

Elosztott rendszerek: Alapelvek és paradigmák Distributed Systems: Principles and Paradigms

Maarten van Steen¹ Kitlei Róbert²

¹VU Amsterdam, Dept. Computer Science

²ELTE Informatikai Kar

1. rész: Bevezetés

2015. május 24.

Tartalomjegyzék

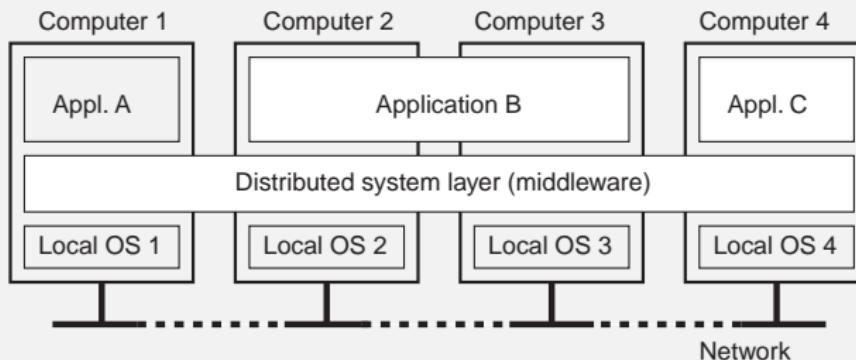
Fejezet
01: Bevezetés
02: Architektúrák
03: Folyamatok
04: Kommunikáció
05: Elnevezési rendszerek
06: Szinkronizáció
07: Konzisztencia & replikáció
08: Hibatűrés
10: Objektumalapú elosztott rendszerek
11: Elosztott fájlrendszerek
12: Elosztott webalapú rendszerek

Definíció: Elosztott rendszer

Az elosztott rendszer

önálló számítógépek olyan összessége, amely kezelői számára egyetlen koherens rendszernek tűnik.

Két szempont: (1) független számítógépek és
(2) egyetlen rendszer ⇒ **köztesréteg (middleware)**.



Az elosztott rendszer céljai

- Távoli erőforrások elérhetővé tétele
- Átlátszóság (distribution transparency)
- Nyitottság (openness)
- Skálázhatóság (scalability)

Átlátszóság

Fajta	Angolul	Mit rejt el az erőforrással kapcsolatban?
Hozzáférési/elérési	Access	Adatábrázolás; elérés technikai részletei
Elhelyezési	Location	Fizikai elhelyezkedés
Áthelyezési	Migration	Elhelyezési + a hely meg is változhat
Mozgatási	Relocation	Áthelyezési + használat közben is történhet az áthelyezés
Többszörözési	Replication	Az erőforrásnak több másolata is lehet a rendszerben
Egyidejűségi	Concurrency	Több versenyhelyzetű felhasználó is elérheti egyszerre
Meghibásodási	Failure	Meghibásodhat és újra üzembe állhat

Megjegyzés

A fentiek lehetséges követelmények, amelyeket a rendszerrel kapcsolatban támaszthatunk. A félév során megvizsgáljuk, hogy melyik mennyire érhető el.

Az átlátszóság mértéke

Mekkora átlátszóságot várhatunk el?

A teljes átlátszóságra törekvés általában túl erős:

- A felhasználók **különböző kontinenseken** tartózkodhatnak
- A hálózatok és az egyes gépek **meghibásodásának** teljes elfedése elméletileg és gyakorlatilag is **lehetetlen**
 - Nem lehet eldöntheti, hogy a szerver csak lassan válaszol vagy meghibásodott
 - Távolról nem megállapítható, hogy a szerver feldolgozta-e a kérésünket, mielőtt összeomlott
- A nagymértékű átlátszóság **a hatékonyság rovására megy**, de a késleltetést is el szeretnénk rejteni
 - Ilyen feladat lehet a webes gyorsítótárak (cache-ek) **tökéletesen** frissen tartása
 - Másik példa: minden változás azonnal lemezre írása nagymértékű hibatűréshez

Elosztott rendszerek nyitottsága

Nyitott elosztott rendszer

A rendszer képes más nyitott rendszerek számára szolgáltatásokat nyújtani, és azok szolgáltatásait igénybe venni:

- A rendszerek jól definiált **interfészekkel** rendelkeznek
- Az alkalmazások **hordozhatóságát** (portability) minél inkább támogatják
- Könnyen elérhető a rendszerek **együttműködése** (interoperability)

A nyitottság elérése

A nyitott elosztott rendszer legyen könnyen alkalmazható **heterogén** környezetben, azaz különböző

- hardvereken,
- platformokon,
- programozási nyelveken.

Nyitottság: követelményrendszerek, mechanizmusok

A nyitottság implementálása

- Fontos, hogy a rendszer könnyen cserélhető részeiből álljon
- Belső interfések használata, nem egyetlen monolitikus rendszer
- A rendszernek minél jobban paraméterezhetőnek kell lennie
- Egyetlen komponens megváltoztatása/cseréje lehetőleg minél kevésbé hasson a rendszer más részeire

Nyitottság: követelményrendszerek, mechanizmusok

Példák

A nyitott rendszer **követelményeket** (policy) ír elő, amelyekhez jó, ha minél több megvalósító **megvalósítás** (mechanism) érhető el a rendszerben. Néhány szóbajöhető példa:

- Mennyire erős konzisztenciát követeljünk meg a kliensoldalon cache-elt adatokra? Legyen dinamikusan beállítható, hogyan döntse el a rendszer az adatról, milyen erősen legyen konzisztens.
- A letöltött kód milyen műveleteket hajthasson végre? A mobil kódhoz rendeljünk különböző megbízhatósági szinteket.
- Milyen QoS követelményeket támasszunk változó sávszéleségű rendszerben? minden folyamra külön lehessen QoS követelményeket beállítani.
- Milyen mértékben titkosítsuk a kommunikációs csatornánkat? A rendszer kínáljon többfajta titkosítási mechanizmust, amelyek közül választani lehet.

Átméretezhetőség (scalability)

Átméretezhetőség

Ha egy „kis” rendszer megnő, az sokfajta kihívást jelenthet. Több különböző jellege is megnőhet a rendszernek:

- **méret szerinti átméretezhetőség:** több felhasználó és/vagy folyamat van a rendszerben
- **földrajzi átméretezhetőség:** a rendszert nagyobb területről veszik igénybe, pl. egyetemen belüli felhasználás → világméretű felhasználóbázis
- **adminisztrációs átméretezhetőség:** biztonsági, karbantartási, együttműködési kérdések merülnek fel, ha új adminisztrációs tartományok kerülnek a rendszerbe

Megjegyzés

A legtöbb rendszer a méret szerinti átméretezhetőséget kezeli.

(Nem minden) megoldás: erősebb szerverek használata.

A másik két jellegű átméretezhetőséget nagyobb kihívás kezelni.

Technikák az átméretezhetőség megvalósítására

A kommunikációs késleltetés elfedése

A válaszra várás közben más tevékenység végzése:

- **Aszinkron kommunikáció** használata
- A beérkező választ külön kezelő dolgozza fel
- **Probléma:** nem minden alkalmazás ültethető át ilyen megközelítésre

Technikák az átméretezhetőség megvalósítására

Elosztás

Az adatokat és a számításokat több számítógép tárolja/végzi:

- A számítások egy részét a kliensoldal végzi (Java appletek)
- Decentralizált elnevezési rendszerek (DNS)
- Decentralizált információs rendszerek (WWW)

Technikák az átméretezhetőség megvalósítására

Replikáció/cache-elés

Több számítógép tárolja egy adat másolatait:

- Replikált fájlszerverek és adatbázisok
- Tükrözött weboldalak
- Weboldalak cache-elése (böngészőkben, proxy szervereken)
- Fájlok cache-elése (a szerver- és kliensoldalon)

Átméretezhetőség – a probléma

Megjegyzés

Az átméretezhetőség könnyen elérhető, de ára van:

- Több másolat fenntartása (cache vagy replika) **inkonzisztenciához** vezet: ha módosítunk egy másolatot, az eltér a többitől.
- A másolatok konzisztensen tartásához **globális szinkronizációra** van szükség minden egyes változtatás után.
- A globális szinkronizáció viszont rosszul skálázható nagy rendszerekre.

Következmény

Ha feladjuk a globális szinkronizációt, akkor kénytelenek vagyunk bizonyos fokú inkonzisztenciát elviselni a rendszerünkben.

Az, hogy ez milyen mértékben elfogadható, **rendszerfüggő**.

Elosztott rendszerek fejlesztése: hibalehetőségek

Megjegyzés

Az elosztott rendszer környezetéről kényelmes lehet feltételezni, hogy megbízható. Ha ez **tévesnek bizonyul**, az a rendszer újratervezéséhez vezethet. Néhány ilyen feltételezés:

- a hálózat hibamentes
- a hálózat biztonságos
- a hálózat homogén
- a hálózati topológia nem változik
- a kommunikációnak nincsen időigénye
- a sávszélesség korlátlan
- a kommunikációnak nincsen költsége
- csak egy adminisztrátor van

Elosztott rendszerek fajtái

- Elosztott számítási rendszerek
 - grid
 - cloud
 - információs rendszerek
- Elosztott információs rendszerek
- Elosztott átható (pervasive, ubiquitous) rendszerek

Elosztott számítási rendszerek

Számítási rendszer

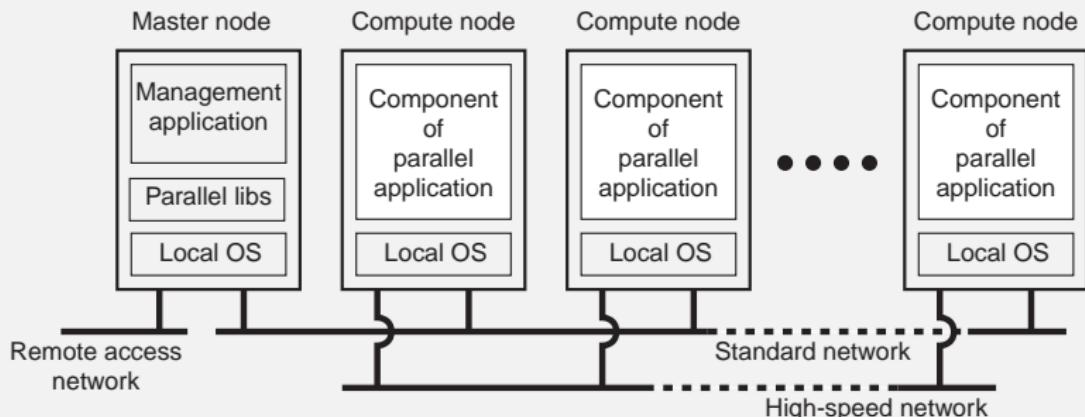
Sok elosztott rendszer célja számítások végzése nagy teljesítménnyel.

Cluster (fürt)

Lokális hálózatra kapcsolt számítógépek összessége.

- Homogén: ugyanaz az operációs rendszer, hardveresen nem vagy csak alig térnek el
- A vezérlés központosítva van, általában egyetlen gépre

Elosztott számítási rendszerek



Elosztott számítási rendszerek

Grid (rács)

Több gép, kevésbé egységesek:

- Heterogén architektúra
- Átívelhet több szervezeti egységen
- Nagyméretű hálózatokra terjedhet ki

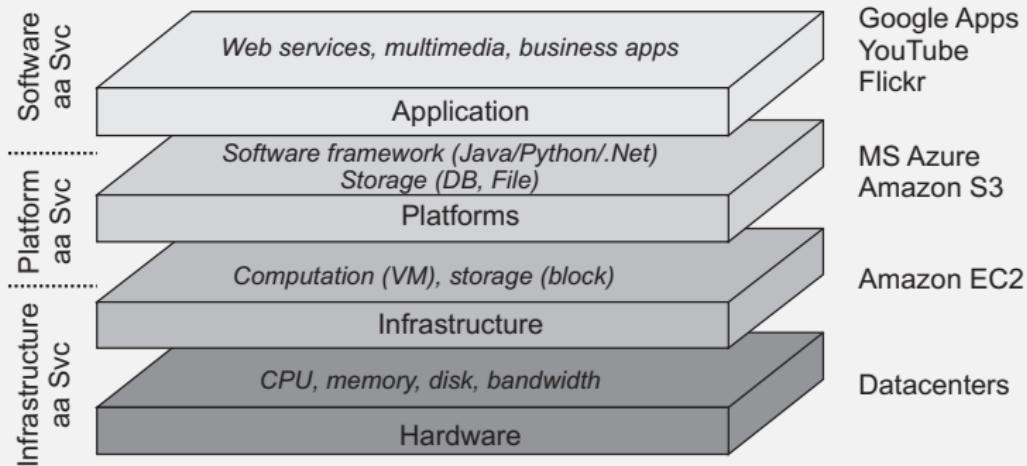
Megjegyzés

Az együttműködést sokszor **virtuális szervezetek** kialakításával segítik. Ez a felhasználókat csoportosítja, akiknek így egységesen lehet erőforrásokat kiutalni.

Elosztott számítási rendszerek

Felhő (cloud)

Többrétegű architektúra.



Elosztott számítási rendszerek

Felhő

Négy különböző réteg:

- **Hardver**: Processzorok, útválasztók (routerek), áramforrások, hűtőberendezések. A felhasználók közvetlenül nem látják.
- **Infrastruktúra**: Virtuális hardvert tesz elérhetővé: szerver, adattároló, hálózati kapcsolat, számítási kapacitás lefoglalása és kezelése.
- **Platform**: Magasabb szintű absztrakciókat biztosít. Pl. az Amazon S3 társzolgáltatás különböző fájlműveleteket biztosít; a felhasználónak vörrei (bucket) vannak, ebbe feltölthet, letölthet stb. fájlokat egy API segítségével .
- **Alkalmazás**: A végfelhasználónak szánt, jellemzően grafikus felületű alkalmazások.

Elosztott információs rendszerek

Elosztott információs rendszerek

Sok elosztott rendszer elsődleges célja adatok kezelése, illetve meglevő ilyen rendszerek elérése. **Példa:** tranzakciókezelő rendszerek.

```
BEGIN_TRANSACTION(server, transaction)
READ(transaction, file-1, data)
WRITE(transaction, file-2, data)
newData := MODIFIED(data)
IF WRONG(newData) THEN
    ABORT_TRANSACTION(transaction)
ELSE
    WRITE(transaction, file-2, newData)
    END_TRANSACTION(transaction)
END IF
```

Elosztott információs rendszerek: tranzakciók

Modell

A tranzakció adatok összességén (adatbázis, objektumok vagy más adattár) végzett művelet (lehetnek részműveletei), melynek az alábbi tulajdonságai vannak. A kezdőbetűk rövidítéséből **ACID**-nek szokás nevezni a követelményrendszeret.

Oszthatatlan, elemi (atomicity): Vagy a teljes tranzakció végbemegy minden részműveletével, vagy pedig az adattár egyáltalán nem változik meg.

Konzisztens (consistency): Az adattárra akkor mondjuk, hogy érvényes, ha (az adattárra jellemző) feltételek teljesülnek rá. A tranzakció konzisztens, ha érvényes állapotot állít elő. Ez a követelmény csak a tranzakció végére vonatkozik: menet közben előállhat érvénytelen állapot.

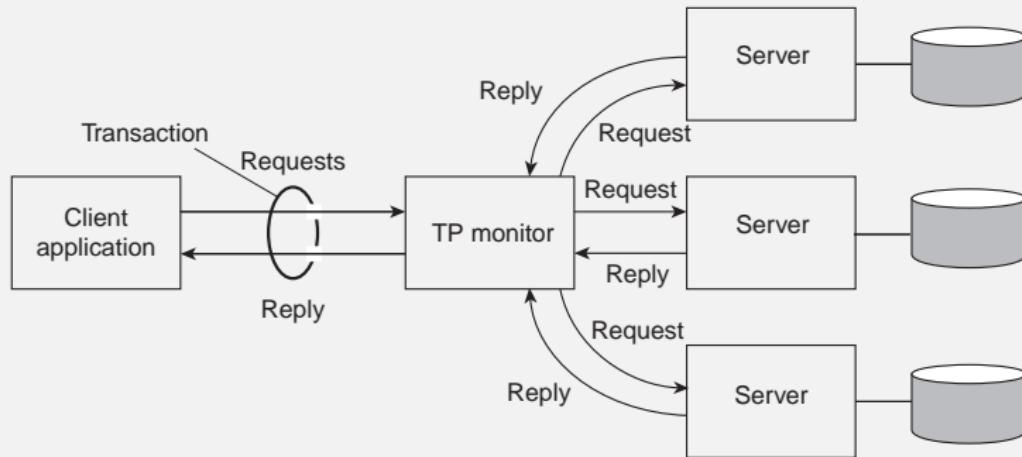
Elkülönülő, sorosítható (isolation): Egyszerre zajló tranzakciók "nem zavarják" egymást: olyan eredményt adnak, mintha egymás után sorban futottak volna le.

Tartósság (durability): Végrehajtás után az eredményt tartós adattárolóra mentjük, így a rendszer esetleges összeomlása után visszaállítható.

Tranzakciófeldolgozó monitor (Transaction Processing Monitor)

Megjegyzés

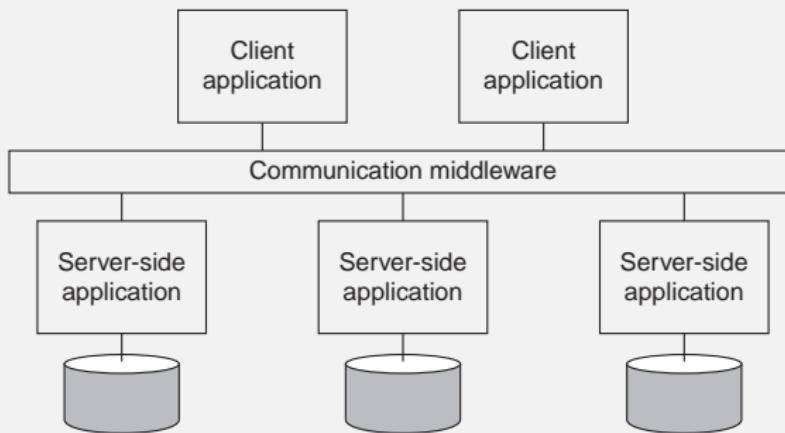
A tranzakciókat sokszor több szerver hajtja végre. Ezeket egy **TP monitor** vezérli.



Alkalmazásintegráció nagy rendszerekben

Probléma

A TP monitor nem választja el az alkalmazásokat az adatbázisoktól.
Továbbá az alkalmazásoknak egymással is kommunikálniuk kell.



- Távoli eljáráshívás (Remote Procedure Call, RPC)
- Üzenetorientált köztesréteg (Message-Oriented Middleware, MOM)

Elosztott átható rendszerek

Átható rendszer

Sok modern elosztott rendszer kicsi, mobil elemekből áll.

Néhány jellemző

- **A környezet megváltozhat:** A rendszernek ezt követnie kell.
- **Ad hoc szerveződés:** A rendszer komponenseit nagyon különböző módokon használhatják a felhasználók. Ezért a rendszernek könnyen konfigurálhatónak kell lennie.
- **Megosztott szolgáltatások:** Mivel a rendszer nagyon változékony, az adatoknak könnyen kell áramlaniuk. Ennek elősegítésére a rendszer elemei általában nagyon egyszerű szerkezetűek.

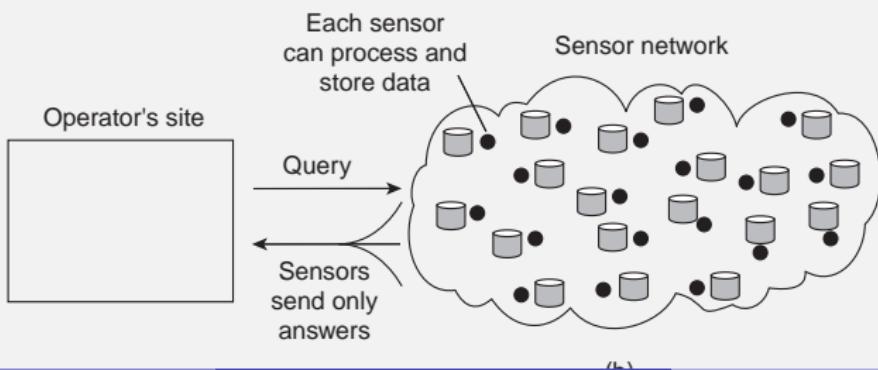
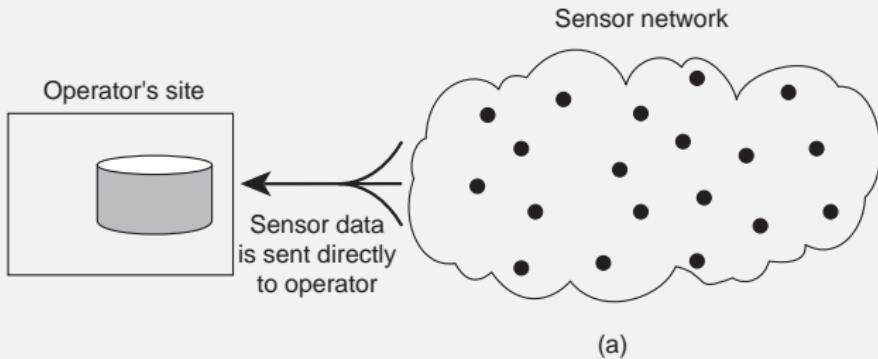
Érzékelőhálózatok

Jellemzők

Az érzékelőket tartalmazó **csúcsok**

- sok van belőlük (nagyságrendileg 10-1000 darab)
- egyszerűek (kevés memória, számítási és kommunikációs kapacitás)
- sokszor elemről működnek, vagy áramforrás sem szükséges hozzájuk

Érzékelőhálózatok mint elosztott rendszerek



Példák

Tömegirányítás

- **Helyzet:** rendezvény fix útvonalakkal (kiállítás, fesztivál)
- **Cél:** az embereket a megfelelő helyre irányítani
 - a hasonló érdeklődésű emberek egy helyre menjenek
 - vész helyzet esetén a fenti csoportokat ugyanahhoz a kijárathoz irányítani
- **Cél:** összetartozó embereket (pl. családokat) egyben tartani



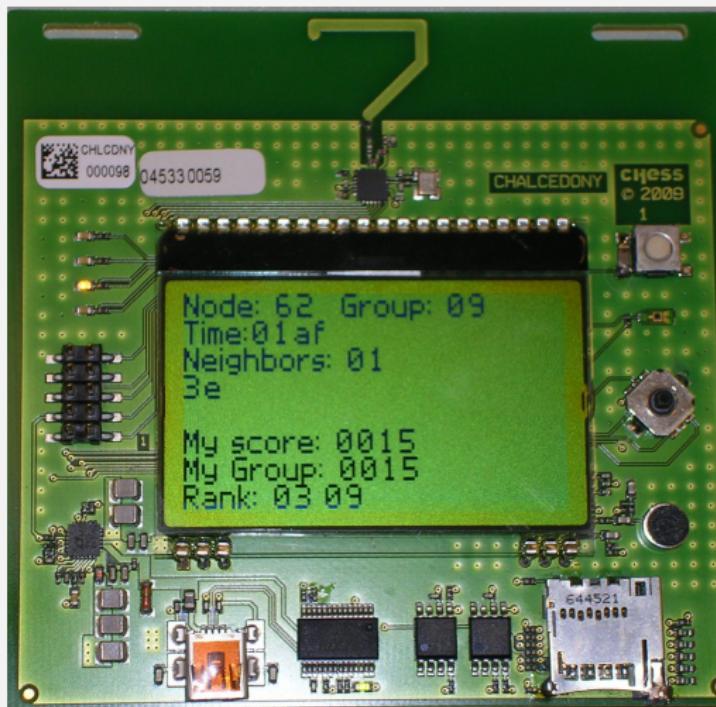
Példák

Szociális játék

- **Helyzet:** Konferencia, a résztvevők különböző csoportokban érkeztek.
- **Cél:** A csoportok vegyítésének elősegítése.
- **Megközelítés:** A rendszer figyeli, hogy a csoportok hogyan viselkednek
 - Ha külön csoportokból származó embereket észlel, bónuszpontokat kapnak az emberek és a csoportok egyaránt.
 - A rendszer kiosztja a csoportszintű pontokat a tagok között.
 - Az eredményeket elektronikus kitűzők mutatják (**feedback** and **social intervention**).

Példa: Szociális játék

A szociális játék egy kitűzője.



Elosztott rendszerek: Alapelvek és paradigmák

Distributed Systems: Principles and Paradigms

Maarten van Steen¹ Kitlei Róbert²

¹VU Amsterdam, Dept. Computer Science

²ELTE Informatikai Kar

2. rész: Architektúrák

2015. május 24.

Tartalomjegyzék

Fejezet
01: Bevezetés
02: Architektúrák
03: Folyamatok
04: Kommunikáció
05: Elnevezési rendszerek
06: Szinkronizáció
07: Konzisztencia & replikáció
08: Hibatűrés
10: Objektumalapú elosztott rendszerek
11: Elosztott fájlrendszerek
12: Elosztott webalapú rendszerek

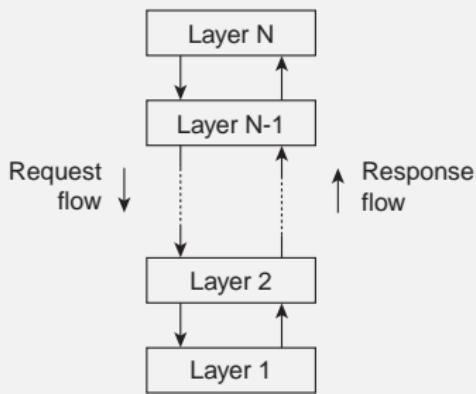
Architektúrák

- Architektúrafajták
- Szoftverarchitektúrák
- Architektúrák és köztesréteg
- Az elosztott rendszerek önszervezése

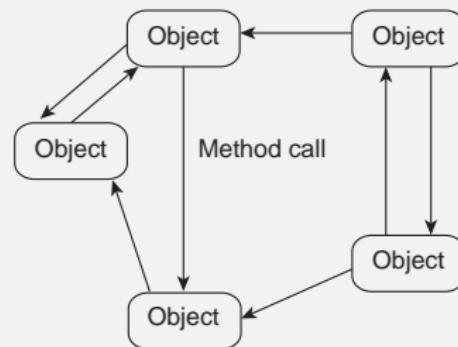
Architektúrafajták

Alapötlet

A rendszer elemeit szervezzük logikai szerepéik szerint különböző komponensekbe, és ezeket osszuk el a rendszer gépein.



(a)



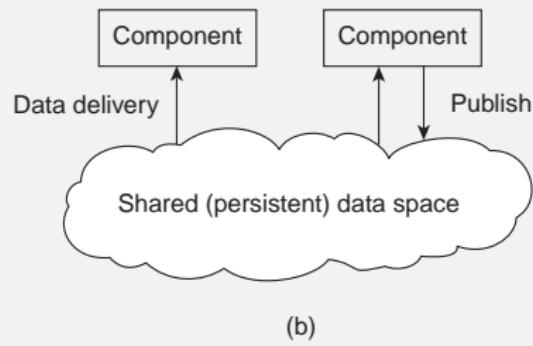
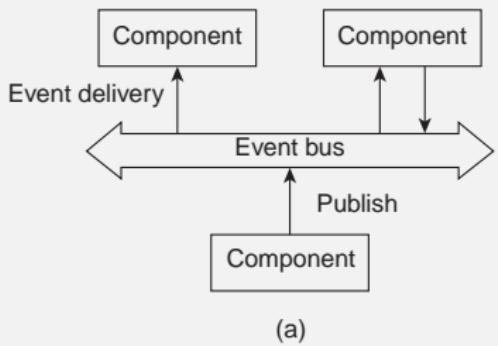
(b)

- (a) A többrétegű megközelítés kliens-szerver rendszerek esetén jól működik
(b) Itt a komponensek (objektumok) összetettebb struktúrában kommunikálnak, mindegyik közvetlenül küld üzeneteket a többieknek.

Architektúrafajták

További architektúrafajták

A komponensek közötti kommunikáció történhet közvetlen kapcsolat nélkül („anonim”), illetve egyidejűség nélkül („aszinkron”).



(a) Publish/subscribe modell (téren független)

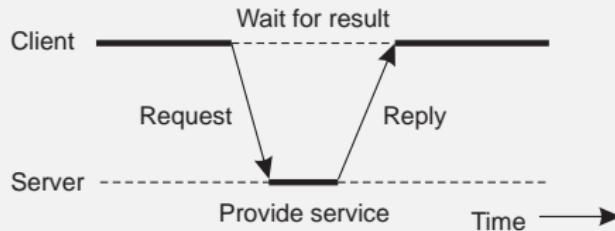
(b) Megosztott, perzisztens adattár (téren és időben független)

Központosított architektúrák

Egyszerű kliens–szerver modell

Jellemzői:

- egyes folyamatok szolgáltatásokat ajánlanak ki (ezek a **szerverek**)
 - más folyamatok ezeket a szolgáltatásokat szeretnék használni (ezek a **kliensek**)
 - a kliensek és a szerverek különböző gépeken lehetnek
 - a kliens kérést küld (amire a szerver válaszol), így veszi igénybe a szolgáltatást



Többrétegű architektúrák

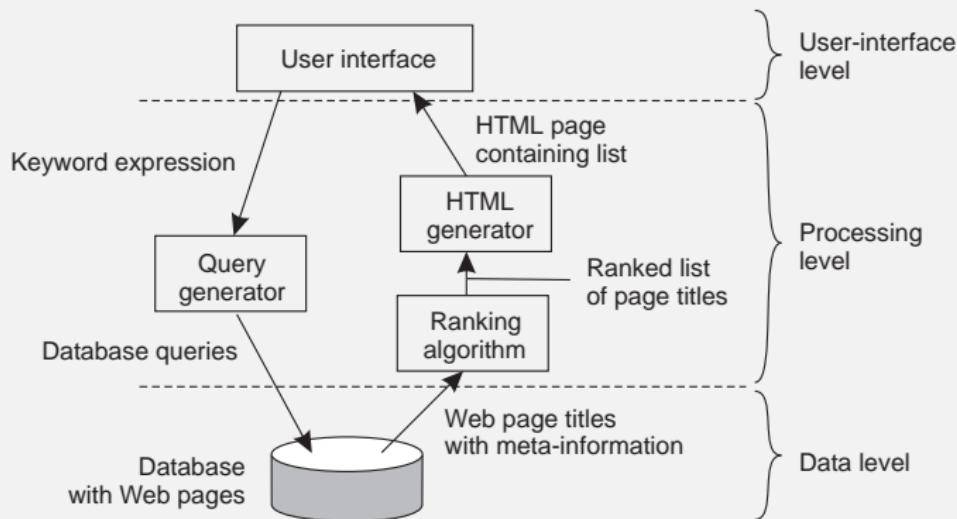
Elosztott információs rendszerek rétegelése

Az elosztott információs rendszerek gyakran három logikai rétegre („layer” vagy „tier”) vannak tagolva.

Háromrétegű architektúra

- **Megjelenítés** (user interface): az alkalmazás felhasználói felületét alkotó komponensekből áll
- **Üzleti logika** (application): az alkalmazás működését írja le (konkrét adatok nélkül)
- **Perzisztencia** (data layer): az adatok tartós tárolása

Többrétegű architektúrák



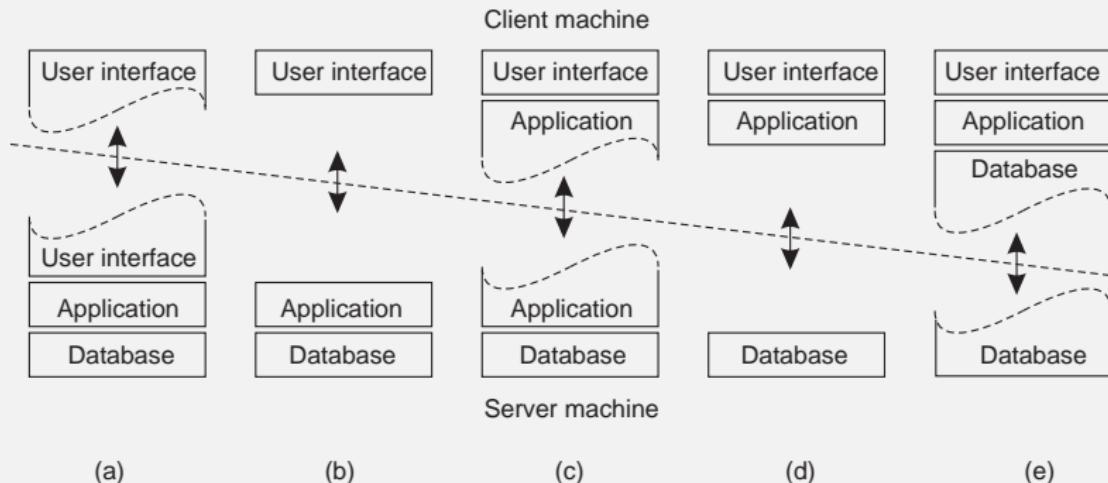
Többrétegű architektúrák

A három rétegből néha több is egy gépen található meg.

Kétrétegű architektúra: kliens/egyszerű szerver

Egyrétegű architektúra: nagygépre (mainframe) kötött terminál

A kétrétegű architektúra többféleképpen bonthatja fel a három réteget:



Decentralizált architektúrák

Peer-to-peer architektúra

Az utóbbi években a **peer-to-peer** (P2P) architektúra egyre népszerűbbé válik. A „peer” szó arra utal, hogy a csúcsok között (többnyire) nincsenek kitüntetett szerepűek.

- **strukturált P2P**: a csúcsok által kiadott gráfszerkezet rögzített
- **strukturálatlan P2P**: a csúcsok szomszédai véletlenszerűek
- **hibrid P2P**: néhány csúcsnak speciális szerepe van, ezek a többitől eltérő szervezésűek

Overlay hálózat

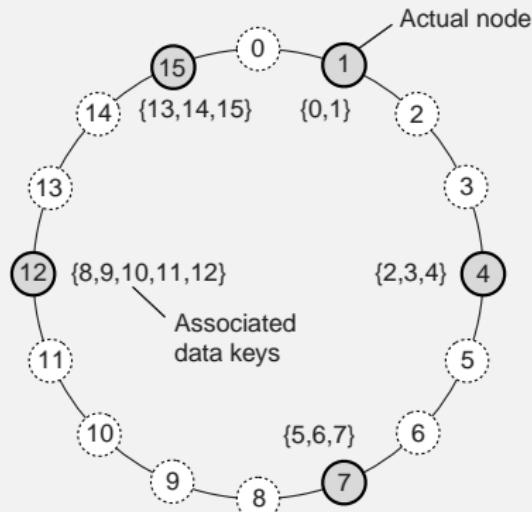
overlay: A gráfban szomszédos csúcsok a fizikai hálózaton lehetnek távol egymástól, a rendszer elfedi, hogy a köztük levő kommunikáció több gépet érintve történik.

- A legtöbb P2P rendszer overlay hálózatra épül.

Strukturált P2P rendszerek

Alapötlet

A csúcsokat valamilyen struktúra szerint overlay hálózatba szervezzük (pl. logikai gyűrű), és a csúcsoktól az azonosítójuk alapján lehet szolgáltatásokat igénybe venni.



Példa: elosztott hasítótábla

Ebben a rendszerben kulcs-érték párokat tárolunk.

Az adott értéket tároló csúcsot hatékonyan meg lehet keresni a kulcsa alapján, akármelyik csúcsra is érkezik be a kérés.

