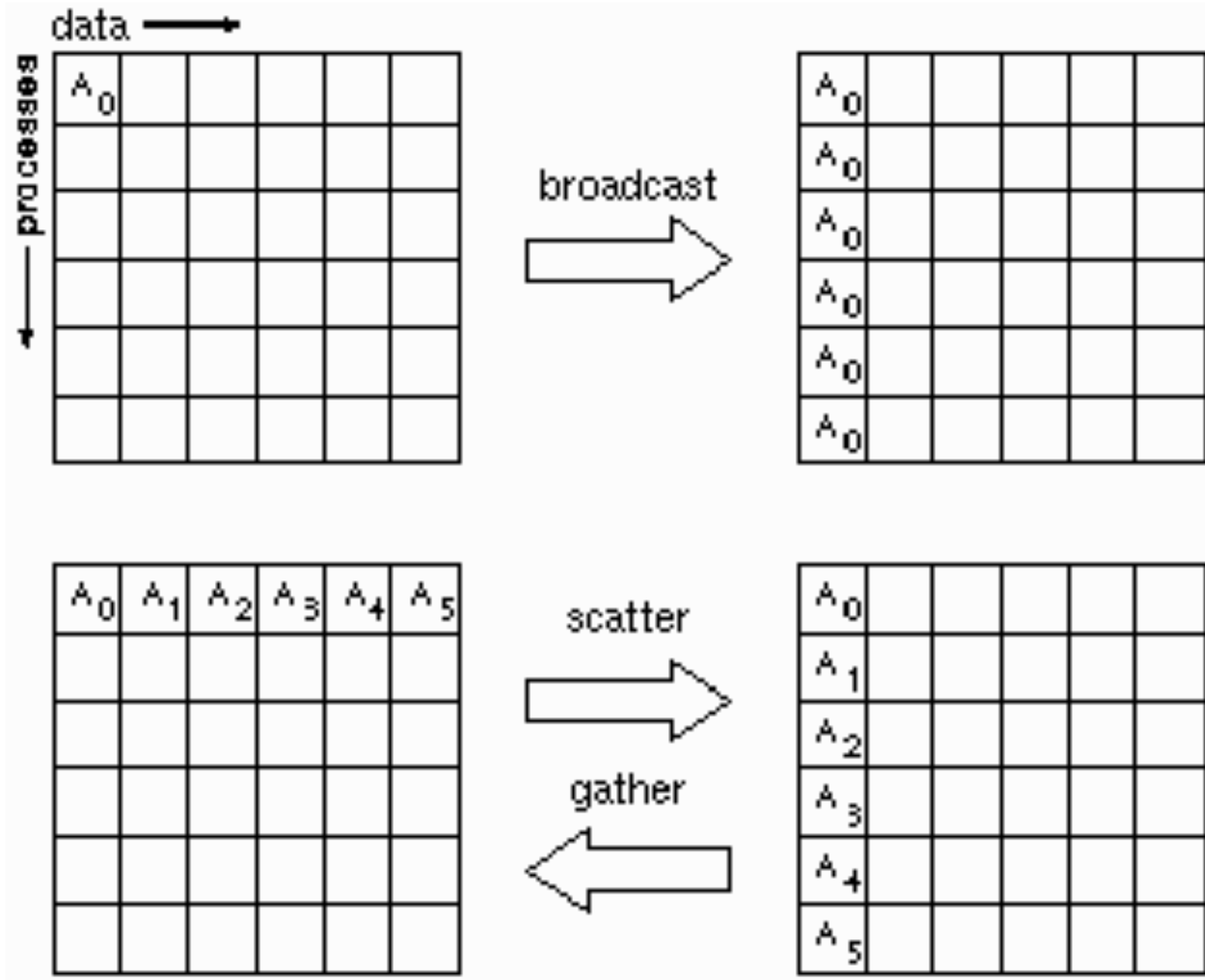
MPI (Message Passing Interface)

- Standardoitu rajapinta viestinvälitysohjelmointiin.
- C, C++, Fortran 77&90
- Useimmille alustoille useita toteutuksia.
 - <http://www.open-mpi.org/>
 - Rinnakkaiskonevalmistajilla omat optimoidut toteutukset
- *MPI_init()* luo prosessit
- send, receive puskuroiden, synkronisesti, jne.

`MPI_Send(buffer, count, datatype, dest, tag, comm)`

`MPI_Recv(buf, count, datatype, src, tag, comm, status)`

- yhdeltä kaikille, kaikilta yhdelle, kaikilta kaikille -operaatiot.
- <http://www.mpi-forum.org/>
- <http://www-unix.mcs.anl.gov/mpi/>



A ₀					
B ₀					
C ₀					
D ₀					
E ₀					
F ₀					

allgather



A ₀	B ₀	C ₀	D ₀	E ₀	F ₀
A ₀	B ₀	C ₀	D ₀	E ₀	F ₀
A ₀	B ₀	C ₀	D ₀	E ₀	F ₀
A ₀	B ₀	C ₀	D ₀	E ₀	F ₀
A ₀	B ₀	C ₀	D ₀	E ₀	F ₀
A ₀	B ₀	C ₀	D ₀	E ₀	F ₀

A ₀	A ₁	A ₂	A ₃	A ₄	A ₅
B ₀	B ₁	B ₂	B ₃	B ₄	B ₅
C ₀	C ₁	C ₂	C ₃	C ₄	C ₅
D ₀	D ₁	D ₂	D ₃	D ₄	D ₅
E ₀	E ₁	E ₂	E ₃	E ₄	E ₅
F ₀	F ₁	F ₂	F ₃	F ₄	F ₅

alltoall



A ₀	B ₀	C ₀	D ₀	E ₀	F ₀
A ₁	B ₁	C ₁	D ₁	E ₁	F ₁
A ₂	B ₂	C ₂	D ₂	E ₂	F ₂
A ₃	B ₃	C ₃	D ₃	E ₃	F ₃
A ₄	B ₄	C ₄	D ₄	E ₄	F ₄
A ₅	B ₅	C ₅	D ₅	E ₅	F ₅

- [1] Coulouris, Dollimore, Kindberg: Distributed Systems – Concepts & Design, 3rd ed. 2001. Addison Wesley. <http://www.cdk3.net/>
- [2] Stevens: Unix Network Programming. <http://www.kohala.com/start/unp.html>.
- [3] Schneier: Applied Cryptography, 2nd ed. 1996. Wiley.
- [4] Tanenbaum, van Steen: Distributed Systems – Principles and Paradigms. 2002. Prentice Hall.
- [5] rfc3117 On the Design of Application Protocols.
- [6] Leue S.: Distributed Systems, U of Freiburg.
- [7] Eles: TDDB 37 Distributed Systems Course. <http://www.ida.liu.se/~TDDB37/>

[8] <http://pandonia.canberra.edu.au/ClientServer/>