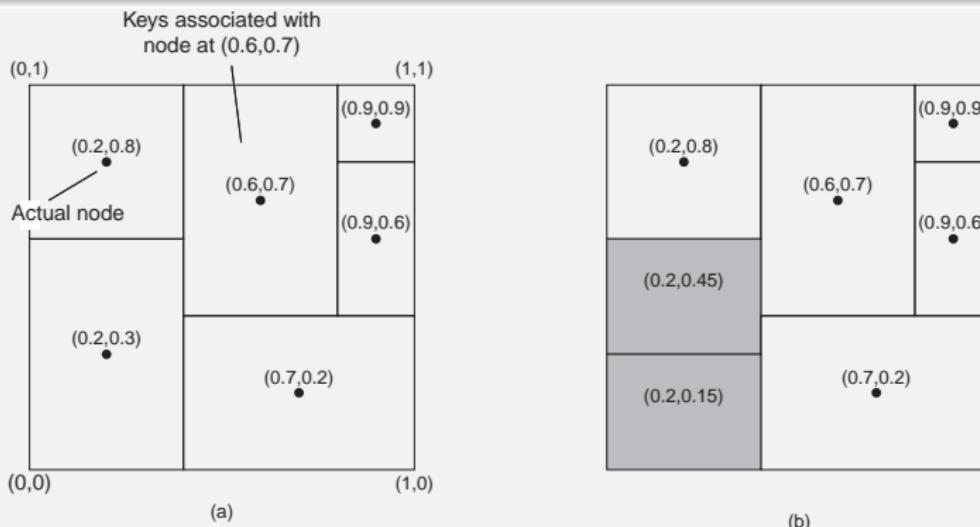
Strukturált P2P rendszerek

Példa: d dimenziós particionált tér

Az adatoknak most d mezője van, kulccsal nem rendelkeznek.

Az így adódó tér fel van osztva annyi tartományra, ahány csúcsunk van; minden csúcs valamelyik tartomány adataiért felelős.

Ha egy új csúcs érkezik, kettébontunk egy tartományt.



Strukturálatlan P2P rendszerek

Strukturálatlan P2P rendszer

A strukturálatlan P2P rendszerek igyekeznek véletlen gráfstruktúrát fenntartani.

- Mindegyik csúcsnak csak részleges nézete van a gráfról (a teljes hálózatnak csak egy kis részét látja).
- minden P csúcs időközönként kiválaszt egy szomszédos Q csúcsot
- P és Q információt cserél, valamint átküldik egymásnak az általuk ismert csúcsokat

Megjegyzés

A rendszer hibatűrését és a gráf véletlenszerűségét nagyban befolyásolja az, hogy a harmadik lépésben pontosan milyen adatok kerülnek át.

Strukturálatlan P2P: pletykálás

Aktív szál

```
selectPeer (&B);  
selectToSend (&bufs);  
sendTo (B, bufs);  
  
receiveFrom (B, &bufr);  
selectToKeep (cache, bufr);
```

Passzív szál

```
receiveFromAny (&A, &bufr);  
selectToSend (&bufs);  
sendTo (A, bufs);  
selectToKeep (cache, bufr);
```

selectPeer: A részleges nézetből kiválaszt egy szomszédot.

selectToSend: Az általa ismert szomszédek közül kiválaszt n darabot.

selectToKeep: (1) A megkapott csúcsokat eltárolja lokálisan.
(2) Eltávolítja a többszörösen szereplő csúcsokat.
(3) A tárolt csúcsok számát m darabra csökkenti. Erre többfajta stratégia lehetséges.

Strukturálatlan P2P: pletykálás

Aktív szál

```
selectPeer (&B) ;  
selectToSend (&bufs) ;  
sendTo (B, bufs) ;  
  
receiveFrom (B, &bufr) ;  
selectToKeep (cache, bufr) ;
```

Passzív szál

```
receiveFromAny (&A, &bufr) ;  
selectToSend (&bufs) ;  
sendTo (A, bufs) ;  
selectToKeep (cache, bufr) ;
```

selectPeer: A részleges nézetből kiválaszt egy szomszédot.

selectToSend: Az általa ismert szomszédek közül kiválaszt *n* darabot.

selectToKeep: (1) A megkapott csúcsokat eltárolja lokálisan.
(2) Eltávolítja a többszörösen szereplő csúcsokat.
(3) A tárolt csúcsok számát *m* darabra csökkenti. Erre többfajta stratégia lehetséges.

Strukturálatlan P2P: pletykálás

Aktív szál

```
selectPeer (&B);  
selectToSend (&bufs);  
sendTo (B, bufs);  
  
receiveFrom (B, &bufr);  
selectToKeep (cache, bufr);
```

Passzív szál

```
receiveFromAny (&A, &bufr);  
selectToSend (&bufs);  
sendTo (A, bufs);  
selectToKeep (cache, bufr);
```

selectPeer: A részleges nézetből kiválaszt egy szomszédot.

selectToSend: Az általa ismert szomszédek közül kiválaszt *n* darabot.

selectToKeep: (1) A megkapott csúcsokat eltárolja lokálisan.
(2) Eltávolítja a többszörösen szereplő csúcsokat.
(3) A tárolt csúcsok számát *m* darabra csökkenti. Erre többfajta stratégia lehetséges.

Strukturálatlan P2P: pletykálás

Aktív szál

```
selectPeer(&B);  
selectToSend(&bufs);  
sendTo(B, bufs);  
  
receiveFrom(B, &bufr);  
selectToKeep(cache, bufr);
```

Passzív szál

```
receiveFromAny(&A, &bufr);  
selectToSend(&bufs);  
sendTo(A, bufs);  
selectToKeep(cache, bufr);
```

selectPeer: A részleges nézetből kiválaszt egy szomszédot.

selectToSend: Az általa ismert szomszédek közül kiválaszt *n* darabot.

selectToKeep: (1) A megkapott csúcsokat eltárolja lokálisan.
(2) Eltávolítja a többszörösen szereplő csúcsokat.
(3) A tárolt csúcsok számát *m* darabra csökkenti. Erre többfajta stratégia lehetséges.

Strukturálatlan P2P: pletykálás

Aktív szál

```
selectPeer(&B);  
selectToSend(&bufs);  
sendTo(B, bufs);  
  
receiveFrom(B, &bufr);  
selectToKeep(cache, bufr);
```

Passzív szál

```
receiveFromAny(&A, &bufr);  
selectToSend(&bufs);  
sendTo(A, bufs);  
selectToKeep(cache, bufr);
```

selectPeer: A részleges nézetből kiválaszt egy szomszédot.

selectToSend: Az általa ismert szomszédek közül kiválaszt *n* darabot.

selectToKeep: (1) A megkapott csúcsokat eltárolja lokálisan.
(2) Eltávolítja a többszörösen szereplő csúcsokat.
(3) A tárolt csúcsok számát *m* darabra csökkenti. Erre többfajta stratégia lehetséges.

Strukturálatlan P2P: pletykálás

Aktív szál

```
selectPeer(&B);  
selectToSend(&bufs);  
sendTo(B, bufs);  
  
receiveFrom(B, &bufr);  
selectToKeep(cache, bufr);
```

Passzív szál

```
receiveFromAny(&A, &bufr);  
selectToSend(&bufs);  
sendTo(A, bufs);  
selectToKeep(cache, bufr);
```

selectPeer: A részleges nézetből kiválaszt egy szomszédot.

selectToSend: Az általa ismert szomszédek közül kiválaszt *n* darabot.

selectToKeep: (1) A megkapott csúcsokat eltárolja lokálisan.
(2) Eltávolítja a többszörösen szereplő csúcsokat.
(3) A tárolt csúcsok számát *m* darabra csökkenti. Erre többfajta stratégia lehetséges.

Strukturálatlan P2P: pletykálás

Aktív szál

```
selectPeer(&B);  
selectToSend(&bufs);  
sendTo(B, bufs);  
  
receiveFrom(B, &bufr);  
selectToKeep(cache, bufr);
```

Passzív szál

```
receiveFromAny(&A, &bufr);  
selectToSend(&bufs);  
sendTo(A, bufs);  
selectToKeep(cache, bufr);
```

selectPeer: A részleges nézetből kiválaszt egy szomszédot.

selectToSend: Az általa ismert szomszédek közül kiválaszt *n* darabot.

selectToKeep: (1) A megkapott csúcsokat eltárolja lokálisan.
(2) Eltávolítja a többszörösen szereplő csúcsokat.
(3) A tárolt csúcsok számát *m* darabra csökkenti. Erre többfajta stratégia lehetséges.

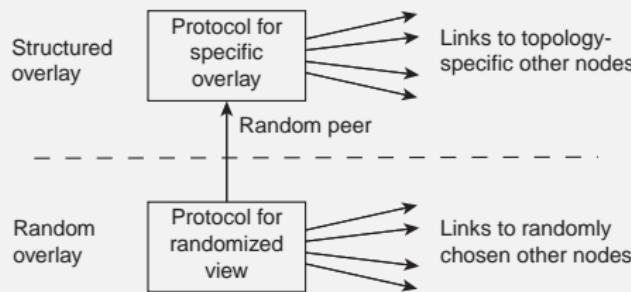
Overlay hálózatok topológiájának kezelése

Alapötlet

Különböztessünk meg két réteget:

- (1) az alsó rétegben a csúcsoknak csak részleges nézete van;
- (2) a felső rétegbe csak kevés csúcs kerülhet.

Az alsó réteg véletlenszerű csúcsokat **ad át** a felső rétegnek; a felső réteg ezek közül csak keveset tart meg.

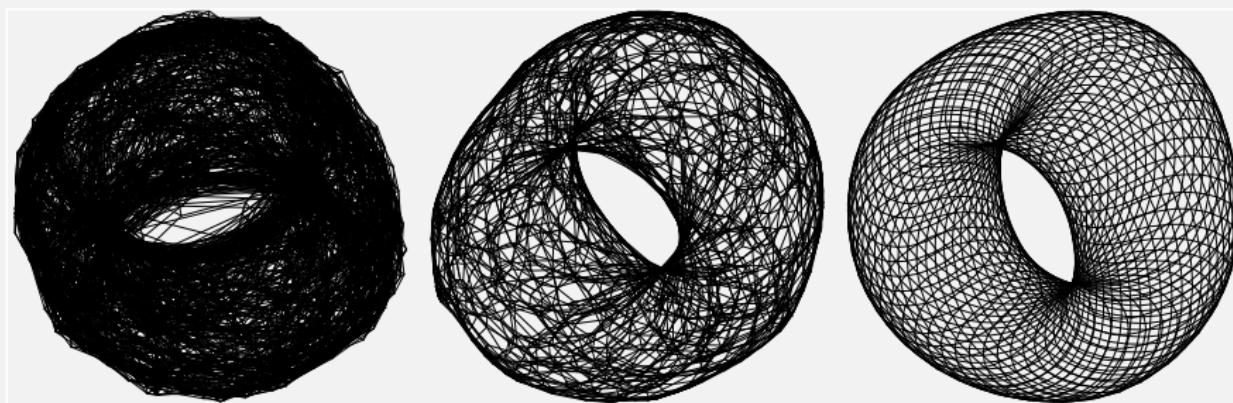


Overlay topológia: példa: tórusz

Tórusz overlay topológia kialakítása

Ha megfelelően választjuk meg, milyen csúcsokat tartson meg a felső réteg, akkor a kezdetben véletlenszerű overlay kapcsolatok hamarosan szabályos alakba rendeződnek.

Itt egy távolságfüggvény szerinti megtartó szabály hat (az overlay a közelieket veszi át), és már az első néhány lépés után jól látszik a kijövő tórusz-alakzat.



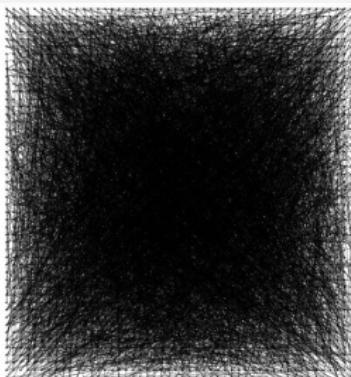
Time

Overlay topológia: példa: clusterezés

Most minden csúcsnak hozzárendelünk egy $GID(i) \in \mathbb{N}$ számot, és azt mondjuk, hogy i a $GID(i)$ csoportba tartozik. Szintén távolságfüggvényt használunk:

$$dist(i, j) = \begin{cases} 1 & \text{ha } GID(i) = GID(j) \\ 0 & \text{ha } GID(i) \neq GID(j) \end{cases}$$

Itt is igen gyorsan kialakul a kívánt szerkezet: csak az azonos csoportbeli csúcsok között lesz kapcsolat, kialakulnak a **clusterek**.

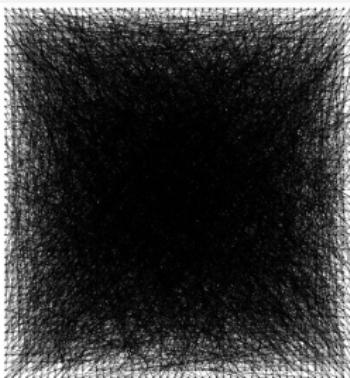


Overlay topológia: példa: clusterezés

Most minden csúcsnak hozzárendelünk egy $GID(i) \in \mathbb{N}$ számot, és azt mondjuk, hogy i a $GID(i)$ csoportba tartozik. Szintén távolságfüggvényt használunk:

$$dist(i, j) = \begin{cases} 1 & \text{ha } GID(i) = GID(j) \\ 0 & \text{ha } GID(i) \neq GID(j) \end{cases}$$

Itt is igen gyorsan kialakul a kívánt szerkezet: csak az azonos csoportbeli csúcsok között lesz kapcsolat, kialakulnak a **clusterek**.

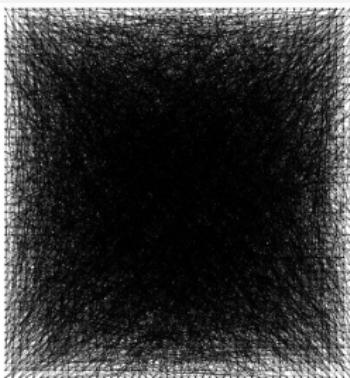


Overlay topológia: példa: clusterezés

Most minden csúcsnak hozzárendelünk egy $GID(i) \in \mathbb{N}$ számot, és azt mondjuk, hogy i a $GID(i)$ csoportba tartozik. Szintén távolságfüggvényt használunk:

$$dist(i, j) = \begin{cases} 1 & \text{ha } GID(i) = GID(j) \\ 0 & \text{ha } GID(i) \neq GID(j) \end{cases}$$

Itt is igen gyorsan kialakul a kívánt szerkezet: csak az azonos csoportbeli csúcsok között lesz kapcsolat, kialakulnak a **clusterek**.

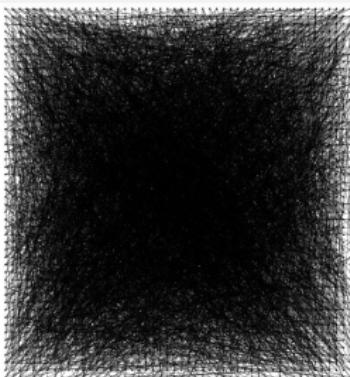


Overlay topológia: példa: clusterezés

Most minden csúcsnak hozzárendelünk egy $GID(i) \in \mathbb{N}$ számot, és azt mondjuk, hogy i a $GID(i)$ csoportba tartozik. Szintén távolságfüggvényt használunk:

$$dist(i, j) = \begin{cases} 1 & \text{ha } GID(i) = GID(j) \\ 0 & \text{ha } GID(i) \neq GID(j) \end{cases}$$

Itt is igen gyorsan kialakul a kívánt szerkezet: csak az azonos csoportbeli csúcsok között lesz kapcsolat, kialakulnak a **clusterek**.

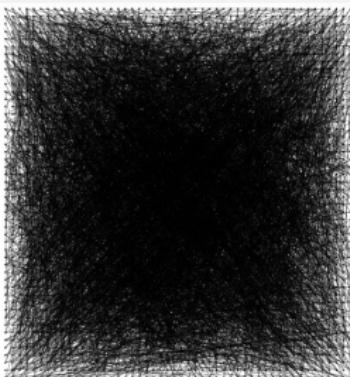


Overlay topológia: példa: clusterezés

Most minden csúcsnak hozzárendelünk egy $GID(i) \in \mathbb{N}$ számot, és azt mondjuk, hogy i a $GID(i)$ csoportba tartozik. Szintén távolságfüggvényt használunk:

$$dist(i, j) = \begin{cases} 1 & \text{ha } GID(i) = GID(j) \\ 0 & \text{ha } GID(i) \neq GID(j) \end{cases}$$

Itt is igen gyorsan kialakul a kívánt szerkezet: csak az azonos csoportbeli csúcsok között lesz kapcsolat, kialakulnak a **clusterek**.

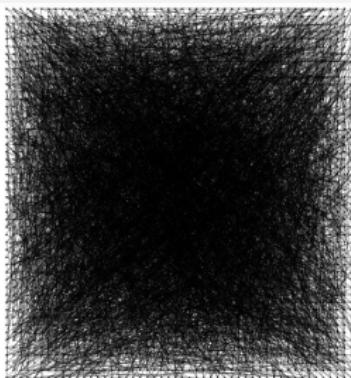


Overlay topológia: példa: clusterezés

Most minden csúcsnak hozzárendelünk egy $GID(i) \in \mathbb{N}$ számot, és azt mondjuk, hogy i a $GID(i)$ csoportba tartozik. Szintén távolságfüggvényt használunk:

$$dist(i, j) = \begin{cases} 1 & \text{ha } GID(i) = GID(j) \\ 0 & \text{ha } GID(i) \neq GID(j) \end{cases}$$

Itt is igen gyorsan kialakul a kívánt szerkezet: csak az azonos csoportbeli csúcsok között lesz kapcsolat, kialakulnak a **clusterek**.

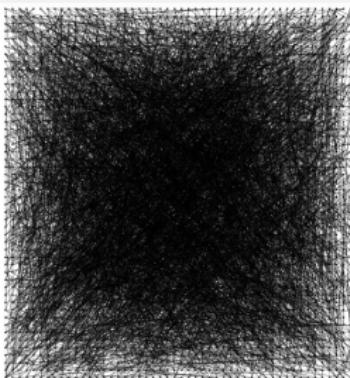


Overlay topológia: példa: clusterezés

Most minden csúcsnak hozzárendelünk egy $GID(i) \in \mathbb{N}$ számot, és azt mondjuk, hogy i a $GID(i)$ csoportba tartozik. Szintén távolságfüggvényt használunk:

$$dist(i, j) = \begin{cases} 1 & \text{ha } GID(i) = GID(j) \\ 0 & \text{ha } GID(i) \neq GID(j) \end{cases}$$

Itt is igen gyorsan kialakul a kívánt szerkezet: csak az azonos csoportbeli csúcsok között lesz kapcsolat, kialakulnak a **clusterek**.

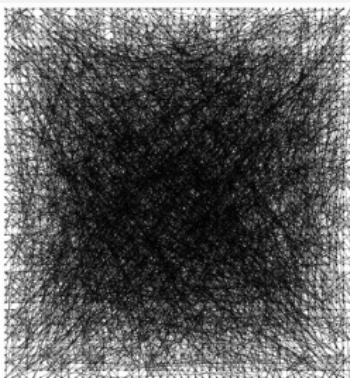


Overlay topológia: példa: clusterezés

Most minden csúcsnak hozzárendelünk egy $GID(i) \in \mathbb{N}$ számot, és azt mondjuk, hogy i a $GID(i)$ csoportba tartozik. Szintén távolságfüggvényt használunk:

$$dist(i, j) = \begin{cases} 1 & \text{ha } GID(i) = GID(j) \\ 0 & \text{ha } GID(i) \neq GID(j) \end{cases}$$

Itt is igen gyorsan kialakul a kívánt szerkezet: csak az azonos csoportbeli csúcsok között lesz kapcsolat, kialakulnak a **clusterek**.

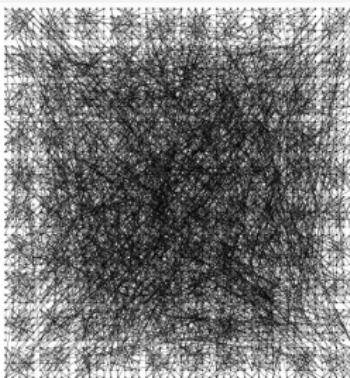


Overlay topológia: példa: clusterezés

Most minden csúcsnak hozzárendelünk egy $GID(i) \in \mathbb{N}$ számot, és azt mondjuk, hogy i a $GID(i)$ csoportba tartozik. Szintén távolságfüggvényt használunk:

$$dist(i, j) = \begin{cases} 1 & \text{ha } GID(i) = GID(j) \\ 0 & \text{ha } GID(i) \neq GID(j) \end{cases}$$

Itt is igen gyorsan kialakul a kívánt szerkezet: csak az azonos csoportbeli csúcsok között lesz kapcsolat, kialakulnak a **clusterek**.

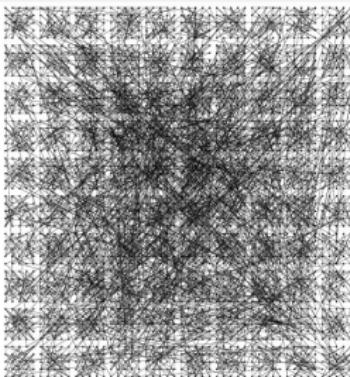


Overlay topológia: példa: clusterezés

Most minden csúcsnak hozzárendelünk egy $GID(i) \in \mathbb{N}$ számot, és azt mondjuk, hogy i a $GID(i)$ csoportba tartozik. Szintén távolságfüggvényt használunk:

$$dist(i, j) = \begin{cases} 1 & \text{ha } GID(i) = GID(j) \\ 0 & \text{ha } GID(i) \neq GID(j) \end{cases}$$

Itt is igen gyorsan kialakul a kívánt szerkezet: csak az azonos csoportbeli csúcsok között lesz kapcsolat, kialakulnak a **clusterek**.

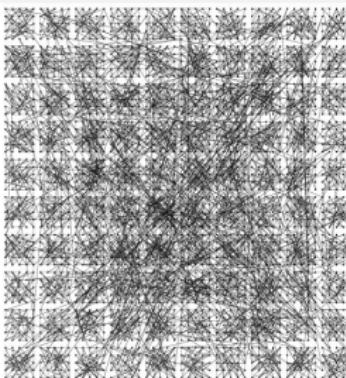


Overlay topológia: példa: clusterezés

Most minden csúcsnak hozzárendelünk egy $GID(i) \in \mathbb{N}$ számot, és azt mondjuk, hogy i a $GID(i)$ csoportba tartozik. Szintén távolságfüggvényt használunk:

$$dist(i, j) = \begin{cases} 1 & \text{ha } GID(i) = GID(j) \\ 0 & \text{ha } GID(i) \neq GID(j) \end{cases}$$

Itt is igen gyorsan kialakul a kívánt szerkezet: csak az azonos csoportbeli csúcsok között lesz kapcsolat, kialakulnak a **clusterek**.

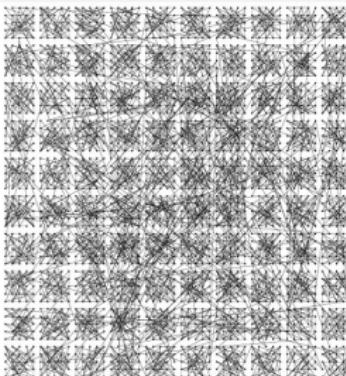


Overlay topológia: példa: clusterezés

Most minden csúcsnak hozzárendelünk egy $GID(i) \in \mathbb{N}$ számot, és azt mondjuk, hogy i a $GID(i)$ csoportba tartozik. Szintén távolságfüggvényt használunk:

$$dist(i, j) = \begin{cases} 1 & \text{ha } GID(i) = GID(j) \\ 0 & \text{ha } GID(i) \neq GID(j) \end{cases}$$

Itt is igen gyorsan kialakul a kívánt szerkezet: csak az azonos csoportbeli csúcsok között lesz kapcsolat, kialakulnak a **clusterek**.

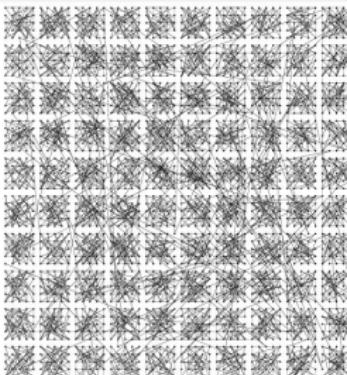


Overlay topológia: példa: clusterezés

Most minden csúcsnak hozzárendelünk egy $GID(i) \in \mathbb{N}$ számot, és azt mondjuk, hogy i a $GID(i)$ csoportba tartozik. Szintén távolságfüggvényt használunk:

$$dist(i, j) = \begin{cases} 1 & \text{ha } GID(i) = GID(j) \\ 0 & \text{ha } GID(i) \neq GID(j) \end{cases}$$

Itt is igen gyorsan kialakul a kívánt szerkezet: csak az azonos csoportbeli csúcsok között lesz kapcsolat, kialakulnak a **clusterek**.

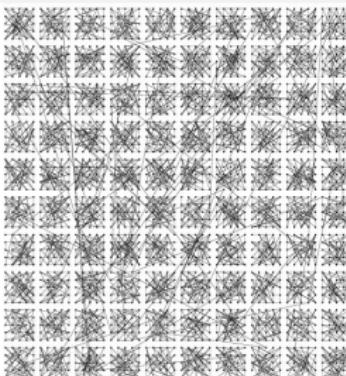


Overlay topológia: példa: clusterezés

Most minden csúcsnak hozzárendelünk egy $GID(i) \in \mathbb{N}$ számot, és azt mondjuk, hogy i a $GID(i)$ csoportba tartozik. Szintén távolságfüggvényt használunk:

$$dist(i, j) = \begin{cases} 1 & \text{ha } GID(i) = GID(j) \\ 0 & \text{ha } GID(i) \neq GID(j) \end{cases}$$

Itt is igen gyorsan kialakul a kívánt szerkezet: csak az azonos csoportbeli csúcsok között lesz kapcsolat, kialakulnak a **clusterek**.

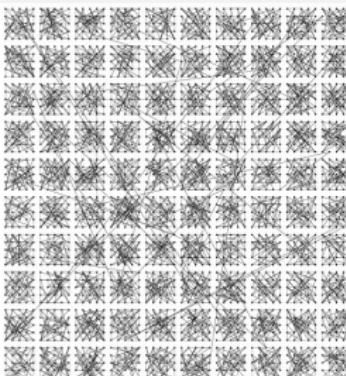


Overlay topológia: példa: clusterezés

Most minden csúcsnak hozzárendelünk egy $GID(i) \in \mathbb{N}$ számot, és azt mondjuk, hogy i a $GID(i)$ csoportba tartozik. Szintén távolságfüggvényt használunk:

$$dist(i, j) = \begin{cases} 1 & \text{ha } GID(i) = GID(j) \\ 0 & \text{ha } GID(i) \neq GID(j) \end{cases}$$

Itt is igen gyorsan kialakul a kívánt szerkezet: csak az azonos csoportbeli csúcsok között lesz kapcsolat, kialakulnak a **clusterek**.

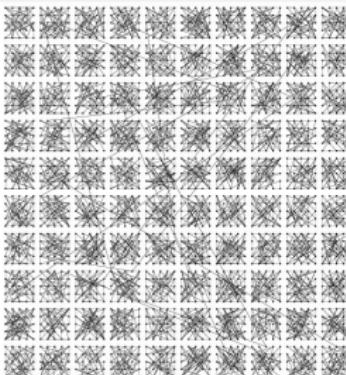


Overlay topológia: példa: clusterezés

Most minden csúcsnak hozzárendelünk egy $GID(i) \in \mathbb{N}$ számot, és azt mondjuk, hogy i a $GID(i)$ csoportba tartozik. Szintén távolságfüggvényt használunk:

$$dist(i, j) = \begin{cases} 1 & \text{ha } GID(i) = GID(j) \\ 0 & \text{ha } GID(i) \neq GID(j) \end{cases}$$

Itt is igen gyorsan kialakul a kívánt szerkezet: csak az azonos csoportbeli csúcsok között lesz kapcsolat, kialakulnak a **clusterek**.

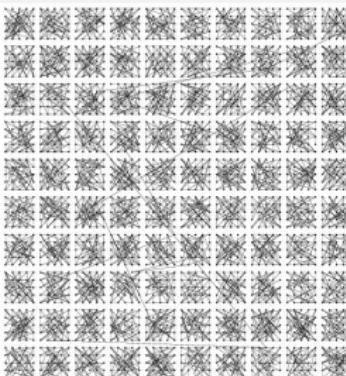


Overlay topológia: példa: clusterezés

Most minden csúcsnak hozzárendelünk egy $GID(i) \in \mathbb{N}$ számot, és azt mondjuk, hogy i a $GID(i)$ csoportba tartozik. Szintén távolságfüggvényt használunk:

$$dist(i, j) = \begin{cases} 1 & \text{ha } GID(i) = GID(j) \\ 0 & \text{ha } GID(i) \neq GID(j) \end{cases}$$

Itt is igen gyorsan kialakul a kívánt szerkezet: csak az azonos csoportbeli csúcsok között lesz kapcsolat, kialakulnak a **clusterek**.

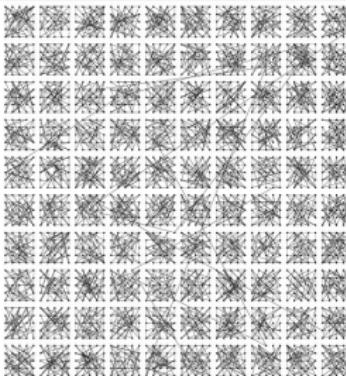


Overlay topológia: példa: clusterezés

Most minden csúcsnak hozzárendelünk egy $GID(i) \in \mathbb{N}$ számot, és azt mondjuk, hogy i a $GID(i)$ csoportba tartozik. Szintén távolságfüggvényt használunk:

$$dist(i, j) = \begin{cases} 1 & \text{ha } GID(i) = GID(j) \\ 0 & \text{ha } GID(i) \neq GID(j) \end{cases}$$

Itt is igen gyorsan kialakul a kívánt szerkezet: csak az azonos csoportbeli csúcsok között lesz kapcsolat, kialakulnak a **clusterek**.

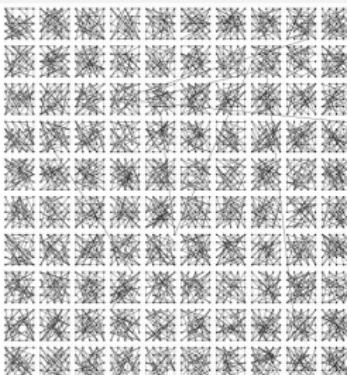


Overlay topológia: példa: clusterezés

Most minden csúcsnak hozzárendelünk egy $GID(i) \in \mathbb{N}$ számot, és azt mondjuk, hogy i a $GID(i)$ csoportba tartozik. Szintén távolságfüggvényt használunk:

$$dist(i, j) = \begin{cases} 1 & \text{ha } GID(i) = GID(j) \\ 0 & \text{ha } GID(i) \neq GID(j) \end{cases}$$

Itt is igen gyorsan kialakul a kívánt szerkezet: csak az azonos csoportbeli csúcsok között lesz kapcsolat, kialakulnak a **clusterek**.

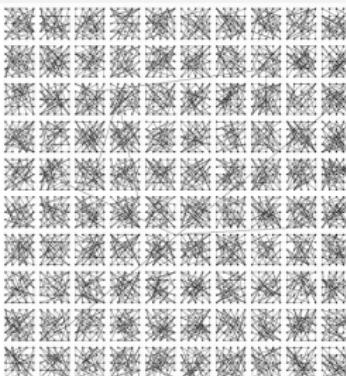


Overlay topológia: példa: clusterezés

Most minden csúcsnak hozzárendelünk egy $GID(i) \in \mathbb{N}$ számot, és azt mondjuk, hogy i a $GID(i)$ csoportba tartozik. Szintén távolságfüggvényt használunk:

$$dist(i, j) = \begin{cases} 1 & \text{ha } GID(i) = GID(j) \\ 0 & \text{ha } GID(i) \neq GID(j) \end{cases}$$

Itt is igen gyorsan kialakul a kívánt szerkezet: csak az azonos csoportbeli csúcsok között lesz kapcsolat, kialakulnak a **clusterek**.

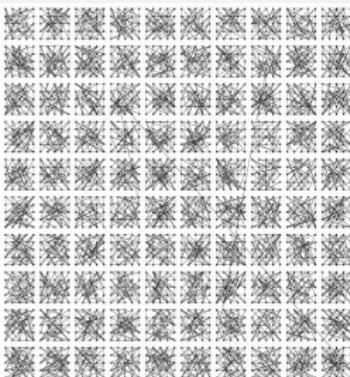


Overlay topológia: példa: clusterezés

Most minden csúcsnak hozzárendelünk egy $GID(i) \in \mathbb{N}$ számot, és azt mondjuk, hogy i a $GID(i)$ csoportba tartozik. Szintén távolságfüggvényt használunk:

$$dist(i, j) = \begin{cases} 1 & \text{ha } GID(i) = GID(j) \\ 0 & \text{ha } GID(i) \neq GID(j) \end{cases}$$

Itt is igen gyorsan kialakul a kívánt szerkezet: csak az azonos csoportbeli csúcsok között lesz kapcsolat, kialakulnak a **clusterek**.

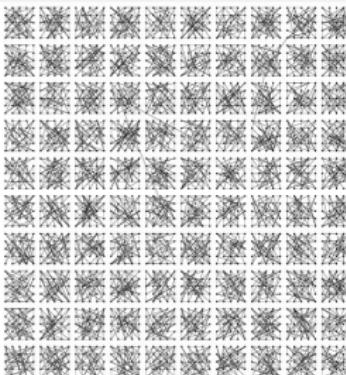


Overlay topológia: példa: clusterezés

Most minden csúcsnak hozzárendelünk egy $GID(i) \in \mathbb{N}$ számot, és azt mondjuk, hogy i a $GID(i)$ csoportba tartozik. Szintén távolságfüggvényt használunk:

$$dist(i, j) = \begin{cases} 1 & \text{ha } GID(i) = GID(j) \\ 0 & \text{ha } GID(i) \neq GID(j) \end{cases}$$

Itt is igen gyorsan kialakul a kívánt szerkezet: csak az azonos csoportbeli csúcsok között lesz kapcsolat, kialakulnak a **clusterek**.

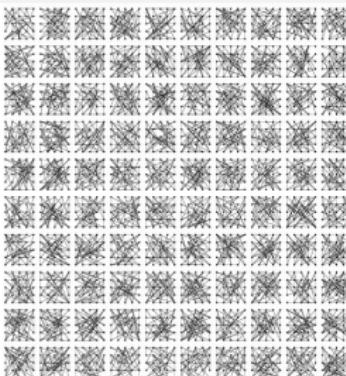


Overlay topológia: példa: clusterezés

Most minden csúcsnak hozzárendelünk egy $GID(i) \in \mathbb{N}$ számot, és azt mondjuk, hogy i a $GID(i)$ csoportba tartozik. Szintén távolságfüggvényt használunk:

$$dist(i, j) = \begin{cases} 1 & \text{ha } GID(i) = GID(j) \\ 0 & \text{ha } GID(i) \neq GID(j) \end{cases}$$

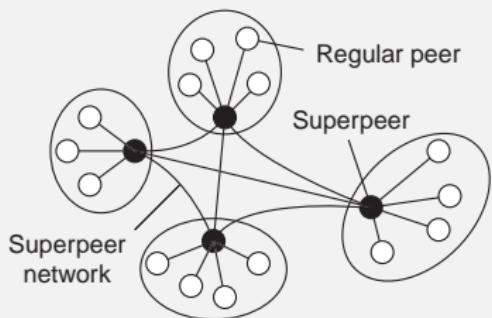
Itt is igen gyorsan kialakul a kívánt szerkezet: csak az azonos csoportbeli csúcsok között lesz kapcsolat, kialakulnak a **clusterek**.



Superpeer csúcsok

Superpeer

superpeer: olyan kisszámú csúcs, amelyeknek külön feladata van



Néhány jellemző feladat

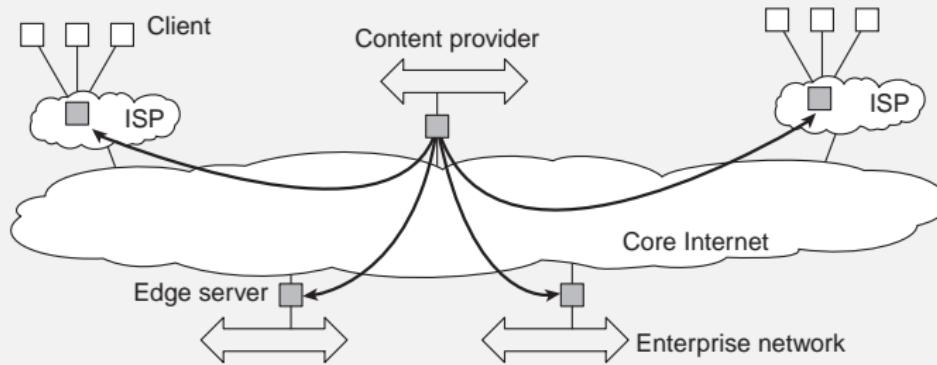
- kereséshez index fenntartása
- a hálózat állapotának felügyelete
- csúcsok közötti kapcsolatok létrehozása

Hibrid arch.: kliens-szerver + P2P: edge szerver

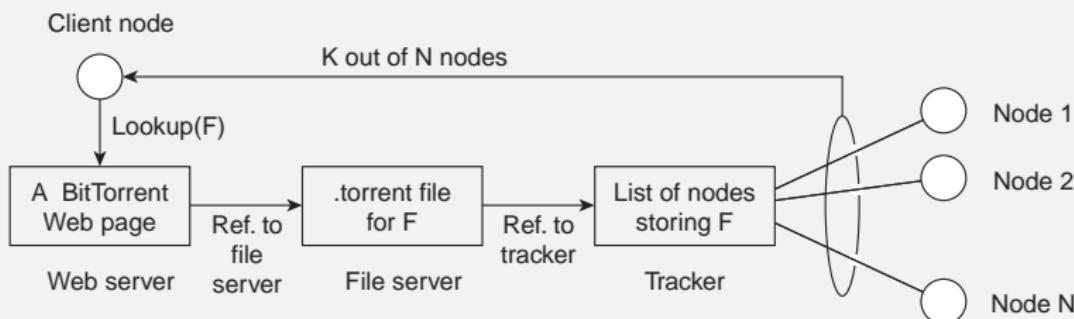
Példa

edge szerver: az adatokat tároló szerver, a kliensekhez minél közelebb van elhelyezve, jellemzően ott, ahol egy nagyobb hálózat az Internetre csatlakozik

Content Delivery Network (CDN) rendszerekben jellemző, a tartalomszolgáltatás hatékonyságát növelik és költségét csökkentik.



Hibrid arch.: kliens-szerver + P2P: BitTorrent



Alapötlet

Miután a csúcs kiderítette, melyik másik csúcsok tartalmaznak részeket a kívánt fájlból, azokat **párhuzamosan** tölti le, és egyúttal önmaga is kiajánlja megosztásra.

Architektúrák és köztesréteg

Probléma

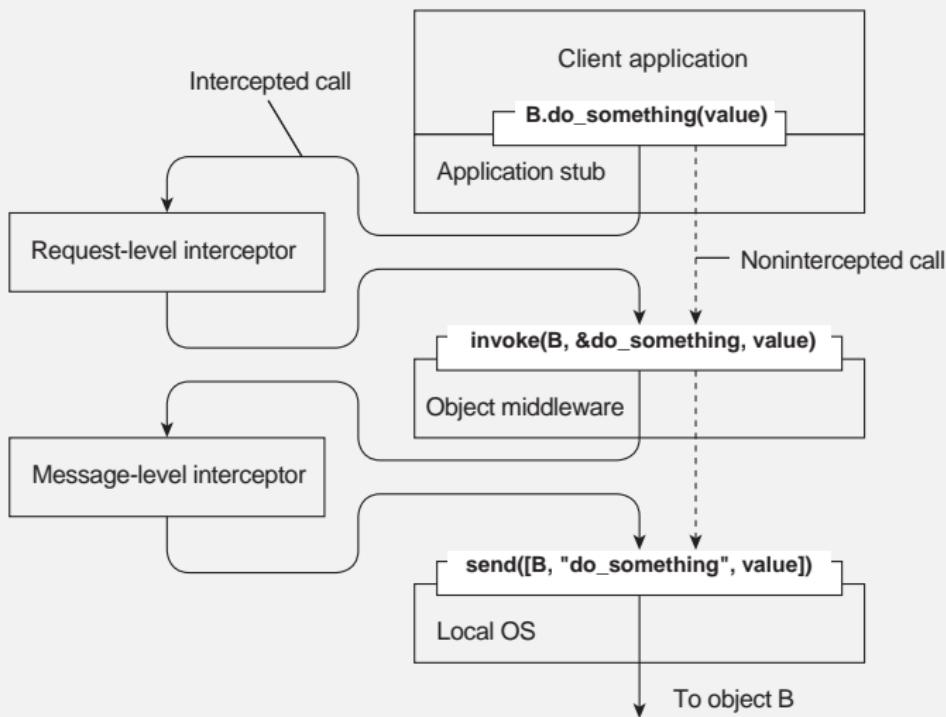
Előfordulhat, hogy az elosztott rendszer/alkalmazás szerkezete nem felel meg a megváltozott igényeknek.

Ilyenkor legtöbbször nem kell újraírni a teljes rendszert: elegendő lehet (dinamikusan) **adaptálni a köztesréteg viselkedését**.

Interceptor

interceptor: Távoli objektum elérése során a vezérlés szokásos menetébe avatkozik bele, pl. átalakíthatja más formátumra a kérést. Jellemzően az architektúra rétegei közé illeszthető.

Interceptors



Adaptív middleware

Funkciók szétválasztása (separation of concerns): A szoftver különböző jellegű funkciói váljanak minél jobban külön, így azokat könnyebb egymástól függetlenül módosítani.

Önvizsgálat (reflection): A program legyen képes feltárni a saját szerkezetét, és futás közben módosítani azt.

Komponensalapú szervezés: Az elosztott alkalmazás legyen moduláris, a komponensei legyenek könnyen cserélhetőek. A komponensek közötti függések legyenek egyértelműek, és csak annyi legyen belőlük, amennyi feltétlenül szükséges.

Önszervező elosztott rendszerek

Adaptív rendszer képességei

Az egyes szoftverelemek adaptivitása kihat a rendszerre, ezért megvizsgáljuk, hogyan lehet adaptív rendszereket készíteni.

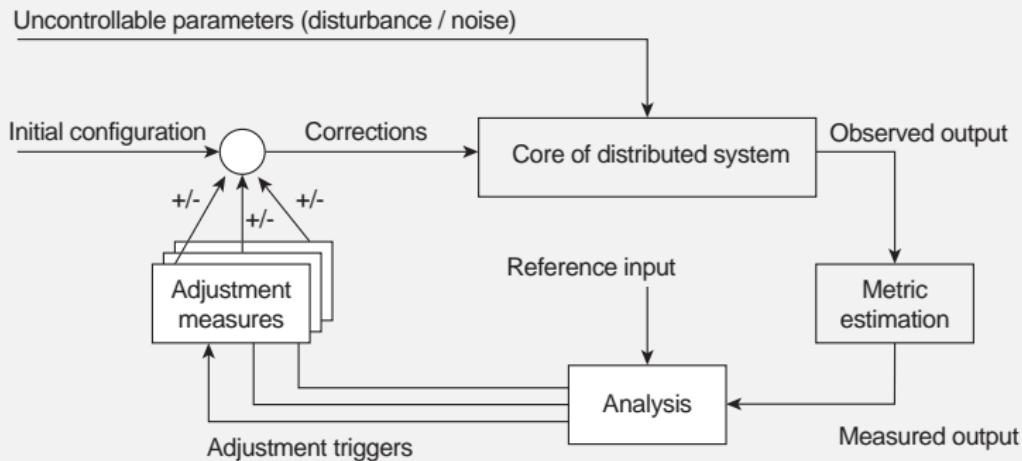
Különféle elvárásaink lehetnek:

- Önkonfiguráció
- Önkezelő
- Öngyógyító
- Ön optimalizáló
- Ön*

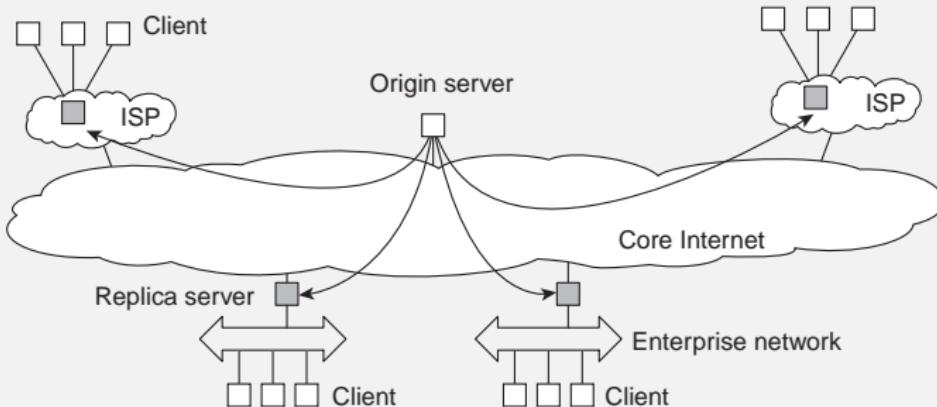
Adaptivitás visszacsatolással

Visszacsatolásos modell

Az ön* rendszerek sokszor az alábbi jellegű **visszacsatolásos vezérléssel** rendelkeznek: mérik, hogy a rendszer mennyire tér el a kívánt tulajdonságoktól, és szükség szerint változtatnak a beállításokon.



Példa: Globule



- Kollaboratív webes CDN, a tartalmakat költségmodell alapján helyezi el (minden szemponra: fontosság \times költség).
- A központi szerver (origin server) elemzi, ami történt, és az alapján állítja be a fontossági paramétereket, hogy mi történt volna, ha P oldalt az S edge szerver tárolta volna.
- A számításokat különböző stratégiákra végzi el, végül a legjobbat választja ki.

Elosztott rendszerek: Alapelvek és paradigmák Distributed Systems: Principles and Paradigms

Maarten van Steen¹ Kitlei Róbert²

¹VU Amsterdam, Dept. Computer Science

²ELTE Informatikai Kar

3. rész: Folyamatok

2015. május 24.

Tartalomjegyzék

Fejezet
01: Bevezetés
02: Architektúrák
03: Folyamatok
04: Kommunikáció
05: Elnevezési rendszerek
06: Szinkronizáció
07: Konzisztencia & replikáció
08: Hibatűrés
10: Objektumalapú elosztott rendszerek
11: Elosztott fájlrendszerek
12: Elosztott webalapú rendszerek

Szálak: bevezetés

Alapötlet

A legtöbb hardvereszköznek létezik szoftveres megfelelője.

Processzor (CPU): Hardvereszköz, utasításokat képes sorban végrehajtani, amelyek egy megadott utasításkészletből származnak.

Szál (thread): A processzor egyfajta szoftveres megfelelője, minimális kontextussal (környezettel). Ha a szálat megállítjuk, a kontextus elmenthető és továbbfuttatáshoz visszatölthető.

Folyamat (process, task): Egy vagy több szálat összefogó nagyobb egység. Egy folyamat szálai közös memóriaterületen (címtartományon) dolgoznak, azonban különböző folyamatok nem látják egymás memóriaterületét.

Hasonló elnevezések

Fontos: nem összekeverendő!

stream = folyam \neq folyamat = processz \neq processzor

Kontextusváltás

Kontextusváltás

- **kontextusváltás:** A másik folyamatnak/szálnak történő vezérlésátadás, illetve a megfelelő kontextusok cseréje. Így egy processzor több folyamatot/szálat is végre tud hajtani.
- **Processzor kontextusa:** Az utasítások végrehajtásában szerepet játszó kisszámú regiszter (elemi értéktároló) tartalma.
- **Szál kontextusa:** Jellemzően nem sokkal bővebb a processzorkontextusnál. A szálak közötti váltáshoz nem kell igénybe venni az operációs rendszer szolgáltatásait.
- **Folyamat kontextusa:** Ahhoz, hogy a régi és az új folyamat memóriaterülete elkülönüljön, a memóriavezérlő (memory management unit, MMU) tartalmának jórészét át kell írni, amire csak a kernel szintnek van joga.
A folyamatok létrehozása, törlése és a kontextusváltás köztük sokkal költségesebb a szálakénál.

Szálak és operációs rendszerek

Hol legyenek a szálak?

A szálakat kezelheti az operációs rendszer, vagy tőle független szálkönyvtárak. Mindkét megközelítésnek vannak előnyei és hátrányai.

Szálak folyamaton belül: szálkönyvtárak

- **előny:** minden műveletet **egyetlen folyamaton belül** kezelünk, ez hatékony.
- **hátrány:** Az operációs rendszer számára a szál **minden művelete a gazdafolyamatról érkezik** ⇒ ha a kernel blokkolja a szálat (pl. lemezművelet során), a folyamat is blokkolódik.
- **hátrány:** Ha a kernel nem látja a szálakat közvetlenül, hogyan tud szignálokat közvetíteni nekik?

Szálak és operációs rendszerek

Szálak folyamaton kívül: kernelszintű szálak

A szálkönyvtárak helyezhetők kernelszintre is. Ekkor *minden* szálművelet rendszerhíváson keresztül érhető el.

- előny: A szálak blokkolása nem okoz problémát: **a kernel be tudja ütemezni a gazdafolyamat egy másik szálát.**
- előny: A szignálokat a kernel a megfelelő szálhoz tudja irányítani.
- hátrány: Mivel minden művelet a kernelt érinti, ez a **hatékonyság rovására megy.**

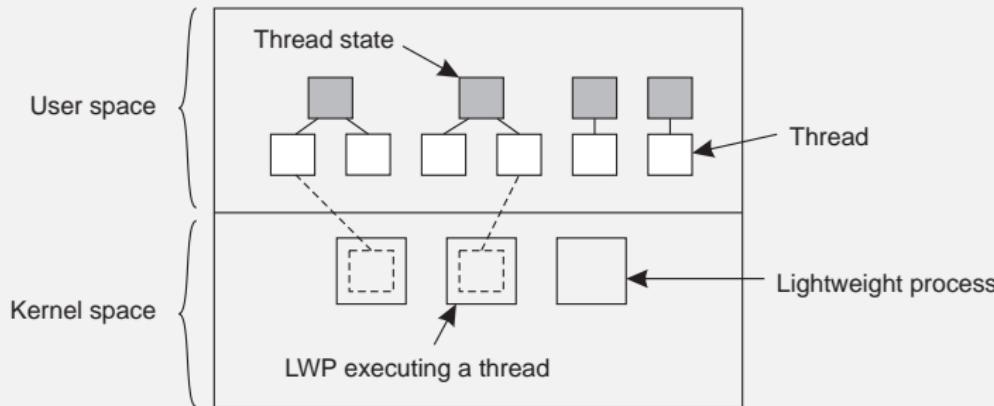
Köztes megoldás?

Lehet-e olyan megoldást találni, ami ötvözi a fenti két megközelítés előnyeit?

Solaris szálak

Könnyűsúlyú folyamatok

könnyűsúlyú folyamat (lightweight process, LWP): Kernelszintű szálak, amelyek felhasználói szintű szálkezelőket futtatnak.



Támogatottsága

A legtöbb rendszer az előző két megközelítés valamelyikét támogatja.

Szálak a kliensoldalon

Példa: többszálú webkliens

A hálózat késleltetésének elfedése:

- A böngésző letöltött egy oldalt, ami több másik tartalomra hivatkozik.
- **Mindegyik tartalmat külön szalon tölti le**, amíg a HTTP kéréseket kiszolgálják, ezek blokkolódnak.
- Amikor egy-egy fájl megérkezik, a blokkolás megszűnik, és a böngésző megjeleníti a tartalmat.

Példa: több távoli eljáráshívás (RPC) egyszerre

- Egy kliens több távoli szolgáltatást szeretne igénybe venni. Mindegyik kérést külön szál kezeli.
- Megvárja, amíg mindenki kérésre megérkezik a válasz.
- Ha különböző gépekre irányulnak a kérések, akár **lineáris mértékű gyorsulás** is elérhető így.

Szálak a szerveroldalon

Cél: a hatékonyság növelése

- Szálakat **sokkal** olcsóbb elindítani, mint folyamatokat (idő- és tárigény szempontjából egyaránt).
- Mivel egy processzor csak egy szálat tud végrehajtani, a **többprocesszoros rendszerek** kapacitását csak többszálú szerverek képesek kihasználni.
- A kliensekhez hasonlóan, **a hálózat késleltetését lehet elfedni** azzal, ha egyszerre több kérést dolgoz fel a szerver.

Cél: a program szerkezetének javítása

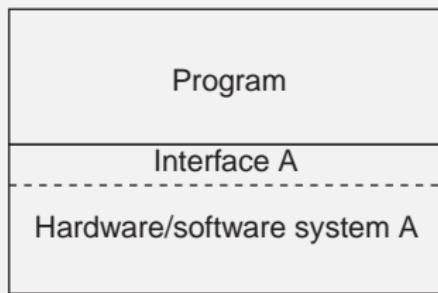
- A program jobban kezelhető lehet, ha sok egyszerű, blokkoló hívást alkalmaz, mint más szerkezet esetén. Ez némi teljesítményvesztéssel járhat.
- A többszálú programok sokszor **kisebbek** és **könnyebben érhetőek**, mert jobban átlátható, merre halad a vezérlés.

Virtualizáció

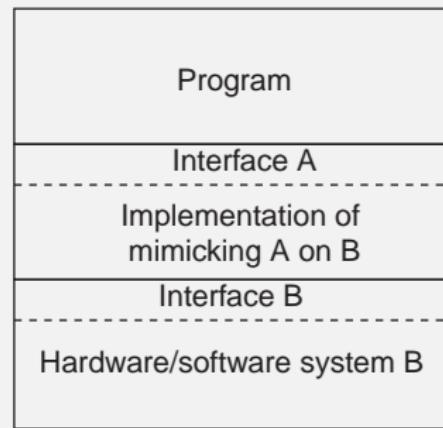
Fontossága

A virtualizáció szerepe egyre nő több okból.

- A hardver gyorsabban fejlődik a szoftvernél
- Növeli a kód hordozhatóságát és költözhetőségét
- A hibás vagy megtámadott rendszereket könnyű így elkülöníteni



(a)

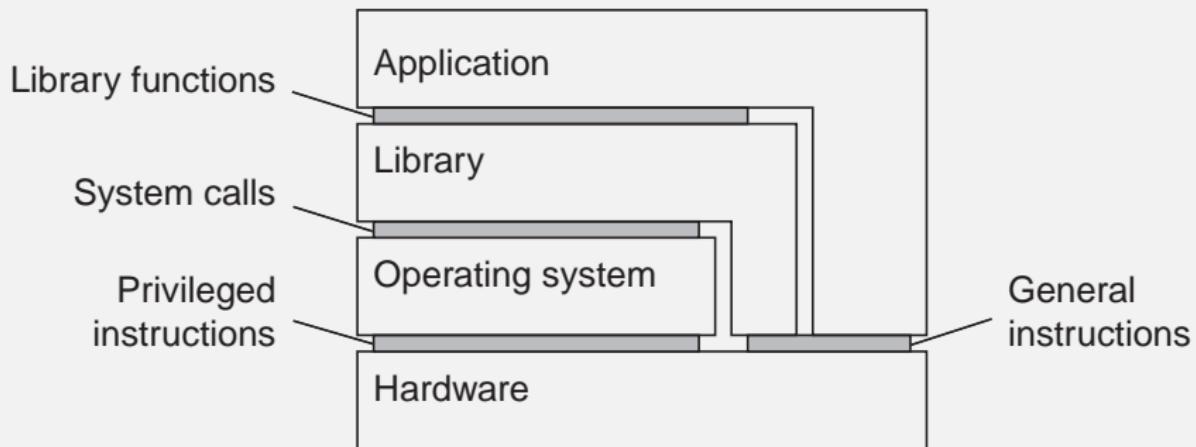


(b)

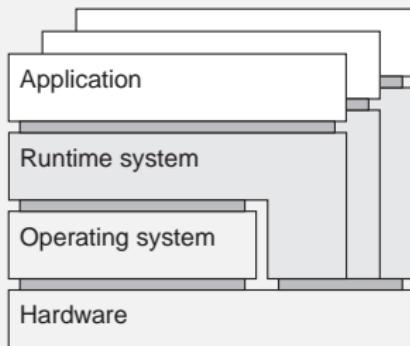
A virtuális gépek szerkezete

Virtualizálható komponensek

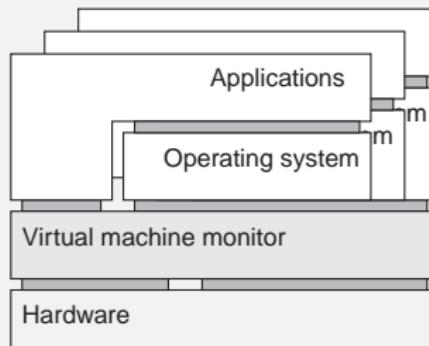
A rendszereknek sokfajta olyan rétege van, amely mentén virtualizálni lehet a komponenseket. Mindig eldöntendő, milyen **interfészeket** kell szolgáltatnia a virtuális gépnek (és milyeneket vehet igénybe).



Process VM, VM monitor



(a)



(b)

- **Process VM:** A virtuális gép (virtual machine, VM) közönséges programként fut, bájkódot (előfordított programkódot) hajt végre. Pl. JVM, CLR, de vannak speciális célúak is, pl. ScummVM.
- **VM Monitor (VMM), hypervisor:** Hardver teljeskörű virtualizációja, bármilyen program és operációs rendszer futtatására képes. Pl. VMware, VirtualBox.

VM monitorok működése

VM monitorok működése

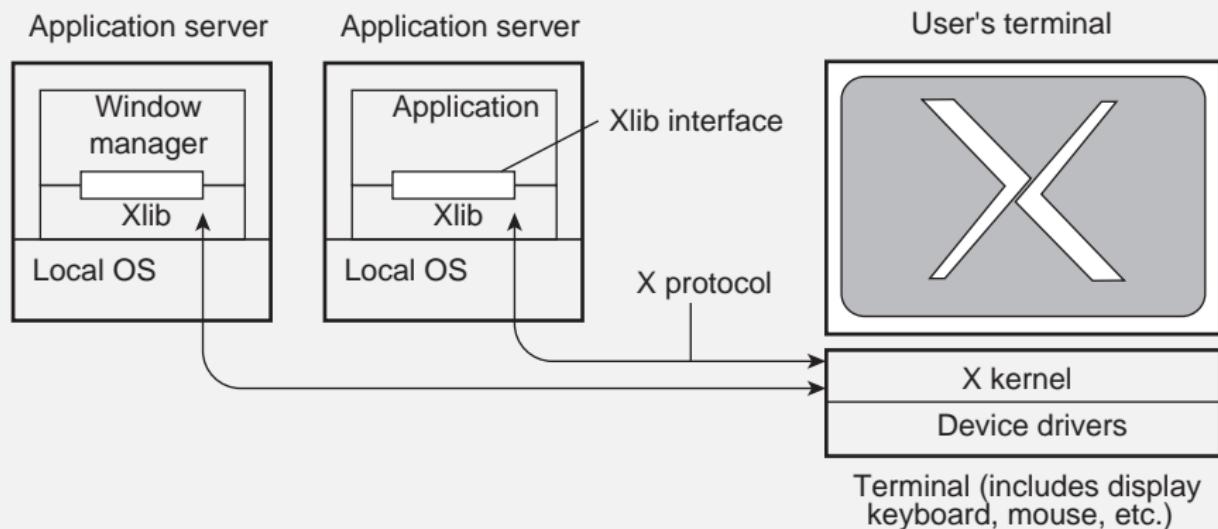
Sok esetben a VMM egy operációs rendszeren belül fut.

- A VMM a futtatott gépi kódú utasításokat átalakítja a gazdagép utasításaivá, és azokat hajtja végre.
- A **rendszerhívásokat** és egyéb **privilegizált utasításokat**, amelyek végrehajtásához az operációs rendszer közreműködésére lenne szükség, megkülönböztetett módon kezeli.

Kliens: felhasználói felület

Essence

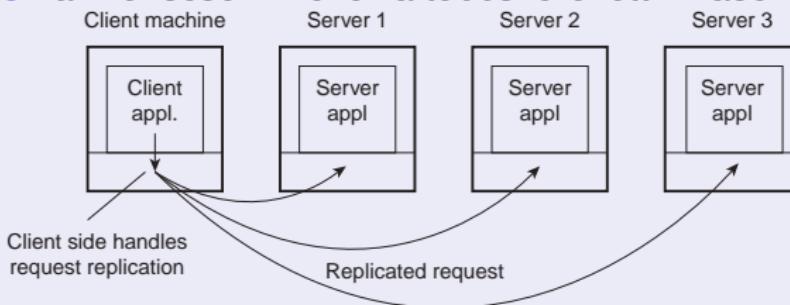
A kliensoldali szoftver egyik legfontosabb feladata a (grafikus) felhasználói interfész biztosítása.



Kliens: átlátszóság

A kliensekkel kapcsolatos főbb átlátszóságok

- **hozzáférési:** az RPC kliensoldali csonkja
- **elhelyezési/áthelyezési:** a kliensoldali szoftver tartja számon, hol helyezkedik el az erőforrás
- **többszörözési:** a klienscsonk kezeli a többszörözött hívásokat



- **meghibásodási:** sokszor csak a klienshez helyezhető – csak ott jelezhető a kommunikációs probléma

Szerver: általános szerkezet

Általános modell

szerver: Olyan folyamat, amely egy (vagy akár több) porton várja a kliensek kéréseit. Egy adott porton (ami egy 0 és 65535 közötti szám) a szerver egyfajta szolgáltatást nyújt.

A 0-1023 portok közismert szolgáltatásokat nyújtanak, ezeket Unix alapú rendszereken csak rendszergazdai jogosultsággal lehet foglalni.

ftp-data	20	File Transfer [adatátvitel]
ftp	21	File Transfer [vezérlés]
ssh	22	Secure Shell
telnet	23	Telnet
smtp	25	Simple Mail Transfer
login	49	Login Host Protocol

Szerver: általános szerkezet

Szerverfajták

szuperszerver : Olyan szerver, amelyik több porton figyeli a bejövő kapcsolatokat, és amikor új kérés érkezik, új folyamatot/szálat indít annak kezelésére. Pl. Unix rendszerekben: *inetd*.

iteratív↔konkurens szerver : Az iteratív szerverek egyszerre csak egy kapcsolatot tudnak kezelni, a konkurensek párhuzamosan többet is.

Szerver: sóvon kívüli kommunikáció

Sürgős üzenetek küldése

Meg lehet-e szakítani egy szerver működését kiszolgálás közben?

Külön port

A szerver két portot használ, az egyik a sürgős üzeneteknek van fenntartva:

- Ezt külön szál/folyamat kezeli
- Amikor fontos üzenet érkezik, a normál üzenet fogadása szünetel
- A szálnak/folyamatnak nagyobb prioritást kell kapnia, ehhez az operrendszer támogatása szükséges

Sóvon kívüli kommunikáció

Sóvon kívüli kommunikáció használata, ha rendelkezésre áll:

- Pl. a TCP protokoll az eredeti kérés kapcsolatán keresztül képes sürgős üzenetek továbbítására
- Szignálok formájában kapható el a szerveren belül

Szerver: állapot

Állapot nélküli szerver

Nem tart fenn információkat a kliensről a kapcsolat bontása után.

- Nem tartja számon, melyik kliens milyen fájlból kért adatokat
- Nem ígéri meg, hogy frissen tartja a kliens gyorsítótárát
- Nem tartja számon a bejelentkezett klienseket: nincsen munkamenet (session)

Következmények

- A kliensek és a szerverek **teljesen függetlenek egymástól**
- Kevésbé valószínű, hogy inkonzisztencia lép fel azért, mert valamelyik oldal összeomlik
- A **hatékonyság rovására mehet**, hogy a szerver nem tud semmit a kliensről, pl. nem tudja előre betölteni azokat az adatokat, amelyekre a kliensnek szüksége lehet

Szerver: állapot

Állapotteljes szerverek

Állapotot tart számon a kliensekről:

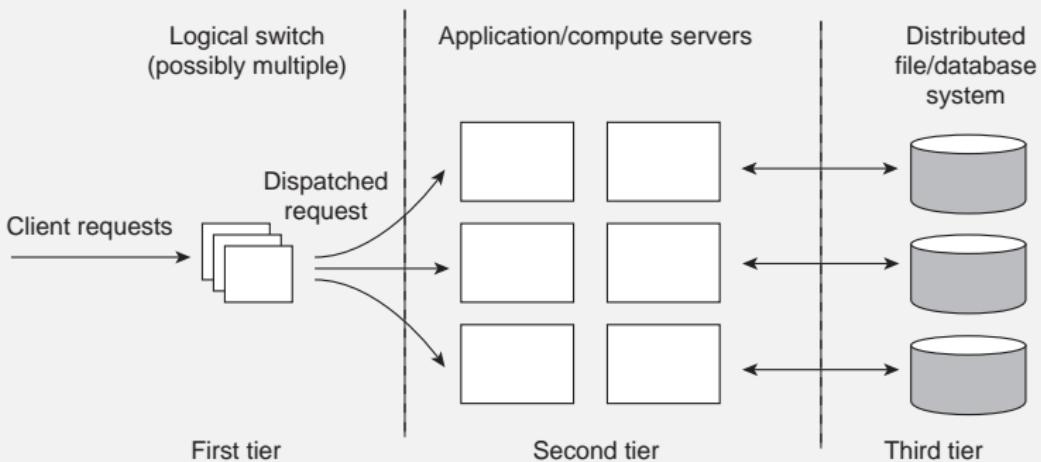
- Megjegyzi, melyik fájlokat használta a kliens, és ezeket előre meg tudja nyitni legközelebb
- Megjegyzi, milyen adatokat töltött le a kliens, és frissítéseket küldhet neki

Előnyök és hátrányok

Az állapotteljes szerverek **nagyon hatékonyak** tudnak lenni, ha a kliensek lokálisan tárolhatnak adatokat.

Az állapotteljes rendszereket megfelelően megbízhatóvá is lehet tenni a hatékonyság jelentős rontása nélkül.

Szerver: háromrétegű clusterek



A diszpeccserréteg

Az első réteg feladata nagyon fontos: a beérkező kéréseket hatékonyan kell a megfelelő szerverhez továbbítani.

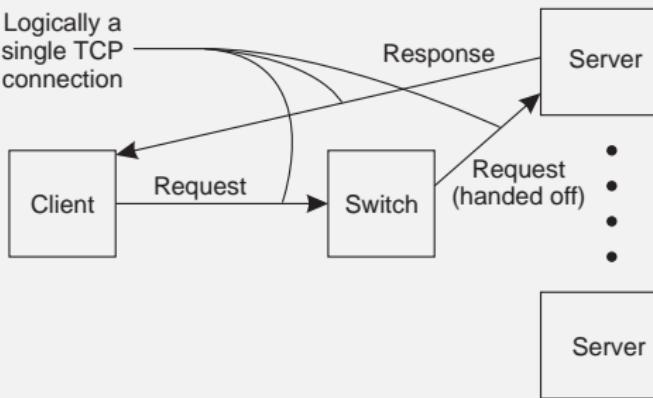
A kérések kezelése

Szűk keresztmetszet

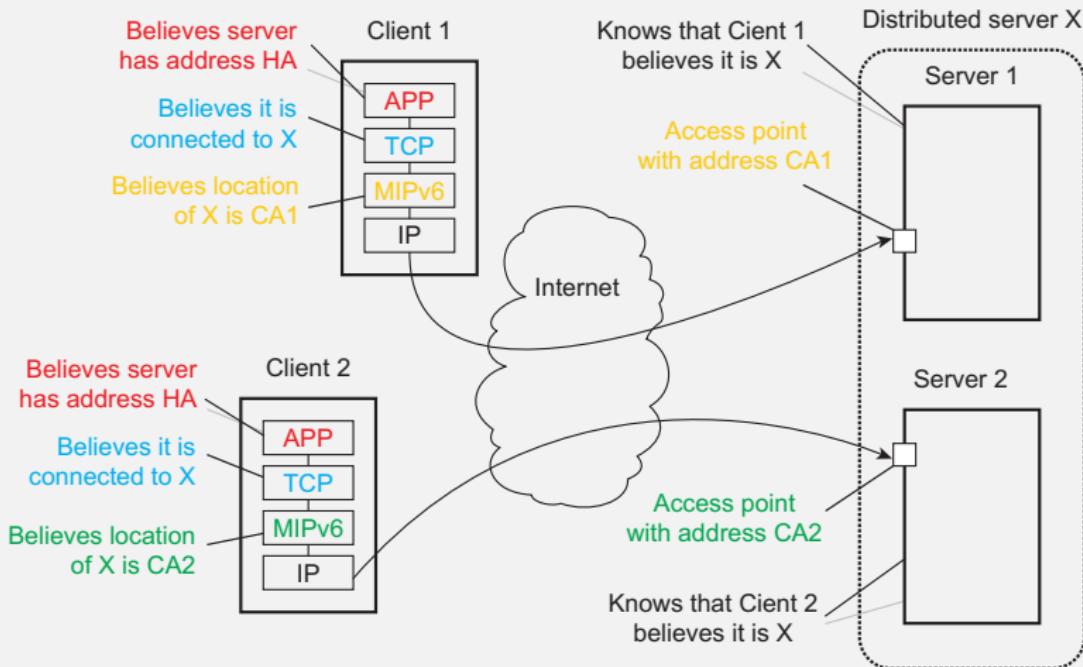
Ha minden kapcsolatot végig az első réteg kezel, könnyen szűk keresztmetszetté válhat.

Egy lehetséges megoldás

A terhelés csökkenthető, ha a kapcsolatot átadjuk ([TCP handoff](#)).



Elosztott rendszerek: mobil IPv6



Elosztott rendszerek: mobil IPv6

Essence

A mobil IPv6-ot támogató kliensek az elosztott szolgáltatás bármelyik peer-jéhez kapcsolódnak.

- A *C* kliens kapcsolódik a szerver **otthonának** (home address, *HA*) IPv6 címéhez
- A *HA* címen a szerver **hazai ügynöke** (home agent) fogadja a kérést, és a megfelelő **felügyeleti címre** (care-of address, *CA*) továbbítja
- Ezután *C* és *CA* már közvetlenül tudnak kommunikálni (*HA* érintése nélkül)

Példa: kollaboratív CDN-ek

Az origin server tölti be *HA* szerepét, és átadja a beérkező kapcsolatot a megfelelő peer szervernek. A kliensek számára az origin és a peer egy szervernek látszik.

Kódmigráció: jellemző feladatok

Kódmigráció

kódmigráció: olyan kommunikáció, amely során nem csak adatokat küldünk át

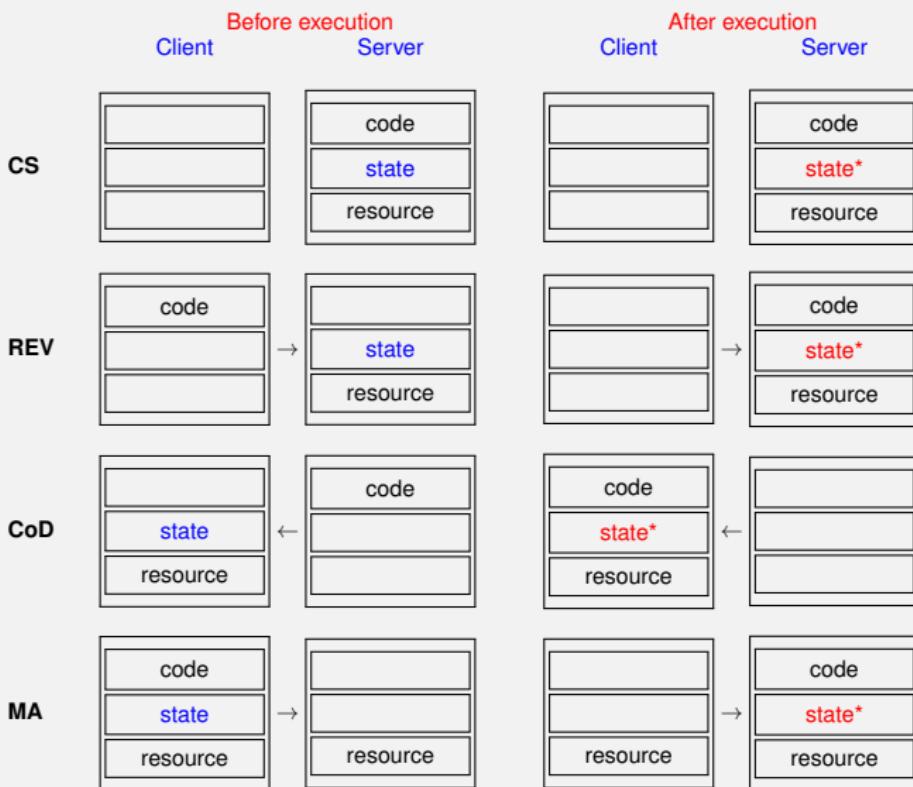
Jellemző feladatok

Néhány jellemző feladat, amelyhez kódmigrációra van szükség.^a

- Client-Server: a szokásos kliens-szerver kommunikáció, nincsen kódmigráció
- Remote Evaluation: a kliens feltölti a kódot, és a szerveren futtatja
- Code on Demand: a kliens letölti a kódot a szerverről, és helyben futtatja
- Mobile Agent: a mobil ágens feltölti a kódját és az állapotát, és a szerveren folytatja a futását

^aA következő fólia ezeket a rövidítéseket tartalmazza.

Kódmigráció: jellemző feladatok



Kódmigráció: gyenge és erős mobilitás

Objektumkomponensek

- **Kódszegmens**: a programkódot tartalmazza
- **Adatszegmens**: a futó program állapotát tartalmazza
- **Véghajtási szegmens**: a futtató szál környezetét tartalmazza

Gyenge mobilitás

A kód- és adatszegmens mozgatása (a kód újraindul):

- Viszonylag egyszerű megtenni, különösen, ha a kód hordozható
- Irány szerint: **feltöltés** (push, ship), **letöltés** (pull, fetch)

Erős mobilitás

A komponens a véghajtási szegmenssel együtt költözik

- **Migráció**: az objektum átköltözik az egyik gépről a másikra
- **Klónozás**: a kód másolata kerül a másik gépre, és ugyanabból az állapotból indul el, mint az eredeti; az eredeti is fut tovább

Kódmigráció: az erőforrások elérése

Erőforrások elérése

Az eredeti gépen található erőforrások költözés után a kód számára távoliakká válnak.

Erőforrás–gép kötés erőssége

- **Mozdíthatatlan:** nem költöztethető (pl. fizikai hardver)
- **Rögzített:** költöztethető, de csak drágán (pl. nagy adatbázis)
- **Csatolatlan:** egyszerűen költöztethető (pl. gyorsítótár)

Komponens–erőforrás kötés jellege

Milyen jellegű erőforrásra van szüksége a komponensnek?

- **Azonosítókapcsolt:** egy konkrét (pl. a cég adatbázisa)
- **Tartalomkapcsolt:** adott tartalmú (pl. bizonyos elemeket tartalmazó cache)
- **Típuskapcsolt:** adott jellegű (pl. színes nyomtató)

Kódmigráció: az erőforrások elérése

Kapcsolat az erőforrással

Hogyan tud a komponens kapcsolatban maradni az erőforrással?

- **Típuskapcsolt** erőforrás esetén a legkönnyebb **újrakapcsolódni** egy lokális, megfelelő típusú erőforráshoz
- **Azonosítókapcsolt** vagy **tartalomkapcsolt** esetben:
 - **rendszerszintű hivatkozást létesíthetünk az eredeti erőforrásra,**
 - **mozdíthatatlan** erőforrások esetén ez az egyetlen lehetőség
 - minden más esetben is szóba jöhet, de általában van jobb megoldás
 - **azonosítókapcsolt** erőforrást érdemes **áthelyezni^a**
 - **tartalomkapcsolt** erőforrást érdemes **lemásolni^a**

^aha nem túl költséges

Kódmigráció: heterogén rendszerben

Nehézségek

- A célgép nem biztos, hogy képes futtatni a migrált kódot
- A processzor-, szál- és/vagy folyamatkörnyezet nagyban függ a lokális hardvertől, operánszertől és futtatókörnyezettől

Megoldás problémás esetekben

Virtuális gép használata: akár process VM, akár hypervisor.

Természetesen a virtuális gépnek elérhetőnek kell lennie mindkét környezetben.

Elosztott rendszerek: Alapelvek és paradigmák Distributed Systems: Principles and Paradigms

Maarten van Steen¹ Kitlei Róbert²

¹VU Amsterdam, Dept. Computer Science

²ELTE Informatikai Kar

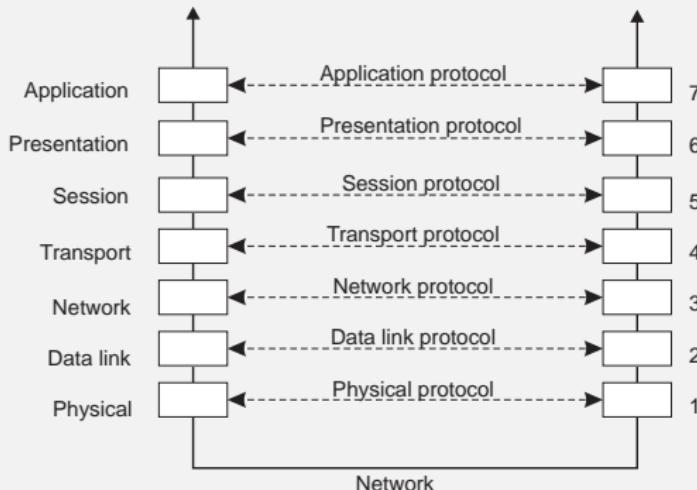
4. rész: Kommunikáció

2015. május 24.

Tartalomjegyzék

Fejezet
01: Bevezetés
02: Architektúrák
03: Folyamatok
04: Kommunikáció
05: Elnevezési rendszerek
06: Szinkronizáció
07: Konzisztencia & replikáció
08: Hibatűrés
10: Objektumalapú elosztott rendszerek
11: Elosztott fájlrendszerek
12: Elosztott webalapú rendszerek

Az ISO/OSI hálózatkezelési modell



Hátrányok

- Csak az üzenetküldésre koncentrál
- Az (5) és (6) rétegek legtöbbször nem jelennek meg ilyen tisztán
- Az elérési átlátszóság nem teljesül ebben a modellben

Az alsó rétegek

A rétegek feladatai

- **Fizikai réteg:** a bitek átvitelének fizikai részleteit írja le
- **Adatkapcsolati réteg:** az üzeneteket keretekre tagolja, célja a hibajavítás és a hálózat terhelésének korlátozása
- **Hálózati réteg:** a hálózat távoli gépei között közvetít csomagokat útválasztás (**routing**) segítségével

Szállítási réteg

Absztraktiós alap

A legtöbb elosztott rendszer a szállítási réteg szolgáltatásaira épít.

A legfőbb protokollok

- TCP: kapcsolatalapú, megbízható, sorrendhelyes átvitel
- UDP: nem (teljesen) megbízható, általában kis üzenetek (datagram) átvitele

Csoportcímzés

IP-alapú többcímű üzenetküldés (multicasting) sokszor elérhető, de legfeljebb a lokális hálózaton belül használatos.

Köztesréteg

Szolgáltatásai

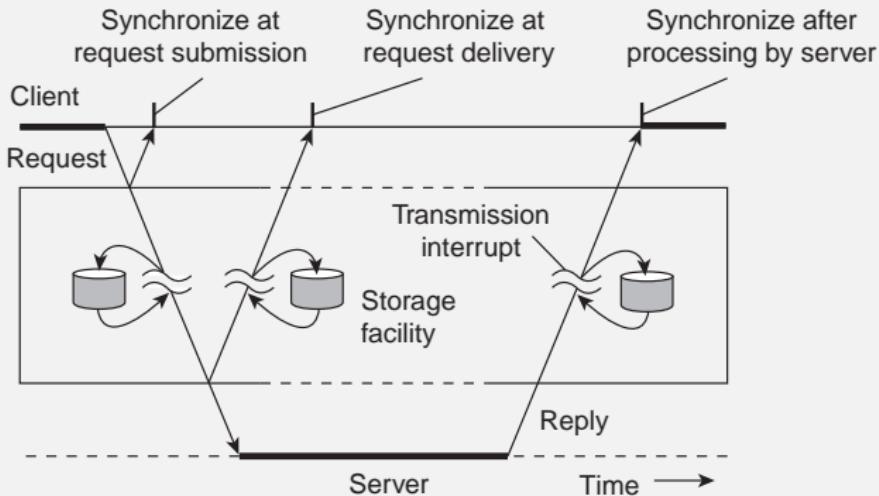
A köztesrétegbe (middleware) olyan szolgáltatásokat és protokollokat szokás sorolni, amelyek sokfajta alkalmazáshoz lehetnek hasznosak.

- Sokfajta **kommunikációs protokoll**
- **Sorosítás** ((de)serialization, (un)marshalling), adatok reprezentációjának átalakítása (elküldésre vagy elmentésre)
- **Elnevezési protokollok** az erőforrások megosztásának megkönnyítésére
- **Biztonsági protokollok** a kommunikáció biztonságossá tételere
- **Skálázási mechanizmusok** adatok replikációjára és gyorsítótárazására

Alkalmazási réteg

Az alkalmazások készítőinek csak az **alkalmazás-specifikus** protokollokat kell önmaguknak implementálniuk.

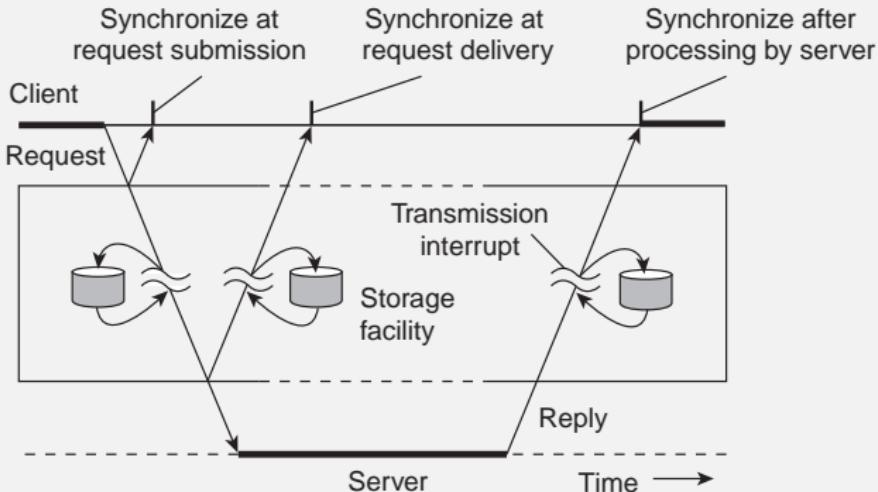
A kommunikáció fajtái



A kommunikáció lehet...

- időleges (transient) vagy megtartó (persistent)
- aszinkron vagy szinkron

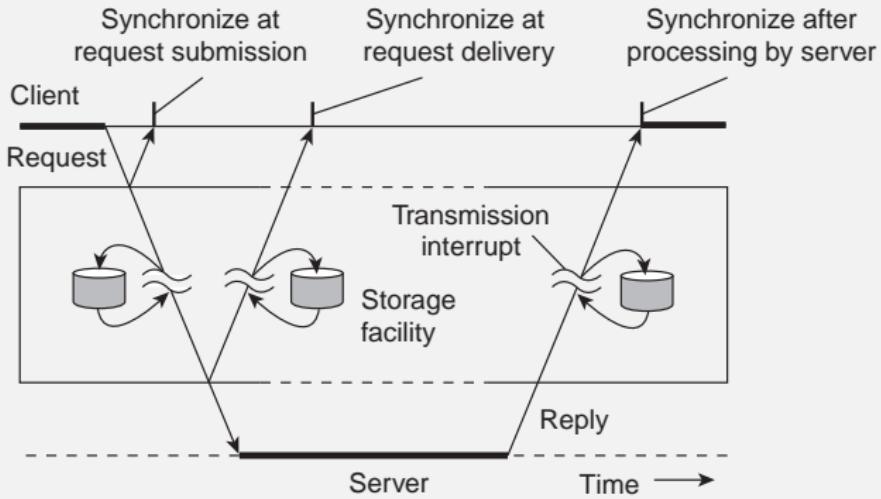
A kommunikáció fajtái



Időleges vs megtartó

- **Megtartó kommunikáció:** A kommunikációs rendszer hajlandó huzamosan tárolni az üzenetet.
- **Időleges kommunikáció:** A kommunikációs rendszer elveti az üzenetet, ha az nem kézbesíthető.

A kommunikáció fajtái



A szinkronizáció lehetséges helyei

- Az üzenet elindításakor
- Az üzenet beérkezésekor
- A kérés feldolgozása után

Kliens–szerver modell

Általános jellemzők

A kliens–szerver modell jellemzően **időleges, szinkron kommunikációt** használ.

- A kliensnek és a szervernek egyidőben kell aktívnak lennie.
- A kliens blokkolódik, amíg a válasz meg nem érkezik.
- A szerver csak a kliensek fogadásával foglalkozik, és a kérések kiszolgálásával.

A szinkron kommunikáció hátrányai

- A kliens nem tud tovább dolgozni, amíg a válasz meg nem érkezik
- A hibákat rögtön kezelni kell, különben feltartjuk a klienst
- Bizonyos feladatokhoz (pl. levelezés) nem jól illeszkedik

Üzenetküldés

Üzenetorientált köztesréteg (message-oriented middleware, MOM)

Megtartó, aszinkron kommunikációs architektúra.

- Segítségével a folyamatok üzeneteket küldhetnek egymásnak
- A küldő félnek nem kell válaszra várakoznia, foglalkozhat másossal
- A köztesréteg gyakran hibatűrést biztosít

RPC: alapok

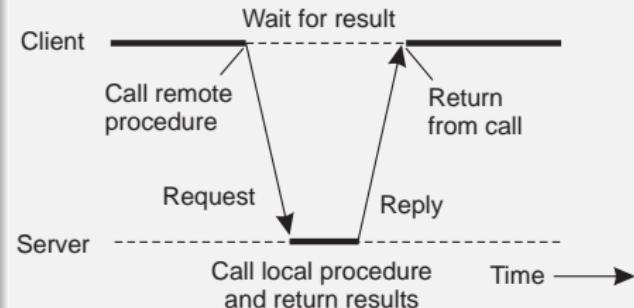
Az RPC alapötlete

- Az alprogramok használata természetes a fejlesztés során
- Az alprogramok a jó esetben egymástól függetlenül működnek („fekete doboz”),
- ... így akár egy távoli gépen is végrehajthatóak

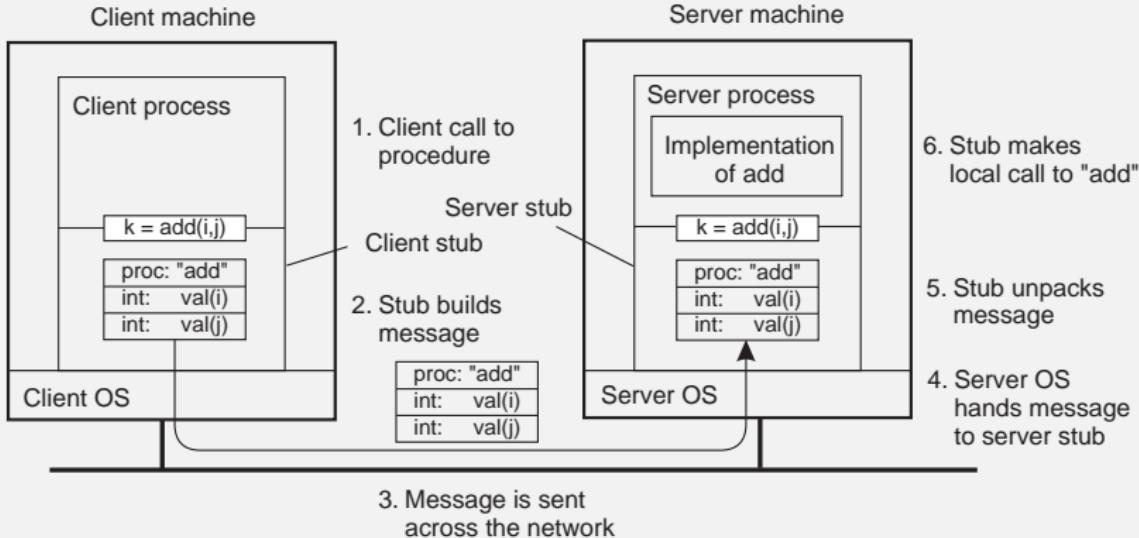
Távoli eljáráshívás (remote procedure call, RPC)

A távoli gépen futtatandó eljárás^a eléréséhez hálózati kommunikációra van szükség, ezt eljáráshívási mechanizmus fedi el.

^atekintsük az alprogram szinonímájának



RPC: a hívás lépései



- 1 A kliensfolyamat lokálisan meghívja a klienscsonkokat.
- 2 Az becsomagolja az eljárás azonosítóját és paramétereit, meghívja az OS-t.
- 3 Az átküldi az üzenetet a távoli OS-nek.
- 4 Az átadja az üzenetet a szervercsonknak.
- 5 Az kicsomagolja a paramétereket, átadja a szervernek.
- 6 A szerver lokálisan meghívja az eljárást, megkapja a visszatérési értéket.
- 7 Ennek visszaküldése a klienshez hasonlóan zajlik, fordított irányban.

RPC: paraméterátadás

A paraméterek sorosítása

A második lépésben a klienscsonk elkészíti az üzenetet, ami az egyszerű bemásolásnál összetettebb lehet.

- A kliens- és a szervergépen eltérhet az adatábrázolás (eltérő bájtsorrend)
- A sorosítás során bájtsorozat készül az értékből
- Rögzíteni kell a paraméterek kódolását:
 - A primitív típusok reprezentációját (egész, tört, karakteres)
 - Az összetett típusok reprezentációját (tömbök, egyéb adatszerkezetek)
- A két csonknak fordítania kell a közös formátumról a gépeik formátumára

RPC: paraméterátadás

RPC paraméterátadás szemantikája

- **Érték–eredmény szerinti paraméterátadási szemantika:** pl. figyelembe kell venni, hogy ha (a kliensoldalon ugyanoda mutató) hivatkozásokat adunk át, azokról ez a hívott eljárásban nem látszik.
- **Minden feldolgozandó adat** paraméterként kerül az eljáráshoz; **nincsen globális hivatkozás.**

Átlátszóság

Nem érhető el teljes mértékű elérési átlátszóság.

Távoli hivatkozás

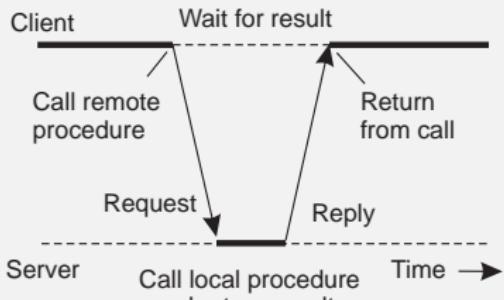
Távoli hivatkozás bevezetésével növelhető az elérési átlátszóságot:

- A távoli adat egységesen érhető el
- A távoli hivatkozásokat **át lehet paraméterként adni**

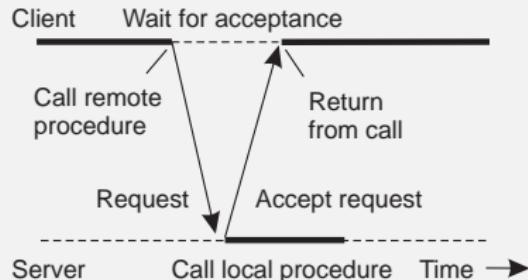
Aszinkron RPC

Az RPC „javítása”

A szerver nyugtálja az üzenet megérkezését. Választ nem vár.



(a)

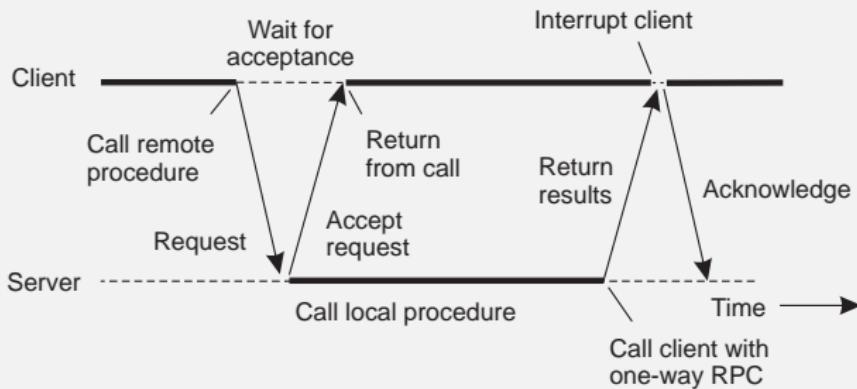


(b)

Késleltetett szinkronizált RPC

Késleltetett szinkronizált RPC

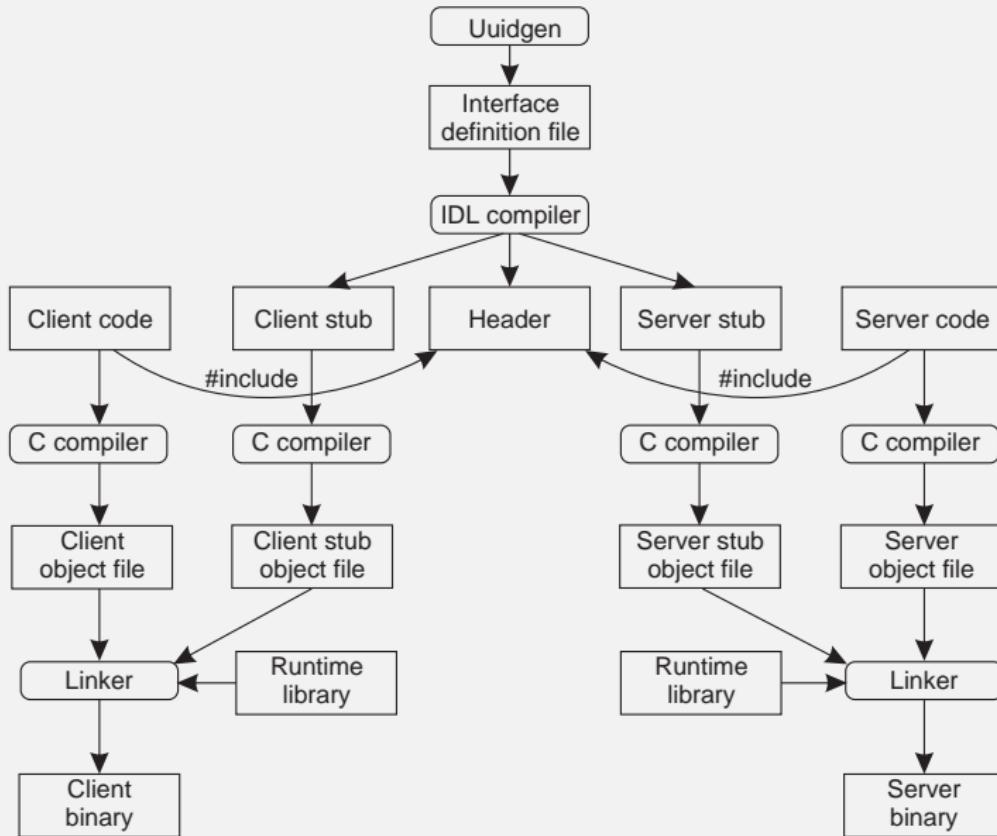
Ez két aszinkron RPC, egymással összehangolva.



További lehetőség

A kliens elküldheti a kérését, majd időnként lekérdezheti a szervertől, kész-e már a válasz.

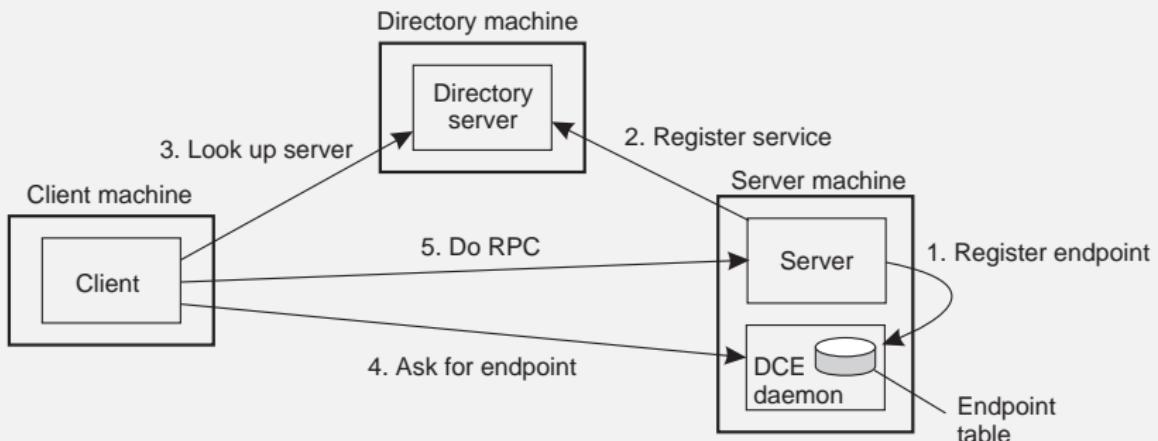
RPC: a használt fájlok



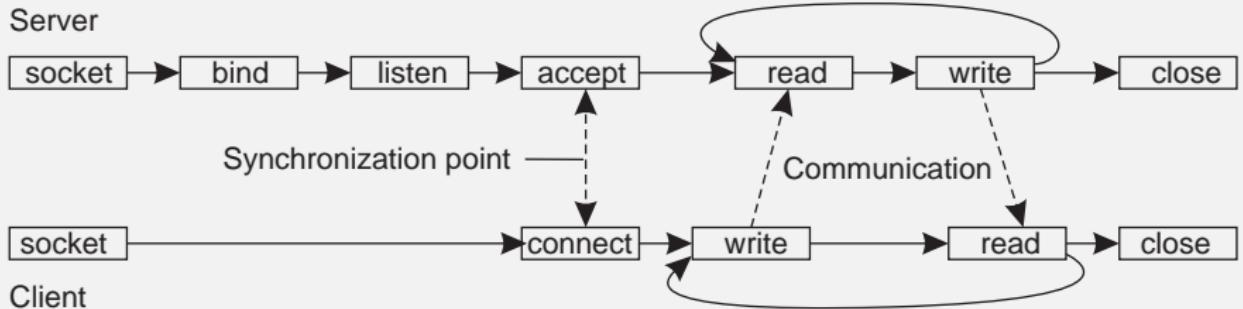
RPC: a kliens csatlakozása a szerverhez

A kliens

- 1 A szolgáltatások katalógusba jegyzik be (globálisan és lokálisan is), melyik gépen érhetőek el. (1-2)
- 2 A kliens kikeresi a szolgáltatást a katalógusból. (3)
- 3 A kliens végpontot igényel a démontól a kommunikációhoz. (4)



Időleges kommunikáció: socket



Socket: példa Python nyelven

Szerver

```
import socket
HOST = ''
PORT = SERVERPORT
srvsock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
srvsock.bind((HOST, PORT))
srvsock.listen(N)          # legfeljebb N kliens várakozhat
clsock, addr = srvsock.accept() # lokális végpont + távoli végpont címe
while True: # potenciálisan végtelen ciklus
    data = clsock.recv(1024)
    if not data: break
    clsock.send(data)
clsock.close()
```

Kliens

```
import socket
HOST = 'distsys.cs.vu.nl'
PORT = SERVERPORT
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((HOST, PORT))
s.send('Hello, world')
data = s.recv(1024)
s.close()
```

Üzenetorientált köztesréteg

Működési elv

A köztesréteg **várakozási sorokat** (queue) tart fenn a rendszer gépein. A kliensek az alábbi műveleteket használhatják a várakozási sorokra.

PUT	Üzenetet tesz egy várakozási sor végére
GET	Blokkol, amíg a sor üres, majd leveszi az első üzenetet
POLL	Nem-blokkoló módon lekérdezi, van-e üzenet, ha igen, leveszi az elsőt
NOTIFY	Kezelőrutint telepít a várakozási sorhoz, amely minden beérkező üzenetre meghívódik

Üzenetközvetítő

Üzenetsorkezelő rendszer homogenitása

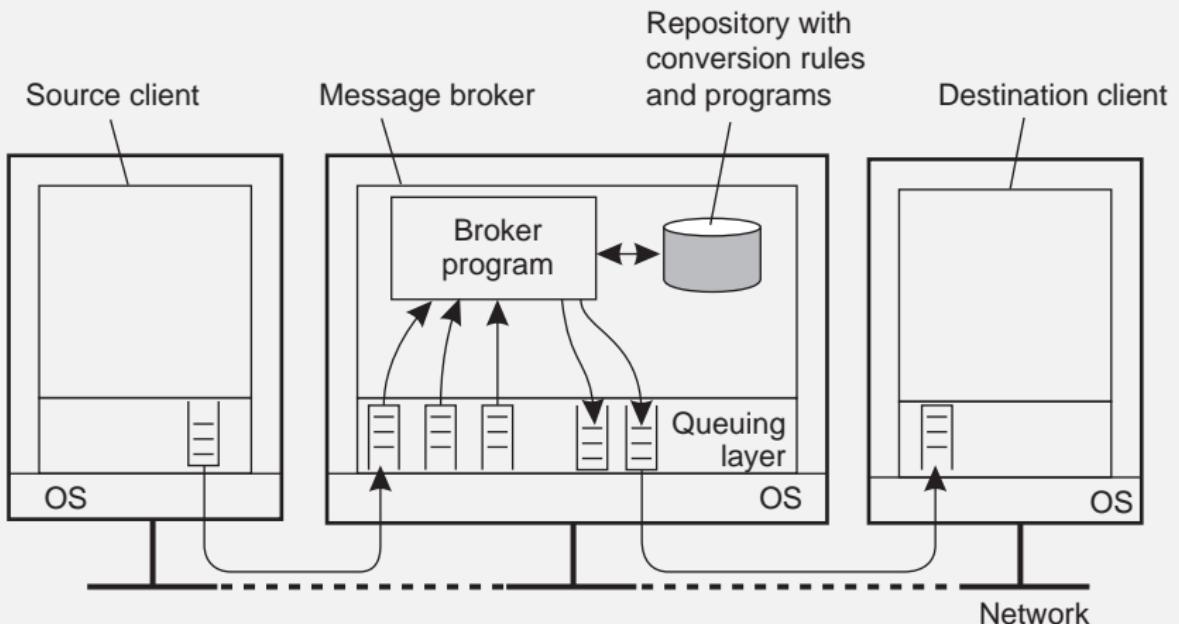
Az üzenetsorkezelő rendszerek feltételezik, hogy a rendszer minden eleme **közös protokollt használ**, azaz az üzenetek szerkezete és adatábrázolása megegyező.

Üzenetközvetítő

üzenetközvetítő (message broker): Olyan központi komponens, amely heterogén rendszerben gondoskodik a megfelelő konverziókról.

- Átalakítja az üzeneteket a fogadó formátumára.
- Szerepe szerint igen gyakran **átjáró** (application-level gateway, proxy) is, azaz a közvetítés mellett további (pl. biztonsági) funkciókat is nyújt
- Az üzenetek tartalmát is megvizsgálhatják az útválasztáshoz (**subject based** vagy **object based** routing) ⇒ **Enterprise Application Integration**

Üzenetközvetítő

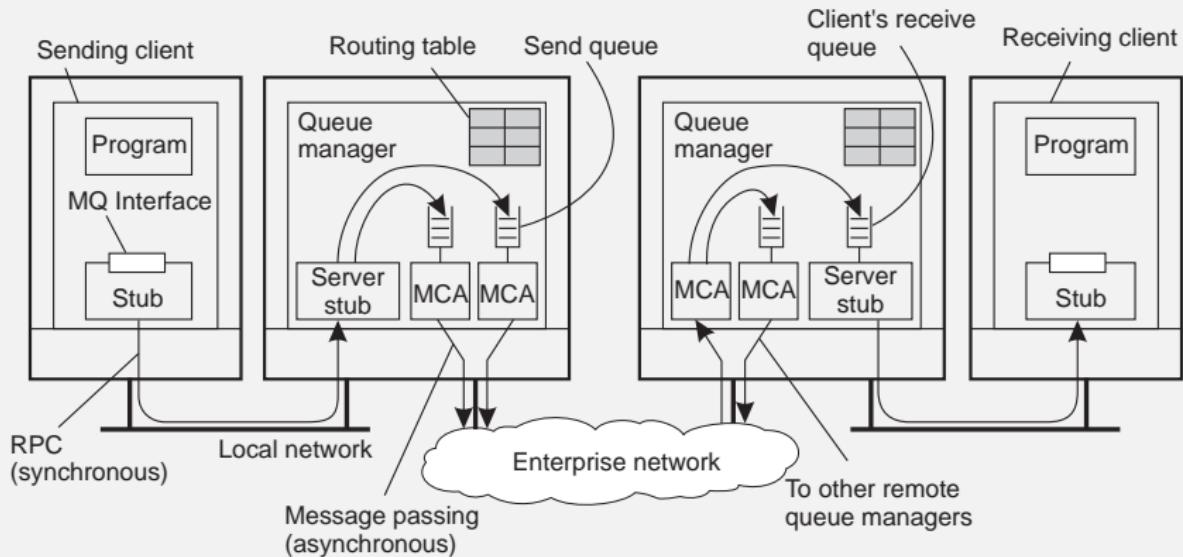


Példa: WebSphere MQ (IBM)

Működési elv

- Az üzenetkezelők neve itt sorkezelő (queue manager); adott alkalmazásoknak címzett üzeneteket fogadnak
 - Az üzenetkezelőt össze lehet szerkeszteni a kliensprogrammal
 - Az üzenetkezelő RPC-n keresztül is elérhető
- Az útválasztótáblák (routing table) megadják, melyik kimenő csatornán kell továbbítani az üzenetet
- A csatornákat üzenetcsatorna-ügynökök (message channel agent, MCA) kezelik
 - Kiépítik a hálózati kapcsolatokat (pl. TCP/IP)
 - Ki- és becsomagolják az üzeneteket, és fogadják/küldik a csomagokat a hálózatról

Példa: WebSphere MQ (IBM)

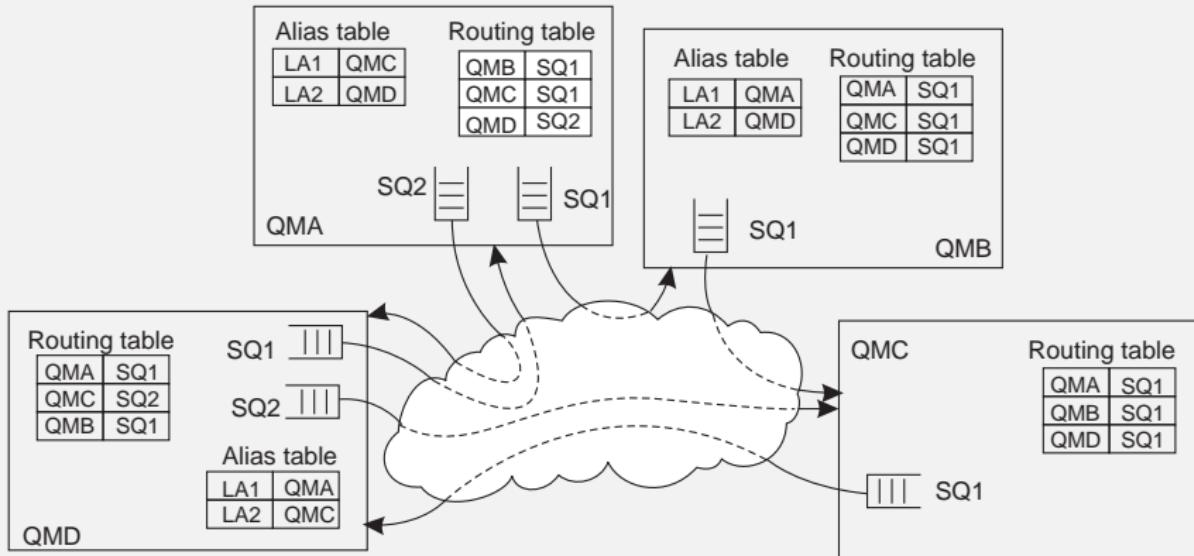


- A csatornák egyirányúak
- A sorkezelőkhöz beérkező üzenetek automatikusan továbbítódnak a megfelelő lokális MCA-hoz
- Az útválasztás paramétereit kézzel adják meg

Példa: WebSphere MQ (IBM)

Álnevek

Távoli üzenetkezelőhöz álnevet (alias) is lehet rendelni, ez csak a lokális üzenetkezelőn belül érvényes.



Folyamatos média

Az idő szerepe

Az eddig tárgyalt kommunikációfajtákban közös, hogy **diszkrét ábrázolásúak**: az adategységek közötti időbeli kapcsolat nem befolyásolja azok jelentését.

Folyamatos ábrázolású média

A fentiekkel szemben itt a továbbított adatok **időfüggők**.

Néhány jellemző példa:

- audio
- videó
- animációk
- szenzorok adatai (hőmérséklet, nyomás stb.)

Folyamatos média

Adatátviteli módok

Többfajta megkötést tehetünk a kommunikáció időbeliségével kapcsolatban.

- **aszinkron**: nem ad megkötést arra, hogy **mikor** kell átvinni az adatot
- **szinkron**: az egyes adatcsomagoknak megadott időtartam alatt célba kell érniük
- **izokron** vagy **izoszinkron^a**: felső és **alsó korlátot** is ad a csomagok átvitelére; a **remegés** (jitter) így korlátozott mértékű

^a α =(fosztóképző), τ_{soc} =egyenlő, σ_{uv} =együtt, χ_{proto} =idő

Folyam

Adatfolyam

adatfolyam: Izokron adatátvitelt támogató kommunikációs forma.

Fontosabb jellemzők

- Egyirányú
- Legtöbbször egy **forrástól** (source) folyik egy vagy több **nyelő** (sink) felé
- A forrás és/vagy a nyelő gyakran közvetlenül kapcsolódik hardverelemekhez (pl. kamera, képernyő)
- **egyszerű folyam:** egyfajta adatot továbbít, pl. egy audiocsatornát vagy csak videót
- **összetett folyam:** többfajta adatot továbbít, pl. sztereo audiot vagy hangot+videót

Folyam: QoS

Szolgáltatás minősége

A folyamokkal kapcsolatban sokfajta követelmény írható elő, ezeket összefoglaló néven a **szolgáltatás minőségének** (Quality of Service, QoS) nevezzük. Ilyen jellemzők a következők:

- A folyam átvitelének „sebessége”: **bit rate**.
- A folyam megindításának **legnagyobb megengedett késleltetése**.
- A folyam adategységeinek **megadott idő** alatt el kell jutniuk a forrástól a nyelőig (**end-to-end delay**), illetve számíthat az oda-vissza út is (**round trip delay**).
- Az adategységek beérkezési időközeinek egyenetlensége: **remegés (jitter)**.

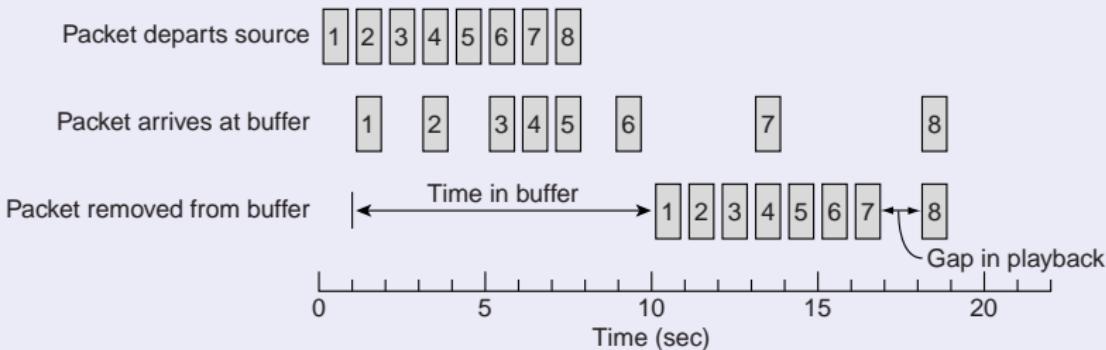
Folyam: QoS biztosítása

Differenciált szolgáltatási architektúra

Több hálózati eszköz érhető el, amelyekkel a QoS biztosítható. Egy lehetőség, ha a hálózat routerei kategorizálják az áthaladó forgalmat a beérkező adatcsomagok tartalma szerint, és egyes csomagfajták elszöbbséggel továbbítanak ([differentiated services](#)).

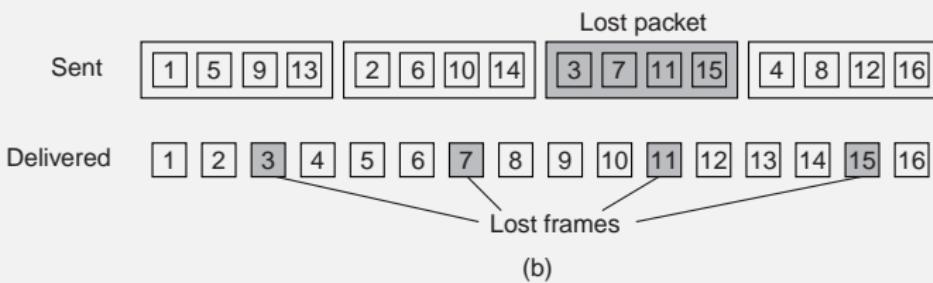
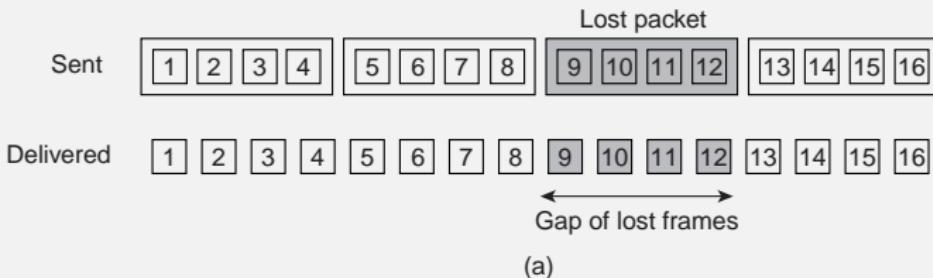
A remegés csökkentése

A routerek [pufferelhetik](#) az adatokat a remegés csökkentésére.



Folyam: QoS biztosítása

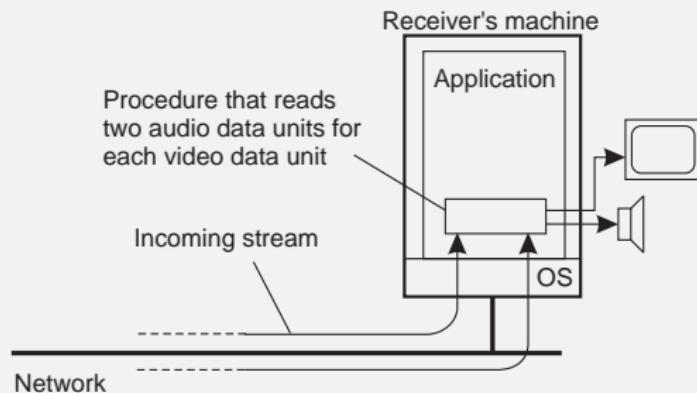
A csomagok elveszhetnek útközben. Ennek hatását mérsékelheti, ha a csomagon belül az adatelemek sorrendjét kissé módosítjuk; ennek ára, hogy a lejátszás lassabban indul meg.



Összetett folyam szinkronizációja

Szinkronizáció a nyelőnél

Az összetett folyam alfolyamait **szinkronizálni** kell a nyelőnél, különben időben elcsúszhatnak egymáshoz képest.



Multiplexálás

Másik lehetőség: a forrás már eleve egyetlen folyamot készít (**multiplexálás**). Ezek garantáltan szinkronban vannak egymással, a nyelőnél csak szét kell őket bontani (**demultiplexálás**).

Alkalmazásszintű multicasting

A hálózat minden csúcsának szeretnénk üzenetet tudjunk küldeni ([multicast](#)). Ehhez hierarchikus [overlay hálózatba](#) szervezzük őket.

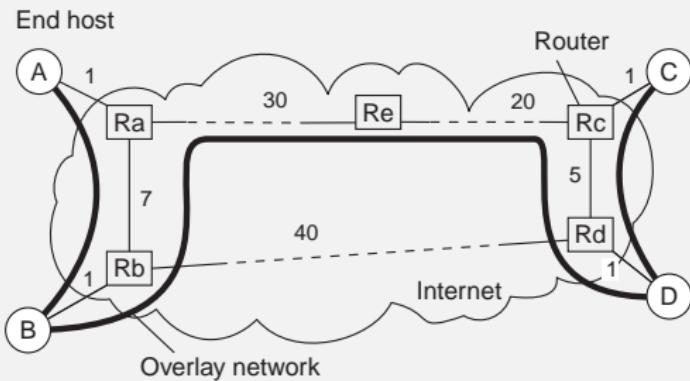
Chord struktúrában tárolt fa készítése

- A multicast hálózatunkhoz generálunk egy azonosítót, így egyszerre több multicast hálózatunk is lehet egy rendszerben.
- Tegyük fel, hogy az azonosító egyértelműen kijelöl egy csúcsot a rendszerünkben^a. Ez a csúcs lesz a fa [gyökere](#).
- Terv: a küldendő üzeneteket mindenki elküldi a gyökérhez, majd onnan a fán lefele terjednek.
- Ha a P csúcs csatlakozni szeretne a multicast hálózathoz, [csatlakozási kérést](#) küld a gyökér felé. A P csúcstól a gyökérig egyértelmű az útvonal^b; ennek minden csúcsát a fa részévé tesszük (ha még nem volt az). Így P elérhetővé válik a gyökértől.

^aEz az azonosító ún. rákövetkezője; a technikai részletek később jönnek.

^bRészletek szintén később.

Alkalmazásszintű multicasting: költségek



- **Kapcsolatok terhelése:** Mivel overlay hálózatot alkalmazunk, előfordulhat, hogy egy üzenetküldés többször is igénybe veszi ugyanazt a fizikai kapcsolatot.
- Példa:** az $A \rightarrow D$ üzenetküldés kétszer halad át az $Ra \rightarrow Rb$ élen.
- **Stretch:** Az overlayt követő és az alacsonyszintű üzenetküldés költségének hányadosa.
- Példa:** $B \rightarrow C$ overlay költsége 71, hálózati 47 $\Rightarrow stretch = 71/47$.

Járványalapú algoritmusok

Alapötlet

- Valamelyik szerveren frissítési műveletet (update) hajtottak végre, azt szeretnénk, hogy ez elterjedjen a rendszerben minden szerverhez.
- minden szerver elküldi a változást néhány szomszédjának (messze nem az összes csúcsnak) lusta módon (nem azonnal)
- Tegyük fel, hogy nincs olvasás-írás konfliktus a rendszerben.

Két alkategória

- **Anti-entrópia:** minden szerver rendszeresen kiválaszt egy másikat, és kicsérélik egymás között a változásokat.
- **Pletykálás (gossiping):** az újonnan frissült (**megfertőzött**) szerver elküldi a frissítést néhány szomszédjának (**megfertőzi** őket).

Járvány: anti-entrópia

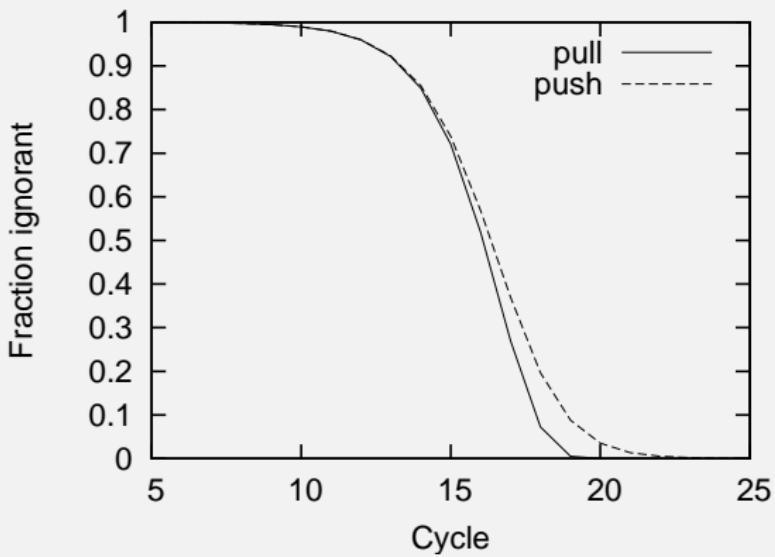
A frissítések cseréje

- P csúcs Q csúcsot választotta ki.
- **Küldés** (push): P elküldi a nála levő frissítéseket Q -nak
- **Rendelés** (pull): P bekéri a Q -nál levő frissítéseket
- **Küldés–rendelés** (push–pull): P és Q **kicserélik az adataikat**, így ugyanaz lesz mindenki tartalma.

Hatókonyság

A küldő–rendelő megközelítés esetében $\mathcal{O}(\log(N))$ nagyságrendű forduló megtétele után az összes csúcshoz eljut a frissítés.
Egy fordulónak az számít, ha mindenki csúcs megtett egy lépést.

Járvány: anti-entrópia: hatékonyság



Járvány: pletykálás

Működési elv

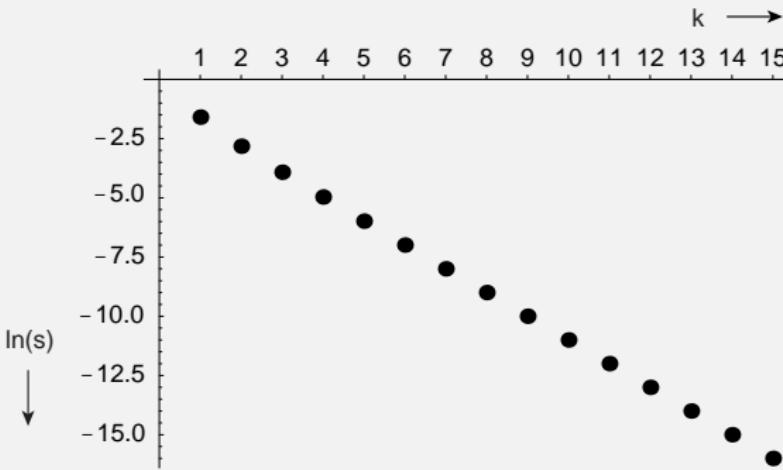
Ha az S szerver új frissítést észlelt, akkor felveszi a kapcsolatot más szerverekkel, és elküldi számukra a frissítést.

Ha olyan szerverhez kapcsolódik, ahol már jelen van a frissítés, akkor $\frac{1}{k}$ valószínűséggel abbahagyja a frissítés terjesztését.

Járvány: pletykálás: hatékonyság

Hatókonyság

Kellően sok szerver esetén a tudatlanságban maradó szerverek (akikhez nem jut el a frissítés) száma exponenciálisan csökken a k valószínűség növekedésével, de ezzel az algoritmussal **nem garantálható, hogy minden szerverhez eljut** a frissítés.



Consider 10,000 nodes		
k	s	N_s
1	0.203188	2032
2	0.059520	595
3	0.019827	198
4	0.006977	70
5	0.002516	25
6	0.000918	9
7	0.000336	3

Járvány: értékek törlése

A törlési művelet nem terjeszthető

Ha egy adat törlésének műveletét is az előzőekhez hasonlóan terjesztenénk a szerverek között, akkor a még terjedő frissítési műveletek újra létrehoznák az adatot ott, ahol a törlés eljutott.

Megoldás

A törlést speciális frissítésként: [halotti bizonyítvány](#) (death certificate) küldésével terjesztjük.

Járvány: értékek törlése

Halotti bizonyítvány törlése

A halotti bizonyítványt nem akarjuk örök ké tárolni. Mikor törölhetőek?

- **Szemétgyűjtés-jellegű megközelítés:** Egy rendszerszintű algoritmussal felismerjük, hogy mindenhol eljutott a bizonyítvány, és ekkor mindenhol eltávolítjuk. Ez a megoldás **nem jól skálázódik**.
- **elavuló bizonyítvány:** Kibocsátás után adott idővel a bizonyítvány elavul, és ekkor törölhető; így viszont nem garantálható, hogy mindenhol elér.

Járvány: példák

Példa: adatok elterjesztése

Az egyik legfontosabb és legjellemzőbb alkalmazása a járványalapú algoritmusoknak.

Példa: adatok aggregálása

Most a cél a csúcsokban tárolt adatokból új adatok kiszámítása.

Kezdetben minden egyik csúcs egy értéket tárol: x_i . Amikor két csúcs pletykál, minden kettő a tárolt értékét a korábbi értékek átlagára állítja:

$$x_i, x_j \leftarrow \frac{x_i + x_j}{2}$$

Mivel az értékek minden lépésben közelednek egymáshoz, de az összegük megmarad, minden egyik érték a teljes átlaghoz konvergál.

$$\bar{x} = \frac{\sum_i x_i}{N}$$

Elosztott rendszerek: Alapelvek és paradigmák Distributed Systems: Principles and Paradigms

Maarten van Steen¹ Kitlei Róbert²

¹VU Amsterdam, Dept. Computer Science

²ELTE Informatikai Kar

5. rész: Elnevezési rendszerek

2015. május 24.

Tartalomjegyzék

Fejezet
01: Bevezetés
02: Architektúrák
03: Folyamatok
04: Kommunikáció
05: Elnevezési rendszerek
06: Szinkronizáció
07: Konzisztencia & replikáció
08: Hibatűrés
10: Objektumalapú elosztott rendszerek
11: Elosztott fájlrendszerek
12: Elosztott webalapú rendszerek

Elnevezési rendszerek

Elnevezési rendszerek

Az elosztott rendszerek entitásai a **kapcsolódási pontjaikon** (access point) keresztül érhetőek el. Ezeket távolról a **címük** azonosítja, amely **megnevezi** az adott pontot.

Célszerű lehet az entitást a kapcsolódási pontjaitól függetlenül is elnevezni. Az ilyen nevek **helyfüggetlenek** (location independent).

Az **egyszerű neveknek** nincsen szerkezete, tartalmuk véletlen szöveg.

Az egyszerű nevek csak összehasonlításra használhatóak.

Azonosító

Egy név **azonosító**, ha egy-egy kapcsolatban áll a megnevezett egyeddel, és ez a hozzárendelés **maradandó**, azaz a név nem hivatkozhat más egyedre később sem.

Strukturálatlan nevek

Strukturálatlan nevek feloldása

Milyen lehetőségek vannak strukturálatlan nevek feloldására? (Azaz: hogyan találjuk meg a hozzárendelt kapcsolódási pontot?)

- egyszerű megoldások (broadcasting)
- otthonalapú megoldások
- elosztott hasítótáblák (strukturált P2P)
- hierarchikus rendszerek

Névfeloldás: egyszerű megoldások

Broadcasting

Kihirdetjük az azonosítót a hálózaton; az egyed visszaküldi a jelenlegi címét.

- Lokális hálózatokon túl nem skálázódik
- A hálózaton minden gépnek figyelnie kell a beérkező kérésre

Továbbítómutató

Amikor az egyed elköltözik, egy mutató marad utána az új helyére.

- A kliens elől el van fedve, hogy a szoftver továbbítómutató-láncot old fel.
- A megtalált címet vissza lehet küldeni a klienshez, így a további feloldások gyorsabban mennek.
- Földrajzi skálázási problémák
 - A hosszú láncok nem hibatűrőek
 - A feloldás hosszú időbe telik
 - Külön mechanizmus szükséges a láncok rövidítésére

Otthonalapú megközelítések

Egyrétegű rendszer

Az egyedhez tartozik egy **otthon**, ez tartja számon az egyed jelenlegi címét.

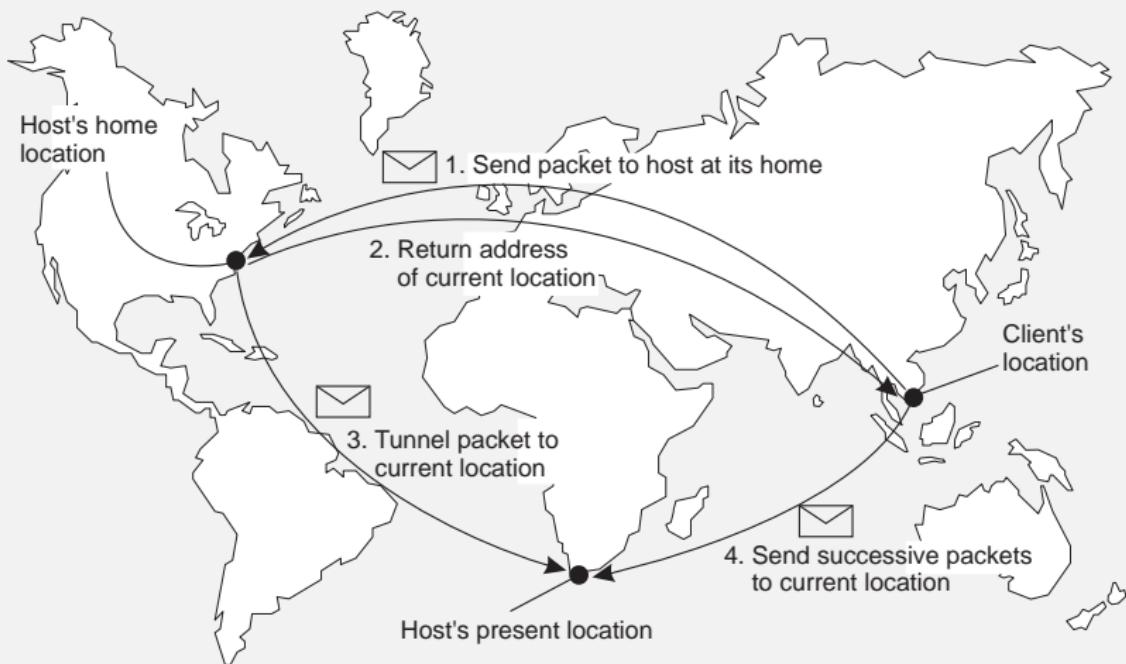
- Az egyed **otthoni címe** (home address) be van jegyezve egy névszolgáltatásba
- Az otthon számon tartja az egyed jelenlegi címét (**foreign address**)
- A kliens az otthonhoz kapcsolódik, onnan kapja meg az aktuális címet

Kétrétegű rendszer

Az egyes (pl. földrajzi alapon meghatározott) környékeken feljegyezzük, hogy melyik egyedek tartózkodnak éppen arrafelé.

- A névfeloldás először ezt a jegyzéket vizsgálja meg
- Ha az egyed nincsen a környéken, csak akkor kell az otthonhoz fordulni

Otthonalapú megközelítések



Otthonalapú megközelítések

Problémák

- Legalább az egyed élettartamán át fenn kell tartani az otthonot
- Az otthon helye rögzített ⇒ költséges lehet, ha az egyed messze költözik
- Rossz földrajzi skálázódás: az egyed sokkal közelebb lehet a klienshez az otthonnál

Eloszott hasítótábla

Chord eloszott hasítótábla

Elosztott hasítótáblát (distributed hash table, DHT) készítünk (konkrétan Chord protokoll szerint), ebben csúcsok tárolnak egyedeket. Az N csúcs gyűrű overlay szerkezetbe van szervezve.

- Mindegyik csúcshoz véletlenszerűen hozzárendelünk egy m bites **azonosítót**, és minden entitáshoz egy m bites **kulcsot**. (Tehát $N \leq 2^m$.)
- A k kulcsú egyed felelőse az az id azonosítójú csúcs, amelyre $k \leq id$, és nincsen köztük másik csúcs. A felelős csúcsot a kulcs **rákövetkezőjének** is szokás nevezni; jelölje $succ(k)$.

Rosszul méreteződő megoldás

A csúcsok eltárolhatnák a gyűrű következő csúcsának elérhetőségét, és így lineárisan végigkereshetnénk a gyűrűt. Ez $\mathcal{O}(N)$ hatékonyságú, rosszul skálázódik, nem hibatűrő...

DHT: Finger table

Chord alapú adattárolás

- Mindegyik p csúcs egy FT_p „finger table”-t tárol m bejegyzéssel:

$$FT_p[i] = succ(p + 2^{i-1})$$

Bináris (jellegű) keresést szeretnénk elérni, ezért minden lépés felezi a keresési tartományt: $2^{m-1}, 2^{m-2}, \dots, 2^0$.

- A k kulcsú egyed kikereséséhez (ha nem a jelenlegi csúcs tartalmazza) a kérést továbbítjuk a j indexű csúcshoz, amelyre

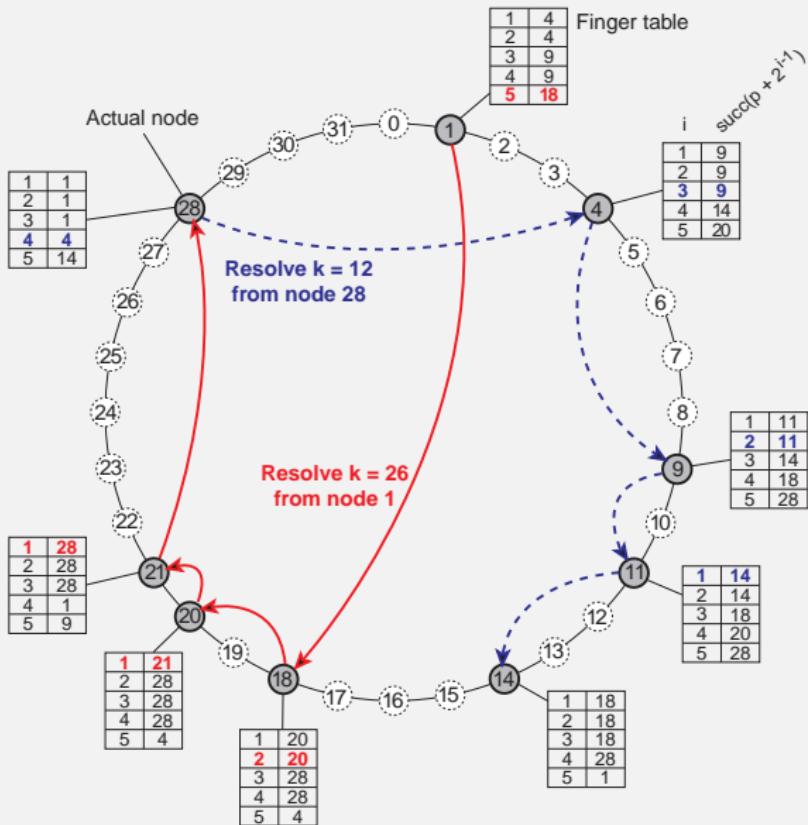
$$FT_p[j] \leq k < FT_p[j+1]$$

illetve, ha $p < k < FT_p[1]$, akkor is $FT_p[1]$ -hez irányítjuk a kérést.

Jól méreteződő megoldás

Ez a megoldás $\mathcal{O}(m)$, azaz $\mathcal{O}(\log(N))$ hatékonyságú.

DHT: Finger table



A hálózati közelség kihasználása

Probléma

Mivel overlay hálózatot használunk, az üzenetek sokat utazhatnak két csúcs között: a k és a $\text{succ}(k+1)$ csúcs messze lehetnek egymástól.

Azonosító topológia szerinti megválasztása: A csúcsok azonosítóját megpróbálhatjuk topológialag közel csúcsokhoz közelínek választani.
Ez nehéz feladat lehet.

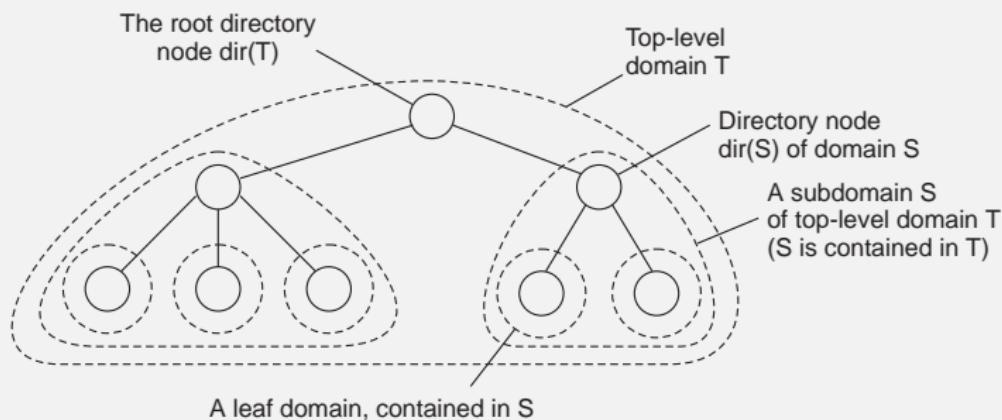
Közelség szerinti útválasztás: A p csúcs FT_p táblája m elemet tartalmaz. Ha ennél több információt is eltárolunk p -ben, akkor egy lépés megtételével közelebb juthatunk a célcímcshoz.

Szomszéd közelség szerinti megválasztása: Ha a Chordtól eltérő ábrázolást követünk, a csúcs szomszédainak megválasztásánál azok közelségét is figyelembe lehet venni.

Hierarchikus módszerek

Hierarchical Location Services (HLS)

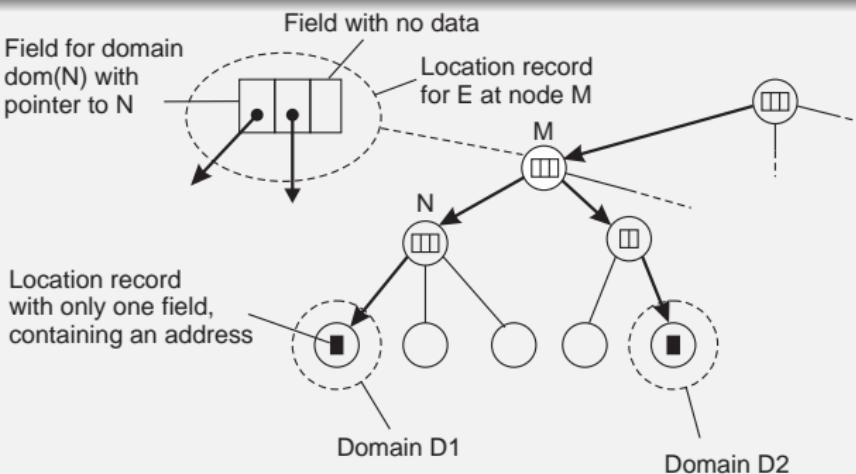
A hálózatot osszuk fel tartományokra, és minden egyik tartományhoz tartozzon egy katalógus. Építünk hierarchiát a katalógusokból.



HLS: Katalógus-csúcsok

A csúcsokban tárolt adatok

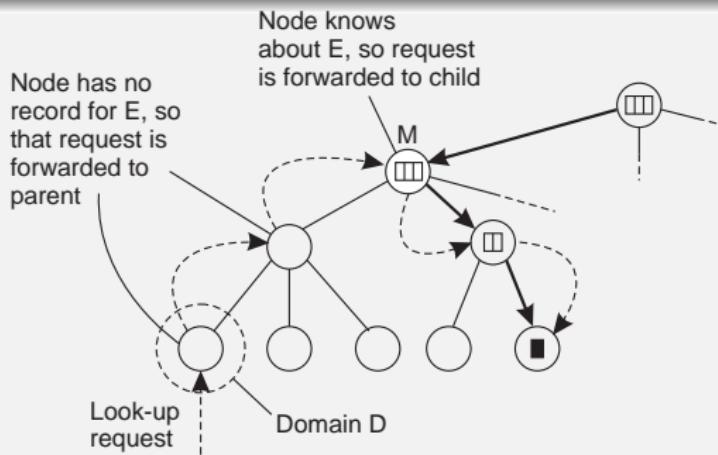
- Az E egyed címe egy levélben található meg
- A gyökértől az E leveléig vezető úton minden belső csúcsban van egy mutató a lefelé következő csúcsra az úton
- Mivel a gyökér minden út kiindulópontja, minden egyedről van információja



HLS: Keresés a fában

Keresés a fában

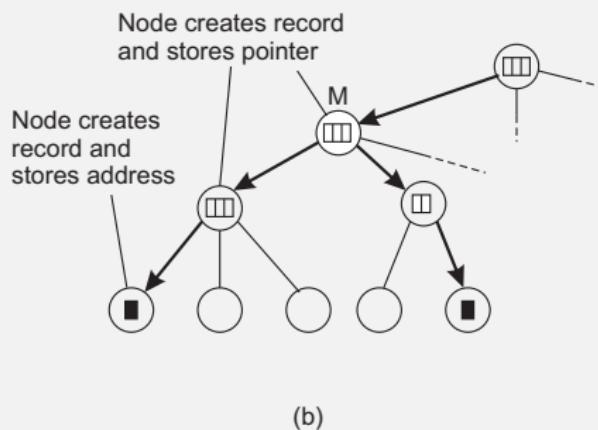
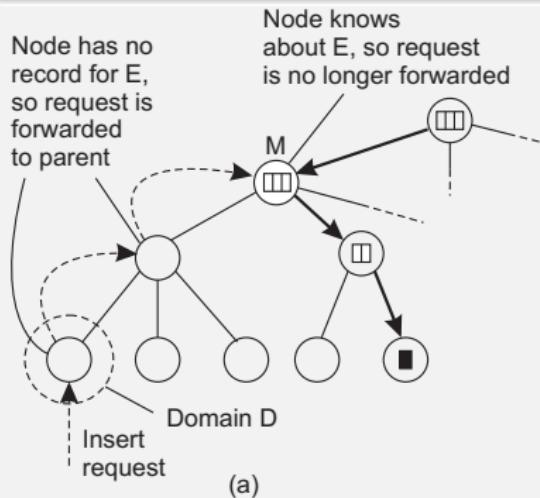
- A kliens valamelyik tartományba tartozik, innen indul a keresés
- Felmegyünk a fában addig, amíg olyan csúcshoz nem érünk, amelyik tud E -ről, aztán követjük a mutatókat a levélig, ahol megvan E címe
- Mivel a gyökér minden egyedet ismer, az algoritmus terminálása garantált



HLS: Beszúrás

Beszúrás a fában

- Ugyanaddig megyünk felfelé a fában, mint keresésnél
- Az érintett belső csúcsokba mutatókat helyezünk
- Egy csúcsban egy egyedhez több mutató is tartozhat

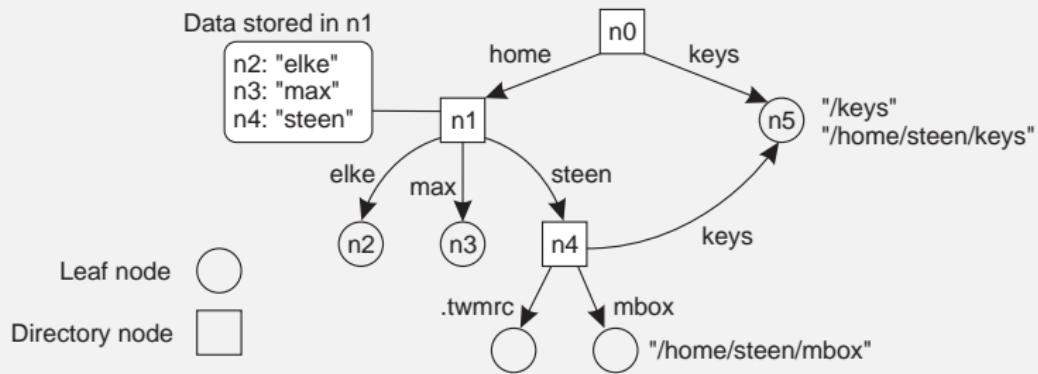


Névtér

Névtér

névtér: gyökeres, irányított, élcímkézett gráf, a levelek tartalmazzák a megnevezett egyedeket, a belső csúcsokat **katalógusnak** vagy **könyvtárnak** (directory) nevezük

Az egyedhez vezető út címkéit összeolvasva kapjuk az egyed egy nevét. A bejárt út, ha a gyökérből indul, **abszolút útvonalnév**, ha máshonnan, **relatív útvonalnév**. Mivel egy egyedhez több út is vezethet, több neve is lehet.



Névtér

Attribútumok

A csúcsokban (akár a levelekben, akár a belső csúcsokban) különféle **attribútumokat** is eltárolhatunk.

- Az egyed típusát
- Az egyed azonosítóját
- Az egyed helyét/címét
- Az egyed más neveit
- ...

Névfeloldás

Gyökér szükséges

Kiinduló csúcsra van szükségünk ahhoz, hogy megkezdhetjük a névfeloldást.

Gyökér megkeresése

A név jellegétől függő környezet biztosítja a gyökér elérhetőségét. Néhány példa név esetén a hozzá tartozó környezet:

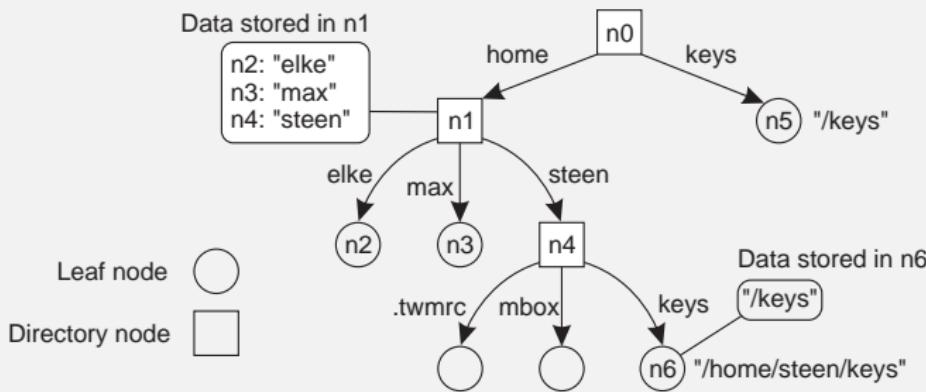
- *www.inf.elte.hu*: egy DNS névszerver
- */home/steen/mbox*: a lokális NFS fájszerver
- *0031204447784*: a telefonos hálózat
- *157.181.161.79*: a *www.inf.elte.hu* webszerverhez vezető út

Csatolás (linking)

Soft link

A gráf csúcsai valódi csatolások (hard link), ezek adják a névfeloldás alapját.

soft link: a levelek más csúcsok álneveit is tartalmazhatják. Amikor a névfeloldás ilyen csúcshoz ér, az algoritmus az álnév feloldásával folytatódik.



A névtér implementációja

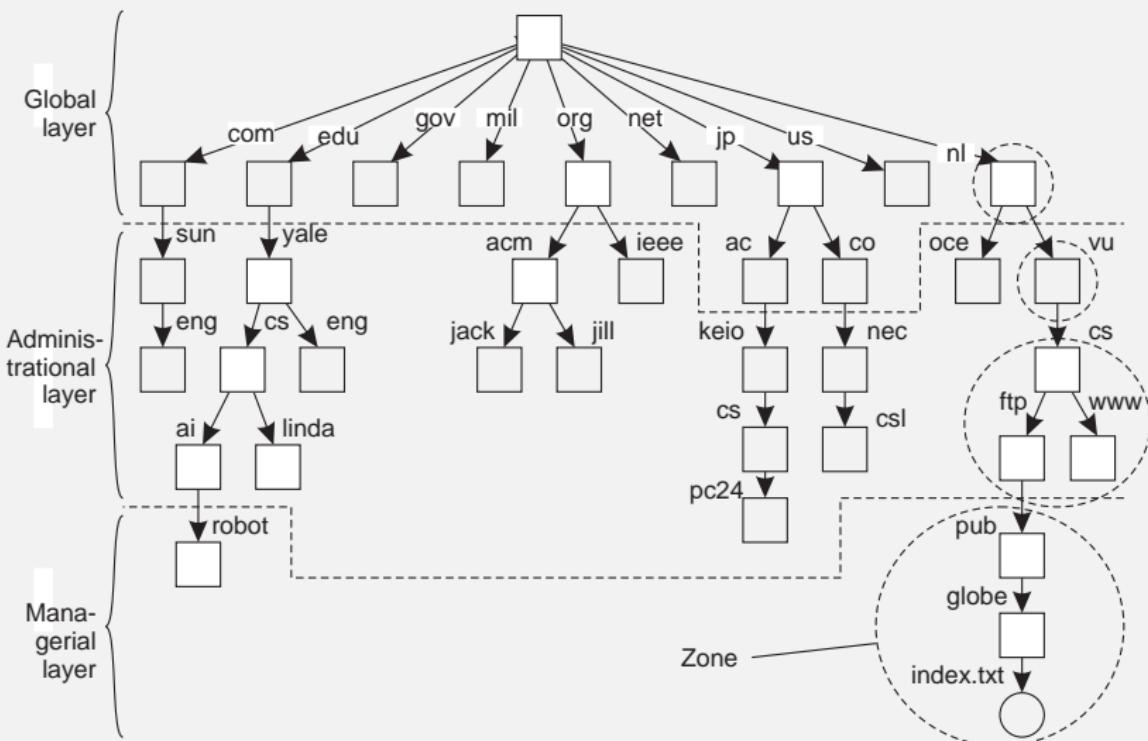
Nagyméretű névtér tagolása

Ha nagy (világméretű) névterünk van, el kell osztanunk a gráfot a gépek között, hogy hatékonytá tegyük a **névfeloldást** és a **névtér kezelését**. Ilyen nagy névteret alkot a DNS (Domain Name System).

- **Globális szint:** Gyökér és felső csúcsok. A szervezetek közösen kezelik.
- **Szervezeti szint:** Egy-egy szervezet által kezelt csúcsok szintje.
- **Kezelői szint:** Egy adott szervezeten belül kezelt csúcsok.

Szempont	Globális	Szervezeti	Kezelői
Földrajzi méret	Világméretű	Vállalati	Vállalati alegység
Csúcsok száma	Kevés	Sok	Rendkívül sok
Keresés ideje	mp.	ezredmp.	Azonnal
Frissítés terjedése	Ráérős	Azonnal	Azonnal
Másolatok száma	Sok	Nincs/kevés	Nincs
Kliens gyorsítótáraz?	Igen	Igen	Néha

A névtér implementációja: DNS



A névtér implementációja: DNS

A DNS egy csúcsában tárolt adatok

Legtöbbször az **A** rekord tartalmát kérdezzük le; a névfeloldáshoz feltétlenül szükséges az **NS** rekord.

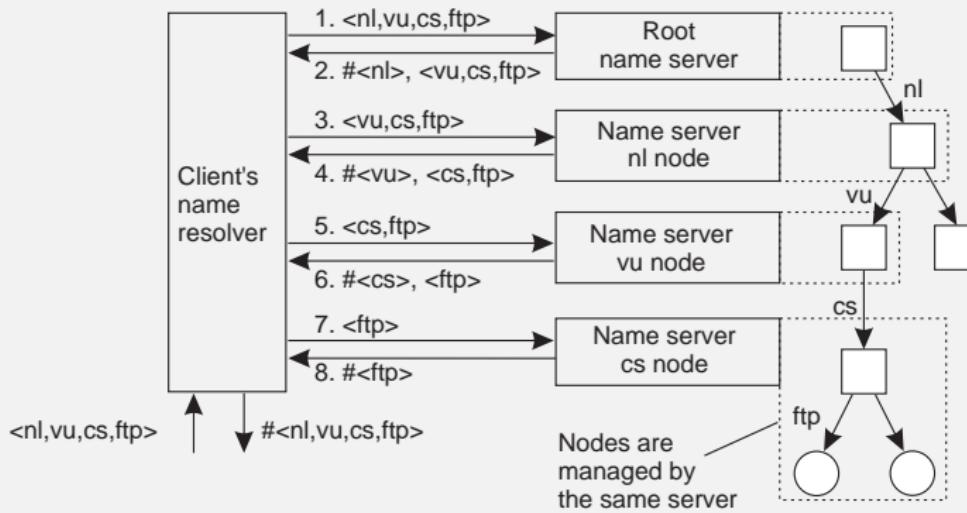
Egy **zóna** a DNS-fa egy összefüggő, adminisztratív egységként kezelt része, egy (ritkábban több) **tartomány** (domain) adatait tartalmazza.

Rekord neve	Adat jellege	Leírás
A	Gazdagép	A csomópont gazdagépének IP címe
NS	Zóna	A zóna névszervere
SOA	Zóna	A zóna paraméterei
MX	Tartomány	A csomópont levelezőszervere

Iteratív névfeloldás

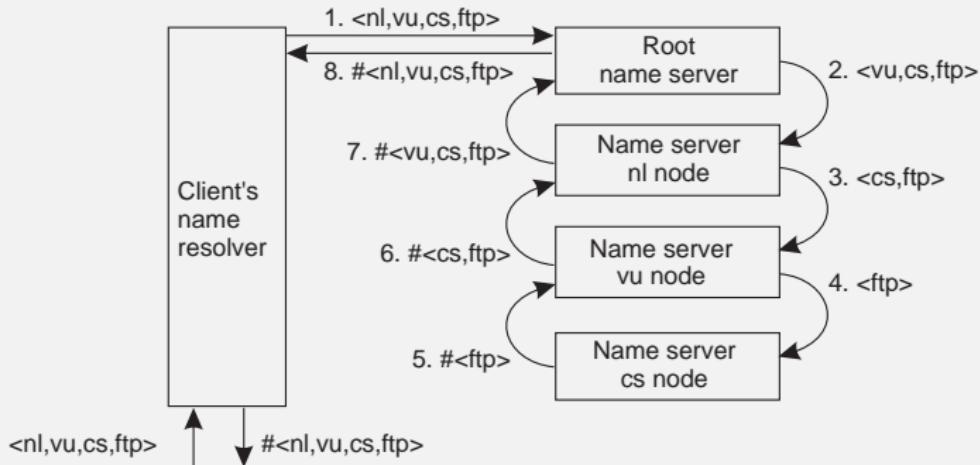
A névfeloldást a gyökér névszerverek egyikétől indítjuk.

Az iteratív névfeloldás során a névnek minden komponensét oldjuk fel, a megszólított névszerver az ehhez tartozó névszerver címét küldi vissza.



Rekurzív névfeloldás

A rekurzív névfeloldás során a névszerverek egymás közt kommunikálva oldják fel a nevet, a kliensoldali névfeloldóhoz rögtön a válasz érkezik.



Rekurzív névfeloldás: cache-elés

A névszerver ezt kezeli	Feloldandó cím	Feloldja	Átadja lefelé	Fogadja és cache-eli	Visszaadja
cs	<ftp>	#<ftp>	—	—	#<ftp>
vu	<cs,ftp>	#<cs>	<ftp>	#<ftp>	#<cs> #<cs, ftp>
nl	<vu,cs,ftp>	#<vu>	<cs,ftp>	#<cs> #<cs,ftp>	#<vu> #<vu,cs> #<vu,cs,ftp>
(gyökér)	<nl,vu,cs,ftp>	#<nl>	<vu,cs,ftp>	#<vu> #<vu,cs> #<vu,cs,ftp>	#<nl> #<nl,vu> #<nl,vu,cs> #<nl,vu,cs,ftp>

Névfeloldás: átméretezhetőség

Méret szerinti átméretezhetőség

Sok kérést kell kezelni rövid idő alatt ⇒ a globális szint szerverei nagy terhelést kapnának.

Csúcsok adatai sok névszerveren

A felső két szinten, és sokszor még az alsó szinten is ritkán változik a gráf. Ezért megtehetjük, hogy a legtöbb csúcs adatairól sok névszerveren készítünk másolatot, így a keresést várhatóan sokkal közelebbről indítjuk.

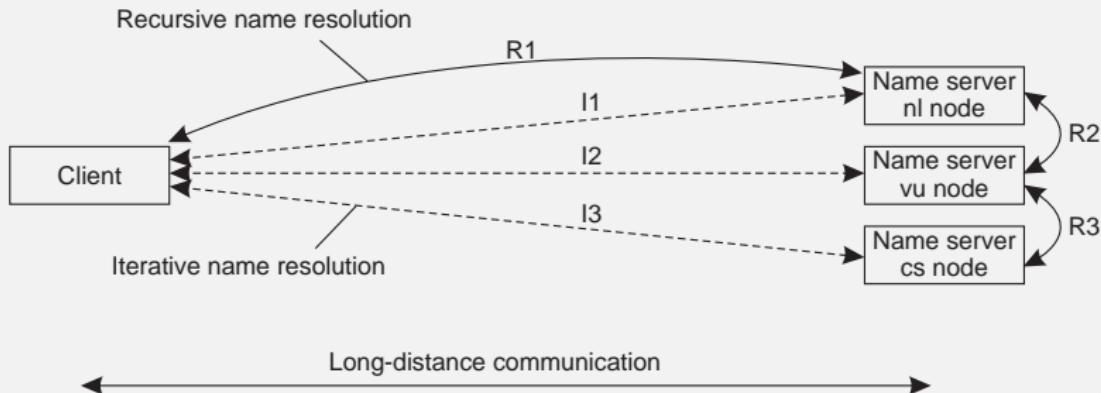
A keresett adat: az entitás címe

A legtöbbször a névfeloldással az entitás címét keressük. A névszerverek nem alkalmasak mozgó entitások címeinek kezelésére, mert azok költözésével gyakran változna a gráf.

Névfeloldás: átméretezhetőség

Földrajzi átméretezhetőség

A névfeloldásnál a földrajzi távolságokat is figyelembe kell venni.



Helyfüggés

Ha egy csúcsot egy adott névszerver szolgál ki, akkor földrajzilag oda kell kapcsolódnunk, ha el akarjuk érni a csúcsot.

Attribútumalapú nevek

Attribútumalapú keresés

Az egyedeket sokszor kényelmes lehet a tulajdonságaik (attribútumaik) alapján keresni.

Teljes általánosságban: nem hatékony

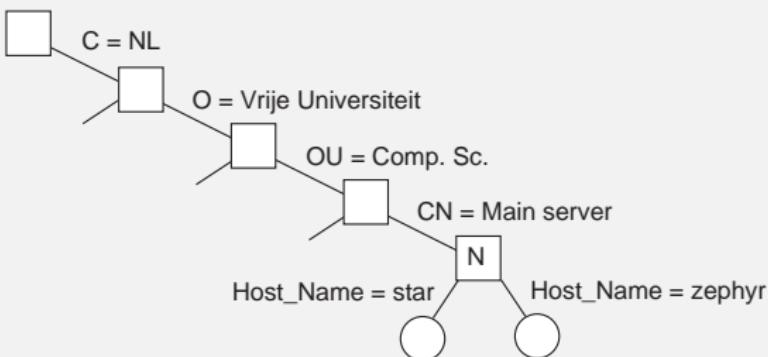
Ha bármilyen kombinációban megadhatunk attribútumértékeket, a kereséshez **az összes egyedet** érintenünk kell, ami nem hatékony.

X.500, LDAP

A **katalógusszolgáltatásokban** (directory service) az attribútumokra megkötések érvényesek. A legismertebb ilyen szabvány az **X.500**, amelyet az **LDAP** protokollon keresztül szokás elérni.

Az elnevezési rendszer fastruktúrájú, élei névalkotó jellemzőkkel (attribútum-érték párokkal) címzettek. Az egyedekre az útjuk jellemzői vonatkoznak, és további párokat is tartalmazhatnak.

Példa: LDAP



Attribute	Value
Country	NL
Locality	Amsterdam
Organization	Vrije Universiteit
OrganizationalUnit	Comp. Sc.
CommonName	Main server
Host_Name	star
Host_Address	192.31.231.42

Attribute	Value
Country	NL
Locality	Amsterdam
Organization	Vrije Universiteit
OrganizationalUnit	Comp. Sc.
CommonName	Main server
Host_Name	zephyr
Host_Address	137.37.20.10

```

answer =
search("&(C = NL) (O = Vrije Universiteit) (OU = *) (CN = Main server)")
```

Elosztott rendszerek: Alapelvek és paradigmák Distributed Systems: Principles and Paradigms

Maarten van Steen¹ Kitlei Róbert²

¹VU Amsterdam, Dept. Computer Science

²ELTE Informatikai Kar

6. rész: Szinkronizáció

2015. május 24.

Tartalomjegyzék

Fejezet
01: Bevezetés
02: Architektúrák
03: Folyamatok
04: Kommunikáció
05: Elnevezési rendszerek
06: Szinkronizáció
07: Konzisztencia & replikáció
08: Hibatűrés
10: Objektumalapú elosztott rendszerek
11: Elosztott fájlrendszerek
12: Elosztott webalapú rendszerek

Fizikai órák

Milyen módon van szükségünk az időre?

Néha a pontos időt szeretnénk tudni, néha elég, ha megállapítható két időpont közül, melyik volt korábban. Foglalkozzunk először az első kérdéssel.

Egyezményes koordinált világidő

Az időegységeket (pl. másodperc) az atomidőből (TAI) származtatjuk.

- Az atomidő definíciója a gerjesztett céziumatom által kibocsátott sugárzás frekvenciáján alapul.
- A Föld forgásának sebessége kissé változékony, ezért a világidő (UTC) néhány (szökő)másodperccel eltér az atomidőtől.
- Az atomidő  kb. 420 atomóra átlagából adódik.
- Az atomórák pontosságának nagyságrendje kb. 1 ns/nap .
- Az atomidőt műholdak sugározzák, a vétel pontossága 0.5 ms nagyságrendű, pl. az időjárás befolyásolhatja.

Fizikai órák

Fizikai idő elterjesztése

Ha a rendszerünkben van UTC-vevő, az megkapja a pontos időt. Ezt a következők figyelembe vételevel terjeszthetjük el a rendszeren belül.

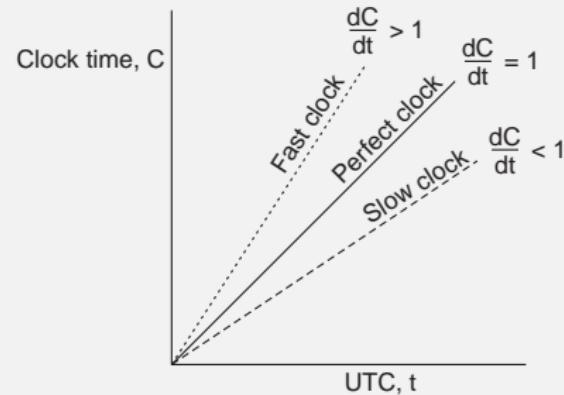
- A p gép saját órája szerint az idő t UTC-időpillanatban $C_p(t)$.
- Ideális esetben mindenkor pontos az idő: $C_p(t) = t$, másképpen $dC/dt = 1$.

Időszinkronizáció üteme

A valóságban p vagy **túl gyors**, vagy **túl lassú**, de viszonylag pontos:

$$1 - \rho \leq \frac{dC}{dt} \leq 1 + \rho$$

Ha csak megadott δ eltérést akarunk megengedni, $\delta/(2\rho)$ másodpercenként szinkronizálnunk kell az időt.



Óraszinkronizálás

Cristian-algoritmus

Mindegyik gép egy központi időszervertől kéri le a pontos időt legfeljebb $\delta/(2\rho)$ másodpercenként ([Network Time Protocol](#)).

- Nem a megkapott időre kell állítani az órát: bele kell számítani, hogy a szerver kezelte a kérést és a válasznak vissza kellett érkeznie a hálózaton keresztül.

Berkeley-algoritmus

Itt nem feltétlenül a **pontos** idő beállítása a cél, csak az, hogy minden gép ideje azonos legyen.

Az időszerver néha bekéri mindegyik gép idejét, ebből átlagot von, majd mindenkit értesít, hogy a saját óráját mennyivel kell átállítania.

- Az idő egyik gépnél sem folyhat visszafelé: ha vissza kellene állítani valamelyik órát, akkor ehelyett a számoltartott idő mérését lelassítja a gép mindaddig, amíg a kívánt idő be nem áll.

Az előbb-történt reláció

Az előbb-történt (happened-before) reláció

Az előbb-történt reláció az alábbi tulajdonságokkal rendelkező reláció. Annak a jelölése, hogy az a esemény előbb-történt-mint b -t: $a \rightarrow b$.

- Ha ugyanabban a folyamatban az a esemény korábban következett be b eseménynél, akkor $a \rightarrow b$.
- Ha a esemény egy üzenet küldése, és b esemény annak fogadása, akkor $a \rightarrow b$.
- A reláció tranzitív: ha $a \rightarrow b$ és $b \rightarrow c$, akkor $a \rightarrow c$

Parcialitás

A fenti reláció parciális rendezés: előfordulhat, hogy két esemény közül egyik sem előzi meg a másikat.

Logikai órák

Az idő és az előbb-történt reláció

Minden e eseményhez időbélyeget rendelünk, ami egy egész szám (jelölése: $C(e)$), és megköveteljük az alábbi tulajdonságokat.

- Ha $a \rightarrow b$ egy folyamat két eseményére, akkor $C(a) < C(b)$.
- Ha a esemény egy üzenet küldése és b esemény annak fogadása, akkor $C(a) < C(b)$.

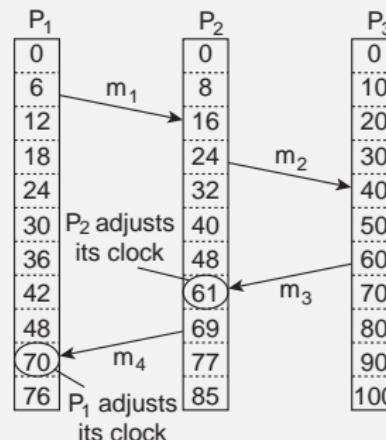
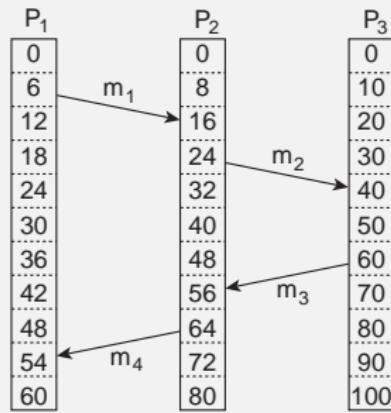
Globális óra nélkül?

Ha a rendszerben van globális óra, azzal a fenti időbélyegek elkészíthetők. A továbbiakban azt vizsgáljuk, hogyan lehet az időbélyegeket globális óra nélkül elkészíteni.

Logikai órák: Lamport-féle időbélyegek

Minden P_i folyamat saját C_i számlálót tart nyilván az alábbiak szerint:

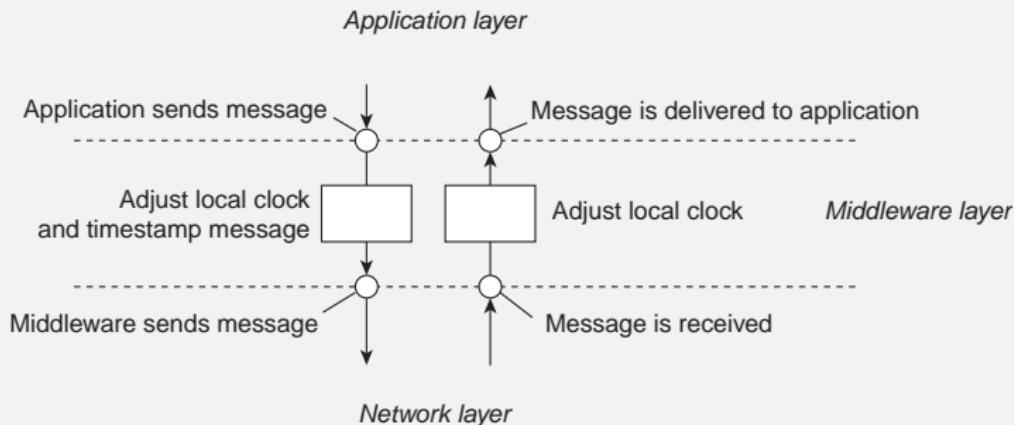
- P_i minden eseménye eggyel növeli a számlálót.
- Az elküldött m üzenetre ráírjuk az időbélyeget: $ts(m) = C_i$.
- Ha az m üzenet beérkezik P_j folyamathoz, ott a számláló új értéke $C_j = \max\{C_j, ts(m)\} + 1$ lesz; így az idő „nem folyik visszafelé”.
- P_i és P_j egybeeső időbélyegei közül tekintsük a P_i -belit elsőnek, ha $i < j$.



Logikai órák

Beállítás: köztesréteg

Az órák állítását és az üzenetek időbélyegeit a köztesréteg kezeli.

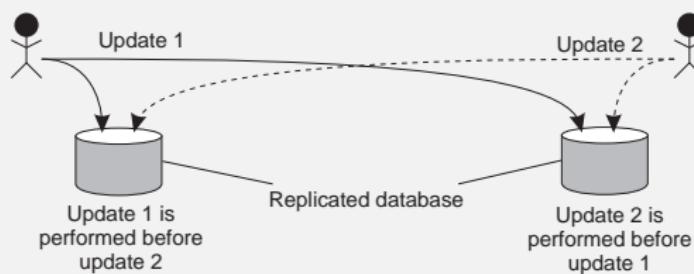


Logikai órák: példa

Pontosan sorbarendezett csoportcímzés

Ha replikált adatbázison konkurens műveleteket kell végezni, sokszor követelmény, hogy minden egyik másolaton ugyanolyan sorrendben hajtódjanak végre a műveletek.

Az alábbi példában két másolatunk van, a számlán kezdetben \$1000 van. P_1 befizet \$100-t, P_2 1% kamatot helyez el.



Probléma

Ha a műveletek szinkronizációja nem megfelelő, érvénytelen eredményt kapunk: $\text{másolat}_1 \leftarrow \1111 , de $\text{másolat}_2 \leftarrow \1110 .

Példa: Pontosan sorbarendezet csoporthoz kötött címzés

Pontosan sorbarendezet csoporthoz kötött címzés

- A P_i folyamat minden műveletet időbélyeggel ellátott üzenetben küld el. P_i egyúttal betesz a küldött üzenetet a saját $queue_i$ prioritásos sorába.
- A P_j folyamat a beérkező üzeneteket az ő $queue_j$ sorába teszi be az időbélyegnek megfelelő prioritással. Az üzenet érkezéséről minden folyamatot értesíti.

P_j akkor adja át a msg_j üzenetet feldolgozásra, ha:

- (1) msg_j a $queue_j$ elején található (azaz az ő időbélyege a legkisebb)
- (2) a $queue_j$ sorban minden P_k ($k \neq j$) folyamatnak megtalálható legalább egy üzenete, amelynek msg_j -nél későbbi az időbélyege

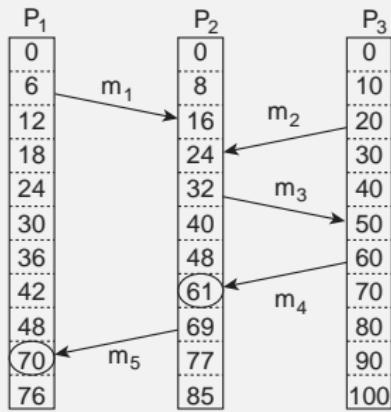
Feltételek

Feltételezzük, hogy a kommunikáció a folyamatok között **megalapozott** és **FIFO sorrendű**.

Időbélyeg-vektor

Okság

Arra is szükségünk lehet, hogy megállapíthassuk két eseményről, hogy az egyik okoz(hat)ta-e a másikat – illetve fordítva, függetlenek-e egymástól. Az eddigi megközelítésünk erre nem alkalmas: abból, hogy $C(a) < C(b)$, nem vonható le az a következtetés, hogy az a esemény **okságilag megelőzi** a b eseményt.



A példában szereplő adatok

a esemény: m_1 beérkezett $T = 16$ időbélyeggel;

b esemény: m_2 elindult $T = 20$ időbélyeggel.

Bár $16 < 20$, a és b nem függenek össze okságilag.

Időbélyeg-vektor

Időbélyeg-vektor

- A P_i most már az összes másik folyamat idejét is számon tartja egy $VC_i[1..n]$ tömbben, ahol $VC_i[j]$ azoknak a P_j folyamatban bekövetkezett eseményeknek a száma, amelyekről P_i tud.
- Az m üzenet elküldése során P_i megnöveli eggyel $VC_i[i]$ értékét (vagyis az üzenetküldés egy eseménynek számít), és a teljes V_i időbélyeg-vektort ráírja az üzenetre.
- Amikor az m üzenet megérkezik a P_j folyamathoz, amelyre a $ts(m)$ időbélyeg van írva, két dolog történik:
 - (1) $VC_j[k] := \max\{ VC_j[k], ts(m)[k] \}$
 - (2) $VC_j[j]$ megnő eggyel, vagyis az üzenet fogadása is egy eseménynek számít

Pontosan sorbarendevezett csoportcímzés

Időbélyeg-vektor alkalmazása

Az időbélyeg-vektorokkal megvalósítható a pontosan sorbarendevezett csoportcímzés: csak akkor kézbesítjük az üzeneteket, ha már minden egyik előzményüket kézbesítettük.

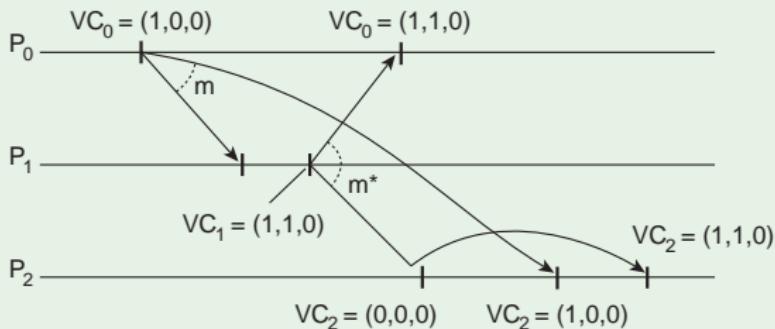
Ehhez annyit változtatunk az előbb leírt időbélyeg-vektorok működésén, hogy amikor P_j fogad egy üzenetet, akkor nem növeljük meg $VC_j[j]$ értékét.

P_j csak akkor kézbesíti az m üzenetet, amikor:

- $ts(m)[i] = VC_j[i] + 1$, azaz a P_j folyamatban P_i minden korábbi üzenetét kézbesítettük
- $ts(m)[k] \leq VC_j[k]$ for $k \neq i$, azaz az üzenet „nem a jövőből jött”

Pontosan sorbarendezeit csoportcímzés

Példa



Kölcsönös kizárási

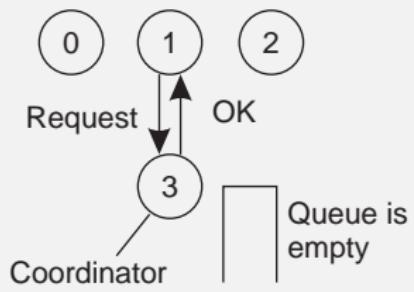
Kölcsönös kizárási: a feladat

Több folyamat egyszerre szeretne hozzáférni egy adott erőforráshoz. Ezt egyszerre csak egynek engedhetjük meg közülük, különben az erőforrás helytelen állapotba kerülhet.

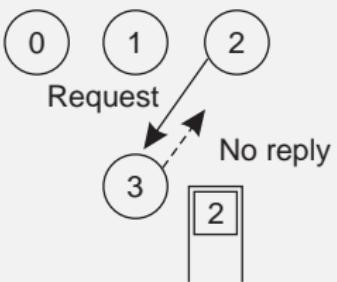
Megoldásfajták

- Központi szerver használata.
- Peer-to-peer rendszeren alapuló teljesen elosztott megoldás.
- Teljesen elosztott megoldás általános gráfszerkezetre.
- Teljesen elosztott megoldás (logikai) gyűrűben.

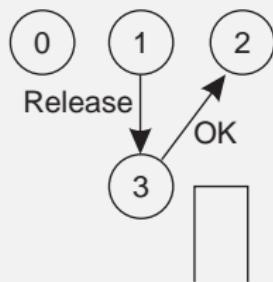
Kölcsönös kizáráás: központosított



(a)



(b)



(c)

Kölcsönös kizárás: decentralizált

Kölcsönös kizárás: decentralizált

Tegyük fel, hogy az erőforrás n -szeresen többszörözött, és minden replikátumhoz tartozik egy azt kezelő koordinátor.

Az erőforráshoz való hozzáférésről **többségi szavazás** dönt: legalább m koordinátor engedélye szükséges, ahol $m > n/2$.

Felte tesszük, hogy egy esetleges összeomlás után a koordinátor hamar felépül – azonban a kiadott engedélyeket elfelejtíti.

Példa: hatékonyúság

Tegyük fel, hogy a koordinátorok rendelkezésre állásának valószínűsége 99.9% („három kilences”), 32-szeresen replikált az erőforrásunk, és a koordinátorok háromnegyedének engedélyére van szükségünk ($m = 0.75n$).

Ekkor annak a valószínűsége, hogy túl sok koordinátor omlik össze, igen alacsony: kevesebb mint 10^{-40} .

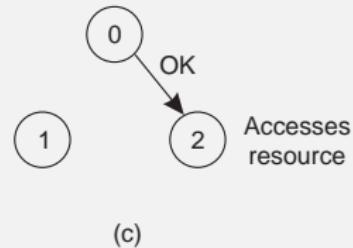
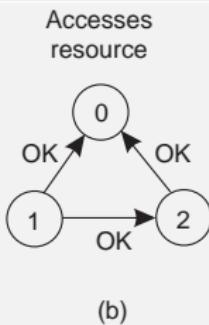
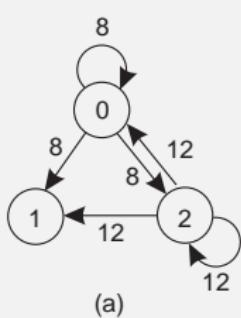
Kölcsönös kizárás: elosztott

Működési elv

Ismét többszörözőtt az erőforrás, amikor a kliens hozzá szeretne férni, kérést küld minden egyik koordinátornak (időbélyeggel).

Választ (hozzáférési engedélyt) akkor kap, ha

- a koordinátor nem igényli az erőforrást, vagy
- a koordinátor is igényli az erőforrást, de kisebb az időbélyege.
- Különben a koordinátor (átmenetileg) nem válaszol.

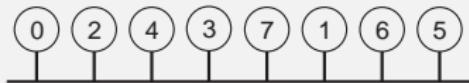


Kölcsönös kizárás: zsetongyűrű

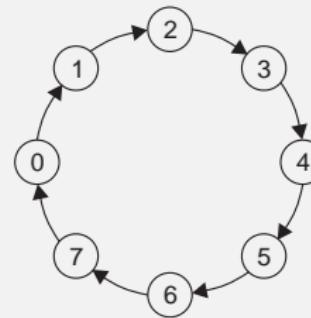
Essence

A folyamatokat logikai gyűrűbe szervezzük (fizikailag lehetnek pl. egy lokális hálózaton).

A gyűrűben egy zsetont küldünk körbe, amelyik folyamat birtokolja, az férhet hozzá az erőforráshoz.



(a)



(b)

Kölcsönös kizárás: összehasonlítás

Algoritmus	Be+kilépési üzenetszám	Belépés előtti késleltetés (üzenetidőben)	Problémák
Központosított	3	2	Ha összeomlik a koordinátor
Decentralizált	$2mk + m$	$2mk$	Kiéheztetés, rossz hatékonyság
Elosztott	$2(n - 1)$	$2(n - 1)$	Bármely folyamat összeomlása
Zsetongyűrű	$1 \dots \infty$	$0 \dots n - 1$	A zseton elvész, a birtokló folyamat összeomlik

Csúcsok globális pozícionálása

Feladat

Meg szeretnénk becsülni a csúcsok közötti kommunikációs költségeket. Erre többek között azért van szükség, hogy hatékonyan tudjuk megválasztani, melyik gépekre helyezzünk replikátumokat az adatainkból.

Ábrázolás

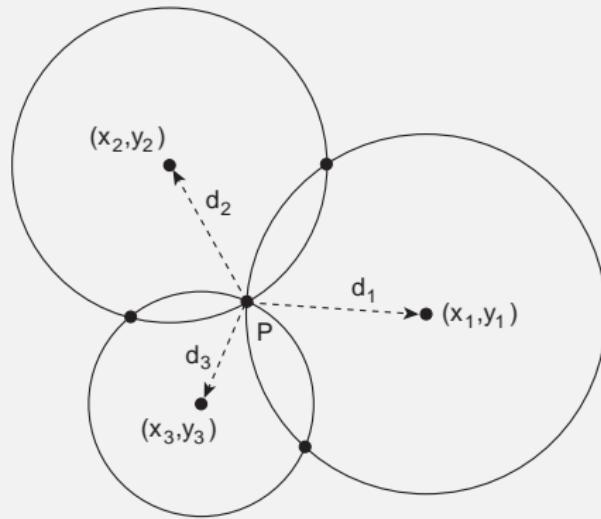
A csúcsokat egy többdimenziós geometriai térben ábrázoljuk, ahol a P és Q csúcsok közötti kommunikációs költséget a csúcsok távolsága jelöli. Így a feladatot visszavezettük távolságok becslésére.

A tér dimenziószáma minél nagyobb, annál pontosabb lesz a becslésünk, de annál költségesebb is.

A pozíció kiszámítása

A becsléshez szükséges csúcsok száma

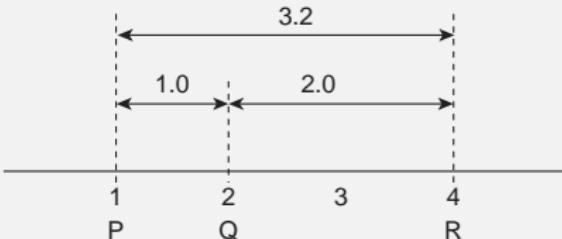
Egy pont pozíciója meghatározható a tér dimenziószámánál eggyel nagyobb számú másik pontból a tőlük vett távolságok alapján.



A pozíció kisszámítása

Nehézségek

- a késleltetések mért értékei ingadoznak
- nem egyszerűen összeadódnak a távolságok →



Megoldás

Válasszunk L darab csúcsot, amelyek pozícióját tegyük fel, hogy nagyon pontosan meghatároztuk.

Egy P csúcsot ezekhez viszonyítva helyezünk el: megmérjük az összeszűrőt mért késleltetését, majd úgy választjuk meg P pozícióját, hogy az össz-hiba (a mért késleltetések és a megválasztott pozícióból geometriailag adódó késleltetés eltérése) a legkisebb legyen.

Vezetőválasztás: zsarnok-algoritmus

Vezetőválasztás: feladat

Sok algoritmusnak szüksége van arra, hogy kijelöljön egy folyamatot, amely aztán a további lépéseket koordinálja.

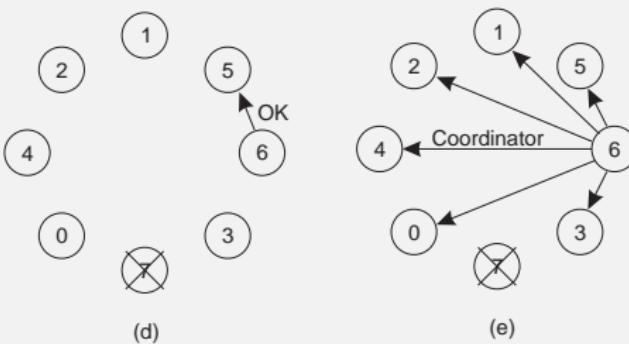
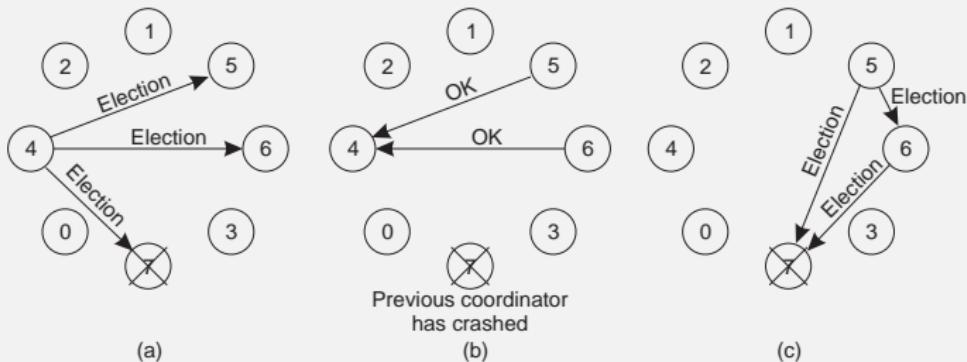
Ezt a folyamatot **dinamikusan** szeretnénk kiválasztani.

Zsarnok-algoritmus

A folyamatoknak sorszámot adunk. A legnagyobb sorszámú folyamatot szeretnénk vezetőnek választani.

- Bármelyik folyamat kezdeményezhet vezetőválasztást. Mindegyik folyamatnak (amelyről nem ismert, hogy kisebb lenne a küldőnél a sorszáma) elküld egy választási üzenetet.
- Ha P_{nagyobb} üzenetet kap P_{kisebb} -től, visszaküld neki egy olyan üzenetet, amellyel kiveszi P_{kisebb} -et a választásból.
- Ha P megadott időn belül nem kap letiltó üzenetet, ő lesz a vezető. Erről minden másik folyamatot értesíti egy üzenettel.

Zsarnok-algoritmus



Vezetőválasztás gyűrűben

Vezetőválasztás gyűrűben

Ismét logikai gyűrűnk van, és a folyamatoknak vannak sorszámai. A legnagyobb sorszámú folyamatot szeretnénk vezetőnek választani.

Bármelyik folyamat kezdeményezhet vezetőválasztást: elindít egy üzenetet a gyűrűn körbe, amelyre mindenki ráírja a sorszámát. Ha egy folyamat összeomlott, az kimarad az üzenetküldés menetéből.

Amikor az üzenet visszajut a kezdeményezőhöz, minden aktív folyamat sorszáma szerepel rajta. Ezek közül a legnagyobb lesz a vezető; ezt egy másik üzenet körbeküldése tudatja mindenkel.

Nem okozhat problémát, ha több folyamat is egyszerre kezdeményez választást, mert ugyanaz az eredmény adódik. Ha pedig az üzenetek valahol elvesznének (összeomlik az éppen őket tároló folyamat), akkor újrakezdhető a választás.

Superpeer-választás

Szempontok

A superpeer-eket úgy szeretnénk megválasztani, hogy teljesüljön rájuk:

- A többi csúcs alacsony késleltetéssel éri el őket
- Egyenletesen vannak elosztva a hálózaton
- A csúcsok megadott hányadát választjuk superpeer-nek
- Egy superpeer korlátozott számú peer-t szolgál ki

Megvalósítás DHT használata esetén

Az azonosítók terének egy részét fenntartjuk a superpeer-ek számára.

Példa: ha m - bites azonosítókat használunk, és S superpeer-re van szükségünk, a $k = \lceil \log_2 S \rceil$ felső bitet foglaljuk le a superpeer-ek számára. Így N csúcs esetén kb. $2^{k-m}N$ darab superpeer lesz.

A p kulcshoz tartozó superpeer: a p AND $\underbrace{11\cdots 1}_{k} \underbrace{00\cdots 0}_{m-k}$ kulcs felelőse az.

Elosztott rendszerek: Alapelvek és paradigmák

Distributed Systems: Principles and Paradigms

Maarten van Steen¹ Kitlei Róbert²

¹VU Amsterdam, Dept. Computer Science

²ELTE Informatikai Kar

7. rész: Konzisztencia & replikáció

2015. május 24.

Tartalomjegyzék

Fejezet
01: Bevezetés
02: Architektúrák
03: Folyamatok
04: Kommunikáció
05: Elnevezési rendszerek
06: Szinkronizáció
07: Konzisztencia & replikáció
08: Hibatűrés
10: Objektumalapú elosztott rendszerek
11: Elosztott fájlrendszerek
12: Elosztott webalapú rendszerek

Hatékonyúság és átméretezhetőség

Konfliktusos műveletek

A replikátumok konzisztensen tartásához garantálni kell, hogy az egymással konfliktusba kerül(het)ő műveletek minden replikátumon egyforma sorrendben futnak le.

Ahogy a tranzakcióknál, írás–olvasás és írás–írás konfliktusok fordulhatnak elő.

Terv: kevesebb szinkronizáció

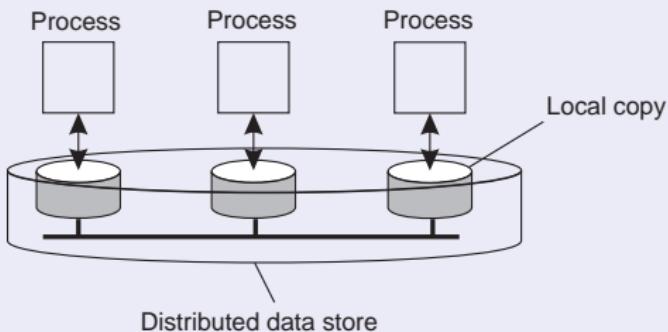
Az összes konfliktusos művelet globális sorbarendezése általában túl költséges.

Megvizsgáljuk, hogyan lehet a konzisztenciakövetelményeket gyengíteni. Jellemzően minél megengedőbbek a feltételek, annál kevesebb szinkronizáció szükséges a biztosításukhoz.

Adatközpontú konzisztencia

Konzisztenciamodell

A **konzisztenciamodell** megszabja, milyen módokon használhatják a folyamatok az adatbázist. **Elosztott adattár** (lásd az ábrát) esetén legfőképpen az egyidejű írási és olvasási műveletekre ad előírásokat. Ha a feltételek teljesülnek, az adattárat **érvényesnek** tekintjük.



Folyamatos konzisztencia

Konzisztencia mértéke

A konzisztencia több módon is sérülhet:

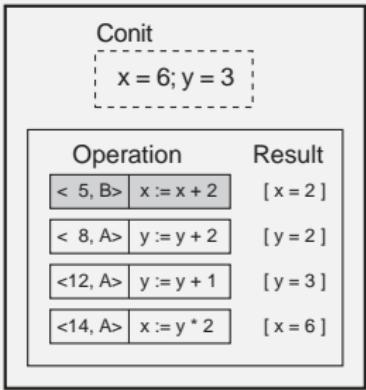
- eltérhet a replikátumok számértéke
- eltérhet, hogy mennyire frissek az egyes replikátumok
- eltérhet, hogy hány frissítési művelet nem történt még meg (illetve: sorrendben melyik műveletek hiányoznak)

Conit

Ha megtehetjük, a konzisztenciafeltéleket nem a teljes adatbázisra írjuk fel, hanem az adatoknak minél szűkebb körére. Az olyan adategység, amelyre közös feltételrendszer vonatkozik, a **conit** (**consistency unit**).

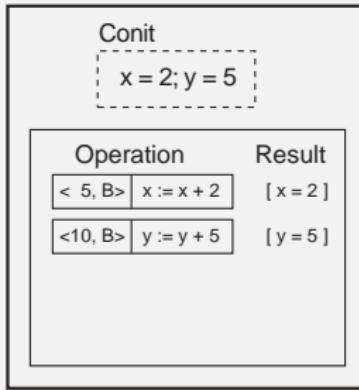
Példa: Conit

Replica A



Vector clock A = (15, 5)
 Order deviation = 3
 Numerical deviation = (1, 5)

Replica B



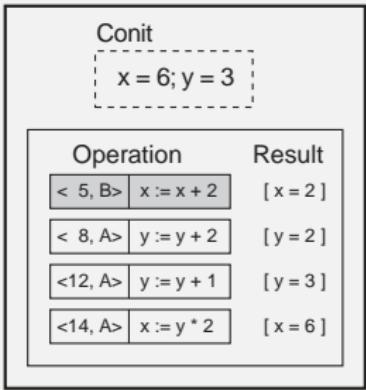
Vector clock B = (0, 11)
 Order deviation = 2
 Numerical deviation = (3, 6)

Conit (most az x és y változó alkotja)

- Mindkét replikáumnak van vektorórája: (A-beli idő, B-beli idő)
- B elküldi A-nak a [$5, B$]: $x := x + 2$ műveletet
- A véglegesíti a kijött eredményt (nem vonható vissza)

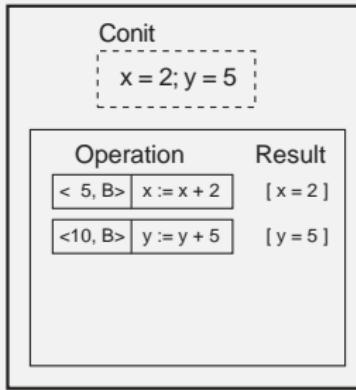
Példa: Conit

Replica A



Vector clock A $= (15, 5)$
 Order deviation $= 3$
 Numerical deviation $= (1, 5)$

Replica B



Vector clock B $= (0, 11)$
 Order deviation $= 2$
 Numerical deviation $= (3, 6)$

Conit (most az x és y változó alkotja)

- A három műveletet nem végzett még el
- A-ban egy B-beli művelet még nem hajtódott végre, a számérték legfeljebb 5-tel térhet el $\Rightarrow (1, 5)$

Soros konzisztencia

Jelölések

Sokszor a feltételeket nem a számértékekre alapozzuk, hanem csupán az írások/olvasások tényére. Jelölje $W(x)$ azt, hogy az x változót írta egy megadott folyamat, $R(x)$ azt, hogy olvasta, a mellettük levő betűk pedig azt jelölik, hogy az olvasás melyik írással írt értéket látja.

Soros konzisztencia

Soros konzisztencia esetén azt várjuk el, hogy a végrehajtás eredménye olyan legyen, mintha az összes folyamat összes művelete egy meghatározott sorrendben történt volna meg, megőrizve bármely adott folyamat saját műveleteinek sorrendjét. **(a) teljesíti, (b) nem**

P1: $W(x)a$

P2: $W(x)b$

P3: $R(x)b$ $R(x)a$

P4: $R(x)b$ $R(x)a$

(a)

P1: $W(x)a$

P2: $W(x)b$

P3: $R(x)b$ $R(x)a$

P4: $R(x)a$ $R(x)b$

(b)

Okozati konzisztencia

Okozati konzisztencia

A potenciálisan okozatilag összefüggő műveleteket kell minden egyik folyamatnak azonos sorrendben látnia. A konkurens írásokat különböző folyamatok különböző sorrendben láthatják.

(b) teljesíti; (a) nem, mert ott P_1 és P_2 írásait „összeköt” az olvasás

P1: $W(x)a$

P2: $R(x)a$ $W(x)b$

P3: $R(x)b$ $R(x)a$

P4: $R(x)a$ $R(x)b$

(a)

P1: $W(x)a$

P2: $W(x)b$

P3: $R(x)b$ $R(x)a$

P4: $R(x)a$ $R(x)b$

(b)

Műveletek csoportosítása

Szinkronizáció

Most **szinkronizációs változókat** használunk, ezek elérései sorosan konzisztensek. Hárromfajta megközelítés:

- **Egy rendszerszintű** S változó használata. S egy elérése után garantált, hogy a korábbi elérései előtti írások megtörténtek.
- **Igényeljük** (acquire) és aztán **feloldjuk** (release) a változókat, ezzel kritikus területeket (más néven: védett területeket) alakítunk ki.
 - Több rendszerszintű szinkronizációs változó használata.
A védelem a területen írt/olvasott adatokra terjed ki.
 - Minden adatelemhez külön változó használata.
A védelem a változó adatalemére terjed ki.

Ha a fenti szinkronizációs határokon belül több írás is történik, a határon belül ezek nem definiált sorrendben látszanak, és a szinkronizáció után csak a végeredmény látszik, az nem, hogy milyen sorrendben történtek az írások.

Kliensközpontú konzisztencia

Cél

Azt helyezzük most előtérbe, hogy a szervereken tárolt adatok hogyan látszanak egy adott kliens számára. A kliens mozog: különböző szerverekhez csatlakozik, és írási/olvasási műveleteket hajt végre.

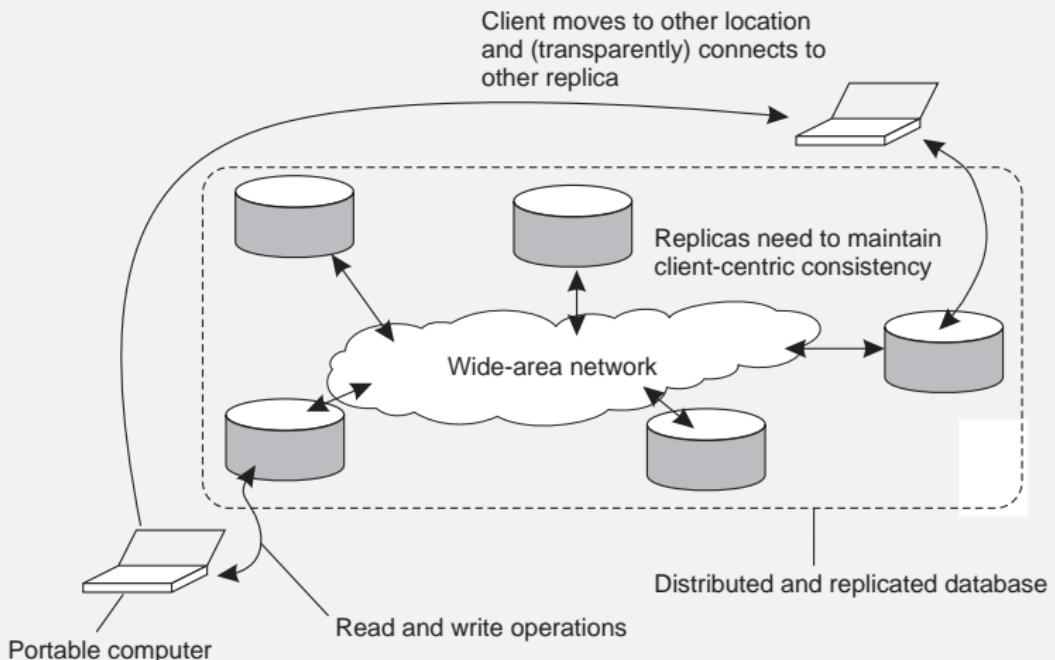
Az *A* szerver után a *B* szerverhez csatlakozva különböző problémák léphetnek fel:

- az *A*-ra feltöltött frissítések lehet, hogy még nem jutottak el *B*-hez
- *B*-n lehet, hogy újabb adatok találhatóak meg, mint *A*-n
- a *B*-re feltöltött frissítések ütközhetnek az *A*-ra feltöltöttekkel

Cél: a kliens azokat az adatokat, amelyeket az *A* szerveren kezelt, ugyanolyan állapotban lássa *B*-n. Ekkor az adatbázis konzisztensnek látszik a kliens számára.

Négyfajta kombináció: a kliens *A*-n (olvasott/írt) adatokat (olvas/ír) *B*-n.

Kliensközpontú konzisztencia



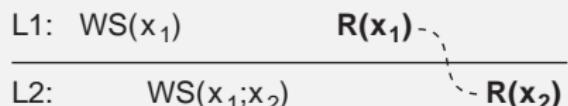
Kliensközpontú konzisztencia: olvasás után olvasás

Jelölések

- $WS(x_i)$: írási műveletek összessége (write set) az x változóra
- $WS(x_i; x_j)$: olyan $WS(x_j)$, amely tartalmazza $WS(x_i)$ frissítéseit

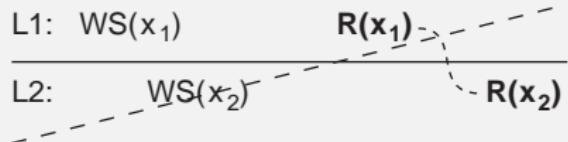
Monoton olvasás (olvasás után olvasás)

Ha egyszer a kliens kiolvasott egy értéket x -ből, minden ezután következő olvasás ezt adja, vagy ennél frissebb értéket.



Példa: határidőnapló

Minden korábbi bejegyzésünknek meg kell lennie az új szerveren is.



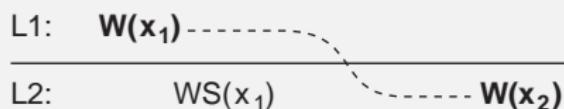
Példa: levelezőkliens

Minden korábban letöltött levelünknek meg kell lennie az új szerveren is.

Kliensközpontú konzisztencia: írás után írás

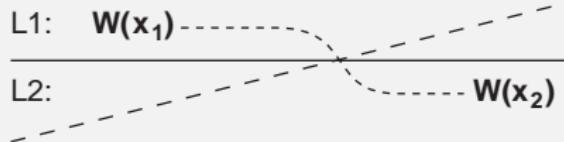
Monoton írás (írás után írás)

A kliens csak akkor írhatja x -et, ha a kliens korábbi írásai x -re már befejeződtek.



Példa: forráskód

A kliens egy programot szerkeszt több szerveren. minden korábbi szerkesztésnek jelen kell lennie, mert az újonnan írt kód függ tőlük.



Példa: verziókezelés

A kliens verziószámokkal ellátott fájlokat készít. minden korábbi verziónak meg kell lennie ahhoz, hogy újat készíthessen.

Kliensközpontú konzisztencia: írás után olvasás

Olvasd az írásodat (írás után olvasás)

Ha a kliens olvassa x -et, a saját legutolsó írásának eredményét kapja (vagy frissebbet).

L1: $W(x_1)$ -----
----- ----- -----
L2: $WS(x_1; x_2)$ ----- $R(x_2)$

L1: $W(x_1)$ -----
----- ----- -----
L2: $WS(x_2)$ ----- $R(x_2)$

Példa: honlap szerkesztése

A kliens a honlapját szerkeszti, majd megnézi a szerkesztés eredményét. Ahelyett, hogy a böngésző gyorsítótárból egy régebbi verzió kerülne elő, a legfrissebb változatot szeretné látni.

Kliensközpontú konzisztencia: olvasás után írás

Írás olvasás után (olvasás után írás)

Ha a kliens kiolvasott egy értéket x -ből, minden ezután kiadott frissítési művelete x legalább ennyire friss értékét módosítja.

L1: $WS(x_1)$ $R(x_1)$ - - -

L2: $WS(x_1; x_2)$ - - - $W(x_2)$

L1: $WS(x_1)$ $R(x_1)$ - - -

L2: - - - $WS(x_2)$ - - - $W(x_3)$

Példa: fórum

A kliens csak olyan hírré tud választ beküldeni, amelyet már látott.

Replikátumszerverek elhelyezése

Tegyük fel, hogy N lehetséges helyre szeretnénk összesen K darab szervert telepíteni. Melyik helyeket válasszuk?

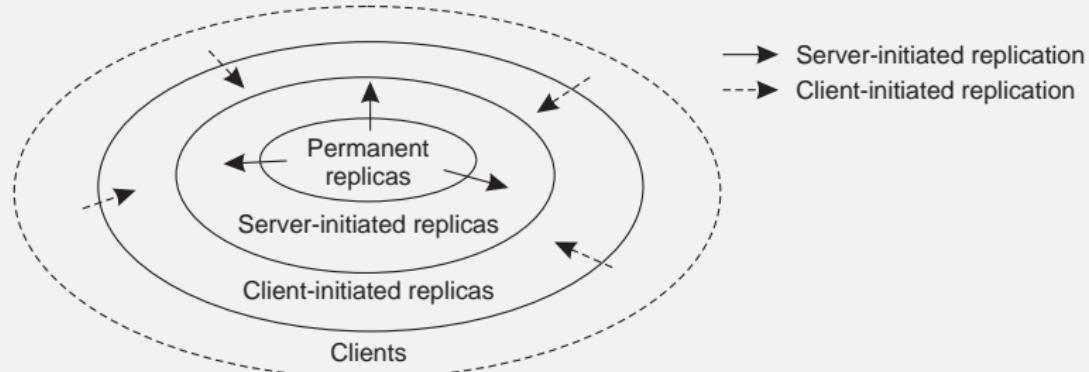
- Tegyük fel, hogy ismert a kliensek helye. Válasszuk meg úgy a szervereket, hogy azok átlagtávolsága a kliensektől minimális legyen. **Egzakt kiszámítása költséges, heurisztika szükséges.**
- Jellemzően a kliensek több autonóm rendszerben^a találhatóak meg. A K legnagyobb rendszerben helyezzünk el egy-egy szervert, minden a rendszeren belül leginkább központi helyre. **Szintén magas a számítási költsége.**
- Mint korábban a csúcsok globális pozicionálásánál, ábrázoljuk a csúcsokat egy d -dimenziós térben, ahol a távolság mutatja a késleltetést. Keressük meg a K „legsűrűbb” részt, és oda helyezzünk szervereket. **Számítási költsége alacsonyabb.**

^aA teljes hálózat útválasztás szempontjából egységként adminisztrált része, gyakran egy Internet-szolgáltató (ISP) felügyeli.

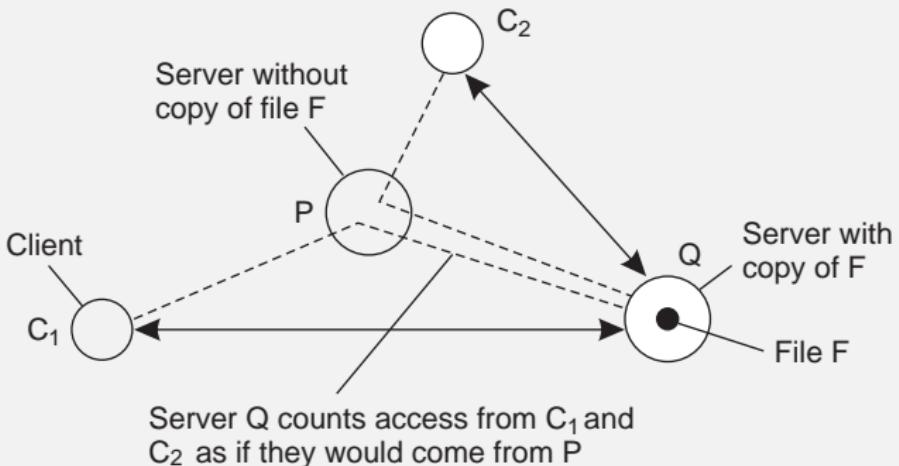
Tartalom replikálása

Különböző jellegű folyamatok tárolhatják az adatokat:

- **Tartós másolat:** a folyamat minden rendelkezik másolattal: eredetszerver (origin server)
- **Szerver által kezdeményezett másolat:** replikátum kihelyezése egy szerverre, amikor az igényli az adatot
- **Kliens által kezdeményezett másolat:** kliensoldali gyorsítótár



Szerver által kezdeményezett másolatok



A rendszer figyeli, hányszor fértek hozzá a fájlhoz (A), úgy számolva, mintha a kérés a klienshez legközelebbi szerverhez (P) érkezett volna be. Adott két szám, D és R , melyekre $D < R$. **Mit tegyük a fájlla?**

- Ha $A < D \Rightarrow$ **töröljük Q -ról** (ha megvan máshol a rendszerben)
- Ha $D < A < R \Rightarrow$ **migráljuk Q -ról P -re**
- Ha $R < A \Rightarrow$ **replikáljuk P -re**

Frissítés terjesztése

Modellek

Megváltozott tartalmat több különböző módon lehet szerver-kliens architektúrában átadni:

- Kizárolag a frissítésről szóló **értesítés/érvénytelenítés** elterjesztése (pl. gyorsítótárknál ez egy egyszerű lehetőség)
- **Passzív replikáció:** adatok átvitеле egyik másolatról a másikra
- **Aktív replikáció:** frissítési művelet átvitеле

Melyiket érdemes választani?

A sávszélesség és az írási/olvasási műveletek aránya a replikátumon nagyban befolyásolja, melyik módszer a leghatékonyabb adott esetben.

Frissítés terjesztése

- **Küldésalapú:** a szerver a kliens kérése nélkül küldi a frissítést
- **Rendelésalapú:** a kliens kérvényezi a frissítést

Témakör	Küldésalapú	Rendelésalapú
Kezdeményező	Szerver	Kliens
Szerverállapot	Klienscache-ek listája	Nincsen
Küldött üzenetek	Frissítés*	Lekérdezés és frissítés
Válaszidő a kliensnél	Azonnali*	Letöltő frissítés ideje

*: A kliens később még alkalmazhat letöltő frissítést (fetch update).

Frissítés terjesztése

Haszonbérlet

Haszonbérlet (lease): a szerver ígéretet tesz a kliensnek, hogy át elküldi neki a frissítéseket, amíg a haszonbérlet aktív

Rugalmas haszonbérlet

Fix lejárat helyett rugalmasabb, ha a rendszer állapotától függhet a haszonbérlet időtartama:

- **Kor szerinti**: minél régebben változott egy objektum, annál valószínűbb, hogy sokáig változatlan is marad, ezért hosszabb lejárat adható
- **Igénylés gyakorisága szerinti**: minél gyakrabban igényli a kliens az objektumot, annál hosszabb időtartamokra kap haszonbérletet rá
- **Terhelés szerinti**: minél nagyobb a szerver terhelése, annál rövidebb haszonbérleteket ad ki

Folyamatos konzisztencia: számszerű eltérések

Alapelv

- Az egyszerűség kedvéért most egyetlen adatelemet vizsgálunk.
- Az S_i szerver a $\log(S_i)$ naplóba írja az általa végrehajtott műveleteket.
- A W írási műveletet elsőként végrehajtó szervert jelölje $origin(W)$, a művelet hatására bekövetkező értékváltozást pedig $weight(W)$. Tegyük fel, hogy ez minden pozitív szám.
- S_i olyan írásait, amelyek S_j -ről származnak, jelölje $TW[i,j]$:

$$TW[i,j] = \sum \{ weight(W) \mid origin(W) = S_j \text{ \& } W \in \log(S_i) \}$$

- Ekkor a változó összértéke (v) és értéke az i -edik másolaton (v_i):

$$v = v_{kezdeti} + \sum_k TW[k,k]$$

$$v_i = v_{kezdeti} + \sum_k TW[i,k]$$

Folyamatos konzisztencia: számszerű eltérések

Cél: minden S_i szerveren teljesüljön: $v - v_i < \delta_i$ (rögzített δ_i értékre).

Algoritmus

$TW[i, j]$ értékét minden S_k szerver becsülje egy $TW_k[i, j]$ értékkel.
(Azaz: S_k „mit tud”, S_i milyen frissítéseket kapott már meg S_j -től.)

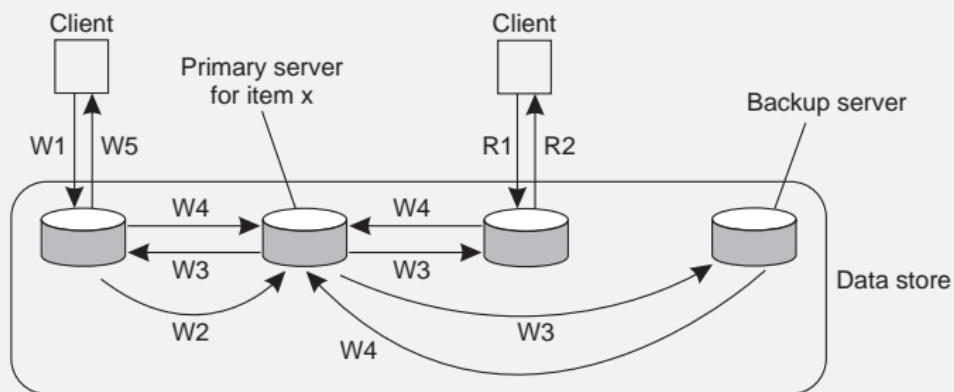
Mivel a számértékeink nemnegatívak, fennáll:

$$0 \leq TW_k[i, j] \leq TW[i, j] \leq TW[j, j] = TW_j[j, j]$$

- Amikor új frissítési művelet érkezik be egy szerverre, pletykálással értesíti erről a többi szervert.
- Amikor S_j azt látja, hogy $TW_j[i, j]$ túl messzire kerül $TW[j, j]$ -től^a, akkor küldje el S_i -nek $\log(S_j)$ műveleteit.

^a $TW[j, j] - TW_j[i, j] > \delta_i / (N - 1)$, ahol N a replikátumok száma

Elsődleges másolaton alapuló protokoll távoli írással

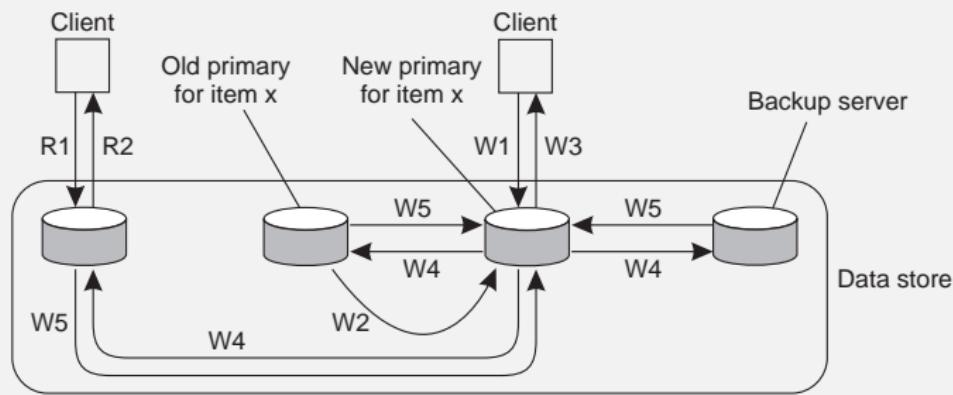


- W1. Write request
- W2. Forward request to primary
- W3. Tell backups to update
- W4. Acknowledge update
- W5. Acknowledge write completed

- R1. Read request
- R2. Response to read

Példa: nagy hibatűrést igénylő elosztott adatbázisokban és fájlrendszerben használatos. A másolatok gyakran egy lokális hálózatra kerülnek.

Elsődleges másolaton alapuló protokoll helyi írással



W1. Write request

W2. Move item x to new primary

W3. Acknowledge write completed

W4. Tell backups to update

W5. Acknowledge update

R1. Read request

R2. Response to read

Példa: kapcsolat nélküli munka, időnként szinkronizálás a rendszerrel annak különböző pontjaihoz csatlakozva.

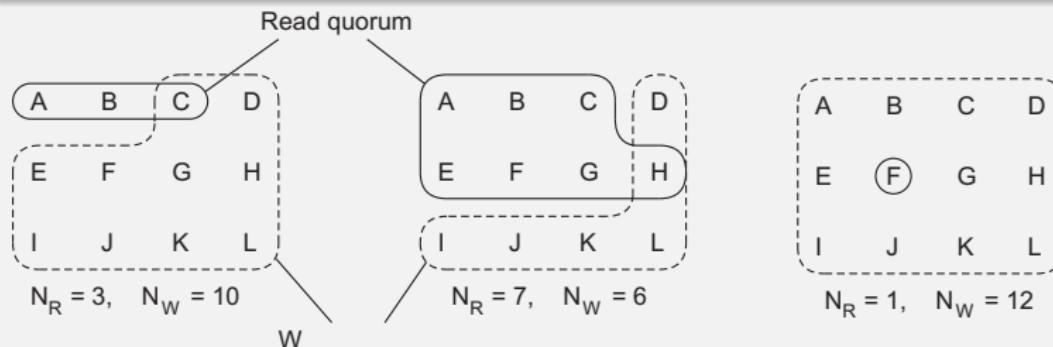
Többszörözöttírás-protokoll

Testületalapú protokoll

Többszörözött írás: az írási műveletet több szerveren hajtjuk végre.

Testület (quorum): egy művelet végrehajtása előtt meghatározott számú szervertől kell engedélyt kérni. Jelölés: írási N_W , olvasási N_R .

Egy írási művelet ütközhetne egy olvasási művelettel, vagy egy másik írásival; az első elkerüléséhez $N_R + N_W > N$, a másodikhoz $N_W + N_W > N$, azaz $N_W > N/2$ szükséges a skatulya-elv alapján.



Elosztott rendszerek: Alapelvek és paradigmák Distributed Systems: Principles and Paradigms

Maarten van Steen¹ Kitlei Róbert²

¹VU Amsterdam, Dept. Computer Science

²ELTE Informatikai Kar

8. rész: Hibatűrés

2015. május 24.

Tartalomjegyzék

Fejezet
01: Bevezetés
02: Architektúrák
03: Folyamatok
04: Kommunikáció
05: Elnevezési rendszerek
06: Szinkronizáció
07: Konzisztencia & replikáció
08: Hibatűrés
10: Objektumalapú elosztott rendszerek
11: Elosztott fájlrendszerek
12: Elosztott webalapú rendszerek

Megbízhatóság

Alapok

A komponensek feladata, hogy a kliensek számára szolgáltatásokat tesz elérhetővé. Ehhez sokszor maga is szolgáltatásokat vesz igénybe más komponensektől ⇒ függ tőlük.

A függésekkel most főleg abból a szempontból vizsgáljuk, hogy a hivatkozott komponensek helyessége kihat a hivatkozó helyességére.

Elérhetőség

A komponens reagál a megkeresésre

Megbízhatóság

A komponens biztosítja a szolgáltatást

Biztonságosság

A komponens ritkán romlik el

Karbantarthatóság

Az elromlott komponens könnyen javítható

Terminológia

Hasonló nevű, de különböző fogalmak

- **Hibajelenség** (failure): a komponens nem a tőle elvártaknak megfelelően üzemel
- **Hiba** (error): olyan rendszerállapot, amely hibajelenséghez vezethet
- **Hibaok** (fault): a hiba (feltételezett) oka

Hibákkal kapcsolatos tennivalók

- **Megelőzés**
- **Hibatűrés**: a komponens legyen képes **elfedni** a hibákat
- **Mérséklés**: lehet mérsékelni a hibák kiterjedését, számát, súlyosságát
- **Előrejelzés**: előre becsülhető lehet a hibák száma, jövőbeli előfordulása, következményei

Lehetséges hibaokok

Lehetséges hibaokok

A hibáknak sok oka lehet, többek között az alábbiak.

- **Összeomlás** (crash): a komponens leáll, de előtte helyesen működik
- **Kiesés** (omission): a komponens nem válaszol
- **Időzítési hiba** (timing): a komponens helyes választ küld, de túl későn (ennek **teljesítménnyel** kapcsolatos oka lehet: a komponens túl lassú)
- **Válaszhiba** (response): a komponens hibás választ küld
 - Értékhiba**: a válaszként küldött érték rossz
 - Állapotátmeneti hiba**: a komponens helytelen állapotba kerül
- **Váratlan hiba** (arbitrary): véletlenszerű válaszok, gyakran véletlenszerű időzítéssel

Összeomlás

Probléma

A kliens számára nem különböztethető meg, hogy a szerver **összeomlott** vagy csak **lassú**.

Ha a kliens a szervertől adatot vár, de nem érkezik, annak oka lehet...

- időzítési vagy kieséses hiba a szerveren.
- a szerver és a kliens közötti kommunikációs csatorna meghibásodása.

Lehetséges feltételezések

- **Fail-silent**: a komponens összeomlott vagy kiesett egy válasz; a kliensek nem tudják megkülönböztetni a kettőt
- **Fail-stop**: a komponens hibajelenségeket produkál, de ezek felismerhetőek (a komponens közzéteszi, vagy időtúllépésből szűrjük le)
- **Fail-safe**: a komponens csak olyan hibajelenségeket produkál, amelyek nem okoznak (nagy) kárt

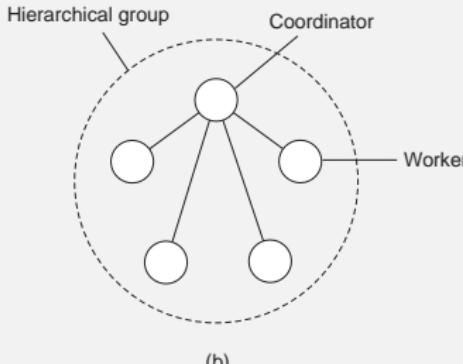
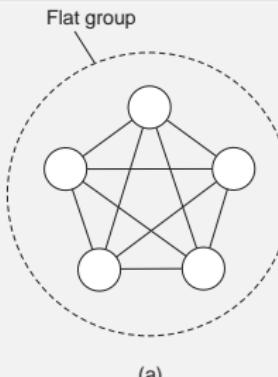
Folyamatok hibatűrése

Cél

Próbáljuk meg a hibákat redundanciával elfedni.

Egyenlő csoport: Jó hibatűrést biztosít, mert a csoport tagjai között közvetlen az információcsere. A nagymértékű többszörözöttség azonban megnehezíti az implementálást (a vezérlés teljesen elosztott).

Hierarchikus csoport: Két fél csak a koordinátoron keresztül képes kommunikálni egymással. Rossz a hibatűrése és a skálázhatósága, de könnyű implementálni.



Csoporttagság kezelése

Csoportkezelő: Egy koordinátor kezeli a csoportot (vagy csoportokat).
Rosszul skálázódik.

Csoportkezelők csoportja: A csoportkezelő nem egyetlen szerver, hanem egy csoportjuk közösen. Ezt a csoportot is kezelni kell, de ezek a szerverek általában elég stabilak, így ez nem probléma: centralizált csoportkezelés elegendő.

Csoportkezelő nélkül: A belépő/kilépő folyamat minden csoporttagnak üzenetet küld.

Hibaelfedés csoportokban

k-hibatűró csoport

k-hibatűró csoport: olyan csoport, amely képes elfedni k tag egyszerre történő meghibásodását.

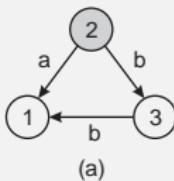
Mekkorának kell a csoportnak lennie?

- Ha a meghibásodottak **összeomlanak**, csak az szükséges, hogy legyen megmaradó tag $\Rightarrow k + 1$ tag
- Ha minden csoporttag **egyformán működik**, de a meghibásodottak **rossz eredményt adnak** $\Rightarrow 2k + 1$ tag esetén többségi szavazással megkapjuk a helyes eredményt
- Ha a csoporttagok **különbözően működnek**, és mindegyikük kimenetét el kell juttatni mindenki másnak („bizánci generálisok”) $\Rightarrow 3k + 1$ tag esetén a „lojális” szerverek képesek megfelelő adatokat továbbítani k „áruló” jelenlétében is

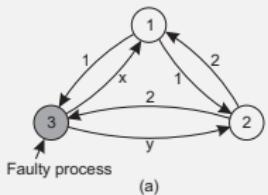
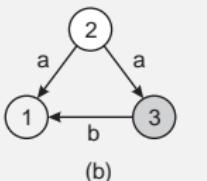
Hibaelfedés csoportokban

Az áruló folyamatok többféleképpen is tudnak helytelen adatokat továbbítani.

Process 2 tells different things



Process 3 passes a different value



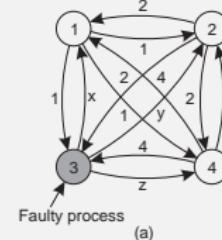
1 Got(1, 2, x)	1 Got (1, 2, y)	2 Got (1, 2, x)
2 Got(1, 2, y)	(a, b, c)	(d, e, f)
3 Got(1, 2, 3)		

(b)

(c)

Az alábbi két ábra illusztrálja $k = 1$ esetben, hogy k áruló esetén $3k + 1$ folyamat elegendő, de $3k$ még nem.

Először mindenki elküldi a saját értékét, majd minden beérkezett értéket ismét továbbítanak.



1 Got(1, 2, x, 4)	1 Got (1, 2, y, 4)	2 Got (1, 2, x, 4)	4 Got (1, 2, x, 4)
2 Got(1, 2, y, 4)	(a, b, c, d)	(e, f, g, h)	(1, 2, y, 4)
3 Got(1, 2, 3, 4)			
4 Got(1, 2, z, 4)	(1, 2, z, 4)	(1, 2, z, 4)	(i, j, k, l)

Hibaelfedés csoportokban

- Folyamatok:** Szinkron: a folyamatok azonos ütemben lépnek-e ([lockstep](#))?
- Késleltetések:** A kommunikációs késleltetésekre van-e felső korlát?
- Rendezettség:** Az üzeneteket a feladás sorrendjében kézbesítik-e?
- Átvitel:** Az üzeneteket egyenként küldjük ([unicast](#)) vagy többcíműen ([multicast](#))?

Egyezség elérése

Néhány feltételrendszer, amely fennállásával elérhető, hogy kialakuljon a közös végeredmény:

- Ha a ütemezés [szinkron](#) és a késleltetés [korlátozott](#).
- Ha az üzeneteket [sorrendtartó](#) módon, [többcíműen](#) továbbítjuk.
- Ha az ütemezés [szinkron](#) és a kommunikáció [sorrendtartó](#).

Hibák észlelése

Időtúllépés

Időtúllépés észlelése esetén feltételezhetjük, hogy hiba lépett fel.

- Az időkorlátok megadása alkalmazásfüggő, és nagyon nehéz
- A folyamatok hibázása nem megkülönböztethető a hálózati hibáktól
- A hibákról értesíteni kell a rendszer többi részét is
 - Pletykálással elküldjük a hiba észlelésének tényét
 - A hibajelenséget észlelő komponens maga is hibaállapotba megy át

Megbízható kommunikáció

Csatornák

A folyamatok megbízhatóságát azok csoportokba szervezésével tudtuk növelni. Mit lehet mondani az őket összekötő kommunikációs csatornák biztonságosságáról?

Hibajelenségek észlelése

- A csomagokat ellenőrző összegekkel látjuk el (felfedi a bithibákat)
- A csomagokat sorszámmal látjuk el (felfedi a kiesések)

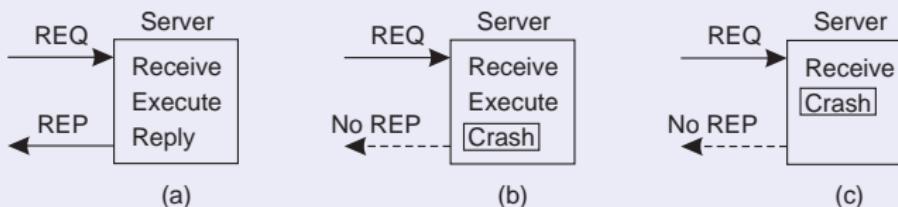
Hibajavítás

- A rendszer legyen annyira redundáns, hogy *automatikusan meg lehessen javítani*
- Kérhetjük a meghibásodott/elveszett utolsó N csomag újraküldését

Megbízható RPC

RPC: Fellépő hibajelenségek és azok kezelése

- 1: **A kliens nem találja a szert.** Ezt egyszerűen ki lehet jelezni a kliensnél.
- 2: **A kliens kérése elveszett.** A kliens újraküldi a kérést.
- 3: **A szerver összeomlott.** Ezt nehezebb kezelní, mert a kliens nem tudja, hogy a szerver mennyire dolgozta fel a kérést.



El kell dönteni, hogy milyen módon működjön a szerver.

- „**Legalább egyszer**” szemantika szerint: A szerver az adott kérést legalább egyszer végrehajtja (de lehet, hogy többször is).
- „**Legfeljebb egyszer**” szemantika szerint: A szerver az adott kérést legfeljebb egyszer hajtja végre (de lehet, hogy egyszer sem).

Megbízható RPC

RPC: Fellépő hibajelenségek és azok kezelése

- 4: **Elveszett a szerver válasza.** Nehéz felismerni, mert a szerver összeomlása is hasonló a kliens szemszögéből. **Nincsen általános megoldás;** ha **idempotens** műveleteink vannak (többszöri végrehajtás ugyanazt az eredményt adja), megpróbálhatjuk újra végrehajtani őket.
- 5: **Összeomlik a kliens.** Ilyenkor a szerver feleslegesen foglalja az erőforrásokat: **árva^a** feladatok keletkeznek.
- A kliens felépülése után az árva feladatokat szükség szerint leállítja/visszagörgeti a szerver.
 - A kliens felépülése után új korszak kezdődik: a szerver leállítja az árva feladatokat.
 - A feladatokra időkorlát adott. A túl sokáig futó feladatokat a szerver leállítja.

^aMás fordításban: fattyú.

Megbízható csoportcímzés

Adott egy többcímű átviteli csatorna; egyes folyamatok üzeneteket küldenek rá és/vagy fogadnak rólá.

Megbízhatóság: Ha m üzenet csatornára küldésekor egy folyamat a fogadók közé tartozik, és köztük is marad, akkor m üzenetet kézbesíteni kell a folyamathoz. Sokszor a **sorrendtartás** is követelmény.

Atomi csoportcímzés: Cél: elérni azt, hogy csak akkor kézbesítsük az üzenetet, ha garantáltan **mindegyik** fogadónak kézbesíthető.

Egyszerű megoldás lokális hálózaton

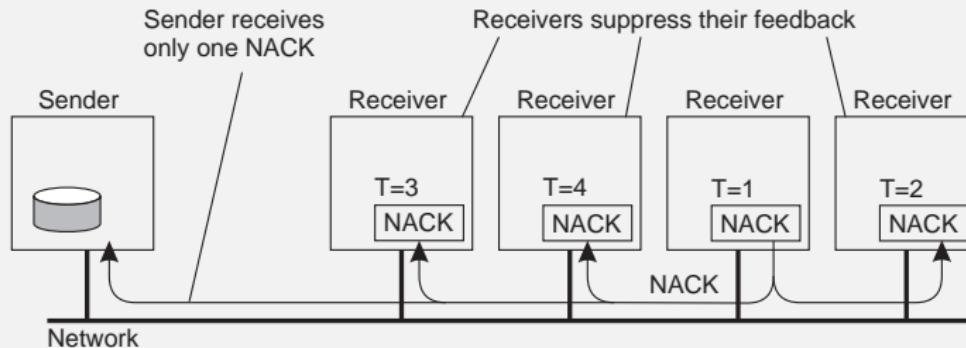
A küldő **számon tartja** az elküldött üzeneteket. minden fogadó **nyugtázza** (ACK) az üzenet fogadását, vagy kéri az üzenet újraküldését (NACK), ha észreveszi, hogy elveszett. Ha minden fogadótól megérkezett a nyugta, a küldő **törlí** az üzenetet a történetből.

Ez a megoldás **nem jól skálázódik**:

- Ha sok a fogadó, a küldőhöz túl sok ACK és NACK érkezik.
- A küldőnek minden fogadót ismernie kell.

Megbízható csoportcímzés: visszajelzés-elfojtás

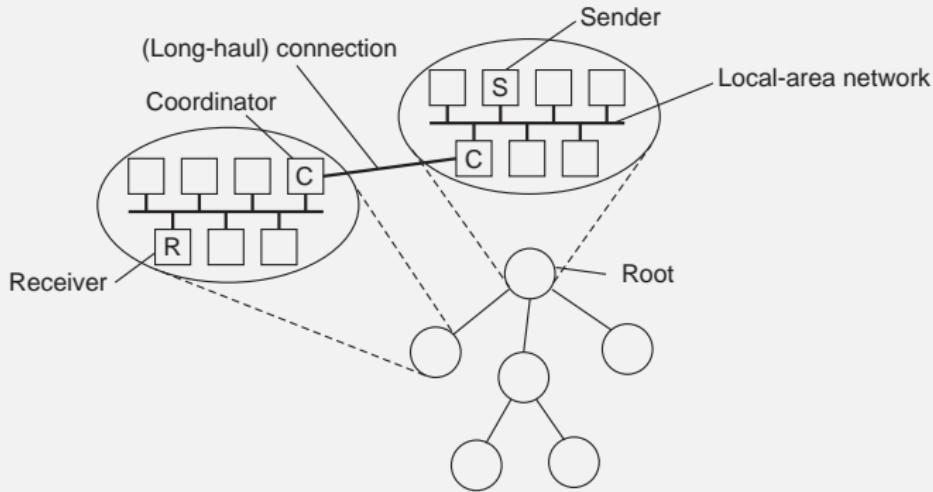
Hagyjuk el az ACK üzeneteket, csak a kímaradó üzenetek esetén küldjenek a fogadók NACK üzeneteket. Tegyük fel, hogy a NACK üzenetek közös csatornán utaznak, és mindenki látja őket. A P fogadó a NACK küldése előtt véletlen ideig vár; ha közben egy másik fogadótól NACK érkezik, akkor P nem küld NACK-t ([elfojtás](#)), ezzel csökkenti az üzenetek számát. A NACK hatására a küldő mindenki fogadóhoz újra eljuttatja az üzenetet, így P -hez is.



Megbízható csoportcímzés: hierarchikus visszajelzés-vezérlés

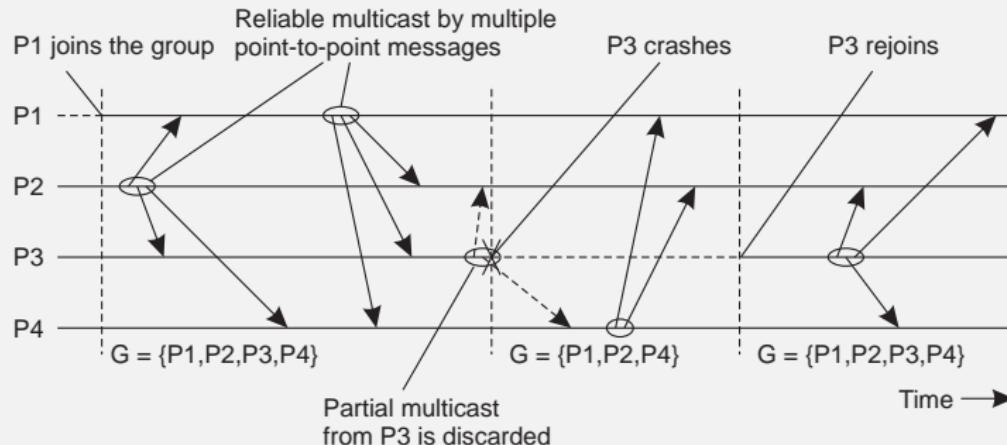
Az előző módszer lokális hálózatban működik hatékonyan. Az átméretezhetőség javításához a lokális hálózatokból építsünk fát; az üzenetek a fában felfelé, a NACK üzenetek felfelé utaznak.

Probléma: a fát dinamikusan kell építeni, a meglévő hálózatok átkonfigurálása nehéz lehet.



Megbízható csoportcímzés: elemi csoportcímzés

A folyamatok számon tartják, hogy kik a csoport tagjai: **nézet** (view). Egy üzenetet csak akkor kézbesítenek, ha az eljutott az adott nézet minden tagjához. Mivel a nézetek időben változhathatnak, ez ún. **látszólagos szinkronizáltságot** (virtual synchrony) ad.



Elosztott vélegesítés: 2PC

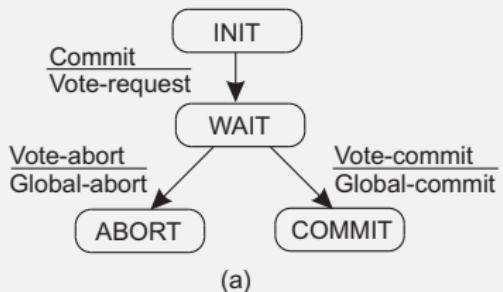
2PC

Egy elosztott számítás végén szeretnénk elérni, hogy vagy minden folyamat vélegesítse az eredményt, vagy egyik sem. (atomiság) Ezt általában **kétfázisú vélegesítéssel** (two-phase commit, 2PC) szokás elérni. A számítást az azt kezdeményező kliens koordinátorként vezérli. Az egyes fázisokban a koordinátor (K) és egy-egy résztvevő (R) az alábbiak szerint cselekszik.

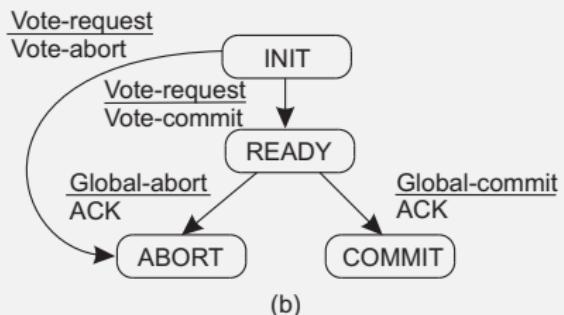
- **1/K:** Megkérdez mindenkit, enged-e vélegesíteni: **vote-request**.
- **1/R:** Dönt: igennel (**vote-commit**) vagy nem (**vote-abort**), utóbbi esetben rögtön el is veti a kiszámított értéket.
- **2/K:** Ha minden válasz **vote-commit**, mindenkinek **global-commit** üzenetet küld, különben **global-abort** üzenetet.
- **2/R:** Végrehajtja a kapott globális utasítást: elmenti a kiszámított adatokat (eddig csak átmenetileg tárolta őket lokálisan), illetve törli őket.

Elosztott végegesítés: 2PC

A vonalak fölött a kapott, alattuk a küldött üzenetek nevei láthatóak.



A 2PC koordinátor állapotgépe



Egy 2PC résztvevő állapotgépe

Elosztott vélegesítés: 2PC, résztvevő összeomlása

Résznevő összeomlása

Egy résztvevő összeomlott, majd felépült. Mit tegyen adott állapotban?

- INIT: A résztvevő nem is tudott a vélegesítésről, döntés: **ABORT**.
- ABORT: Az eredmények **törlendők**. Az állapot marad **ABORT**.
- COMMIT: Az eredmények **elmentendők**. Az állapot marad **COMMIT**.
- READY: A résztvevő a koordinátor 2/K döntésére vár, amit megkérdez tőle, vagy a koordinátor összeomlása esetén a többi résztvevőtől. Azok vagy már tudnak róla (**ABORT** vagy **COMMIT**), vagy még **INIT** állapotban vannak (akkor **ABORT** mellett dönthetnek). Ha mindenki résztvevő **READY** állapotban van, akkor megoldható a helyzet, azonban...

A 2PC protokoll blokkolódhat

Ha a koordinátor és legalább egy résztvevő összeomlott, és a többi résztvevő a **READY** állapotban van, akkor a **protokoll beragad**, ugyanis nem ismert, hogy a kiesett résztvevő kapott-e az összeomlása előtt valamilyen utasítást a koordinátortól. Ez szerencsére **szinte sohasem fordul elő a gyakorlatban**.

Elosztott vélegesítés: 3PC

3PC

- 1/K és 1/R: Mint korábban: megkezdődik a szavazás (**vote-request**), és a résztvevők válaszolnak (**vote-commit** vagy **vote-abort**)
- 2/K: Ha van **vote-abort** szavazat, minden résztvevőnek **global-abort** üzenetet küld, majd leáll. Ha minden szavazat **vote-commit**, **prepare-commit** üzenetet küld mindenkinél.
- 2/R: **global-abort** esetén leáll; **prepare-commit** üzenetre **ready-commit** üzenettel válaszol.
- 3/K: Összegyűjti a **prepare-commit** üzeneteket; küldi: **global-commit**.
- 3/R: Fogadja a **global-commit** üzenetet, és elmenti az adatokat.

A 3PC protokoll nem blokkolódhat

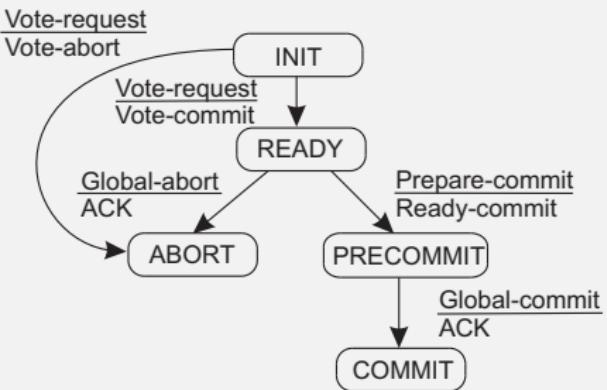
Mivel a koordinátor és a résztvevők mindig legfeljebb „egy távolságban vannak” az állapotaikat tekintve, ha mindegyik aktív résztvevő **READY** állapotban van, akkor a kiesett résztvevő még nem járhat a **COMMIT** fázisban. Megjegyzés: a **PRECOMMIT** állapotból csak vélegesíteni lehet.

Elosztott végegesítés: 3PC



(a)

A 3PC koordinátor állapotgépe



(b)

Egy 3PC résztvevő állapotgépe

Felépülés

Alapok

Ha hibajelenséget érzékelünk, minél hamarabb hibamentes állapotba szeretnénk hozni a rendszert.

- **Előrehaladó felépülés:** olyan új állapotba hozzuk a rendszert, ahonnan az folytathatja a működését
- **Visszatérő felépülés:** egy korábbi érvényes állapotba térünk vissza

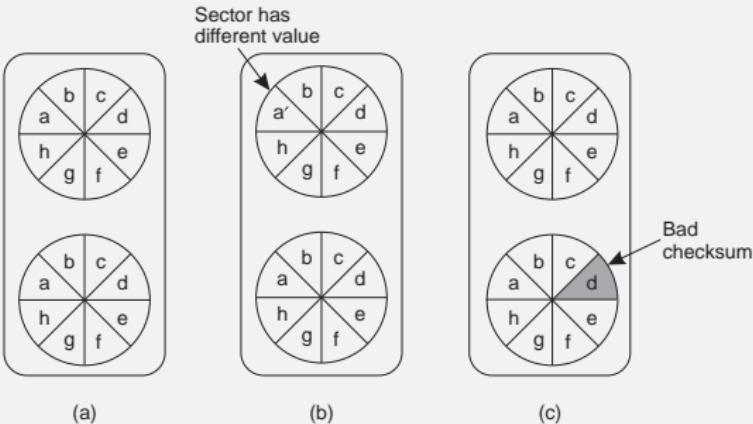
Jellemző választás

Visszatérő felépülést alkalmazunk, ehhez **ellenőrzőpontokat** (checkpoint) veszünk fel.

Konzisztens állapot szükséges

Az elosztott rendszerek visszaállítását nehezíti, hogy ellenőrzőpontot a rendszer egy **konzisztens állapotára** szeretnénk elhelyezni, ennek megtalálásához pedig a folyamatok együttműködése szükséges.

Stabil tárolás (stable storage)



Az ellenőrzőpontokat minél megbízhatóbban kell tárolnunk. Tegyük fel, hogy fizikailag szinte elpusztíthatatlan tárunk van (stable storage), és két másolatot használunk; mit kell tenni összeomlás után?

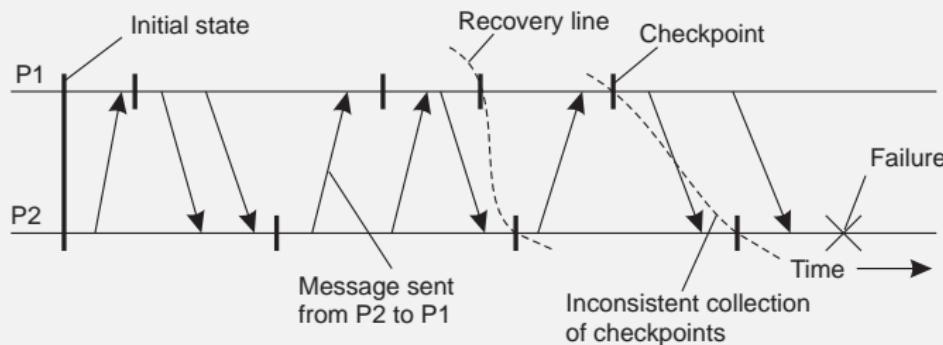
- Ha minden lemez tartalma azonos, minden rendben.
- Ha az ellenőrzőösszegek alapján kiderül, melyik a helyes, azt választjuk.
- Ha helyesek, de eltérnek, az első lemezt választjuk, mert oda írunk először.
- Ha egyik sem helyes, nehezen eldönthető, melyiket kell választani.

Konzisztens rendszerállapot

A folyamatok pillanatfelvételeket készítenek az állapotukról.

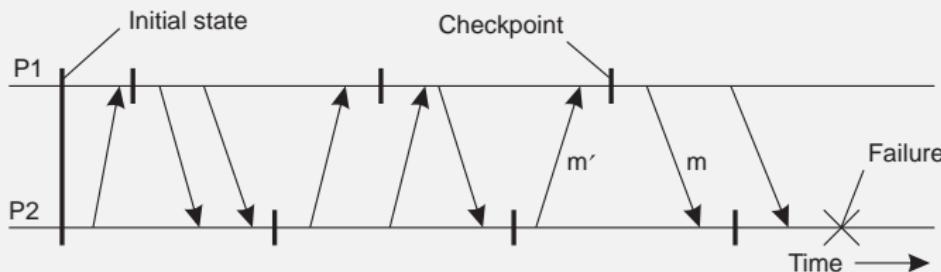
konzisztens metszet: olyan ellenőrzőpont-gyűjtemény, amelyben minden beérkezett üzenethez a küldési esemény is el van tárolva

felépülési vonal: a lehető legkésőbb készült (legfrissebb) konzisztens metszet



Dominóeffektus

Vissza kell keressünk a legutolsó konzisztens metszetet. Ezért minden folyamatban vissza kell térni egy korábbi ellenőrzőponthoz, ha „elküldetlen” fogadott üzenetet találunk. Így viszont további üzenetek válnak nem elküldötté, ami továbbgyűrűző visszagörgetéshez ([cascaded rollback](#)) vezethet.



Független ellenőrzőpontok készítése

Egy lehetőség, ha minden folyamat egymástól függetlenül készít ellenőrzőpontokat (independent checkpointing).

- Jelölje $CP[i](m)$ a P_i folyamat m -edik ellenőrzőpontját.
- Jelölje $INT[i](m)$ a $CP[i](m-1)$ és $CP[i](m)$ közötti időintervallumot.
- Amikor P_i üzenetet küld a $INT[i](m)$ intervallumban, hozzácsatolja i és m értékét.
- Amikor P_j fogadja az üzenetet a $INT[j](n)$ intervallumban, rögzíti, hogy új függőség keletkezett: $INT[i](m) \xrightarrow{\text{függ}} INT[j](n)$, és ezt $CP[j](n)$ készítésekor a stabil tárba menti
- Ha P_i -nek vissza kell állítania $CP[i](m-1)$ -et, akkor P_j -t is vissza kell görgetni $CP[i](m-1)$ -ig. A függőségek egy gráfot adnak ki, ennek a vizsgálatával derül ki, melyik folyamatot pontosan melyik ellenőrzőpontig kell visszagörgetni, legrosszabb esetben akár a rendszer kezdőállapotáig.

Koordinált ellenőrzőpontkészítés

A dominóeffektus elkerülése végett most egy koordinátor vezérli az ellenőrzőpont készítését.

Kétfázisú protokollt használunk:

- Az első fázisban a koordinátor minden folyamatot pillanatfelvétel készítésére szólít fel.
- Amikor ezt megkapja egy folyamat, elmenti az állapotát, és ezt nyugtázza a koordinátor felé. A folyamat továbbá felfüggeszti az üzenetek küldését.
- Amikor a koordinátor minden nyugtát megkapott, minden folyamatnak engedélyezi az üzenetek küldését.

Felépülés naplózással

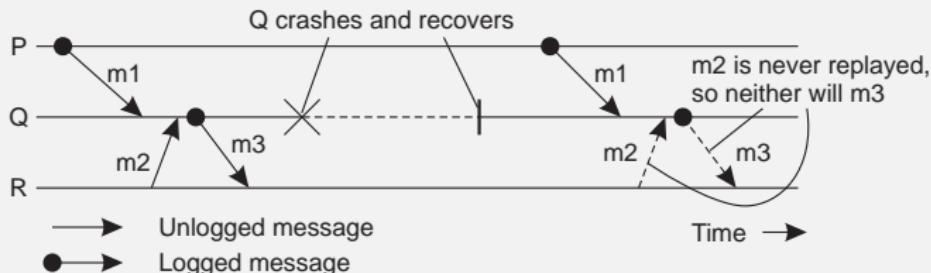
Ahelyett, hogy pillanatfelvételt készítenénk, most megpróbáljuk **újra lejátszani** a kommunikációs lépéseket az utolsó ellenőrzőponttól. Ehhez mindegyik folyamat lokális **naplót vezet**.

A végrehajtási modellről feltételezzük, hogy **szakaszosan determinisztikus** (piecewise deterministic):

- Mindegyik folyamat végrehajtását egymás után következő szakaszok sorozata
- Mindegyik szakasz egy nemdeterminisztikus eseménnyel kezdődik (pl. egy üzenet fogadása, szignál kezelése)
- A szakaszon belül a végrehajtás determinisztikus

A naplóba elég a nemdeterminisztikus eseményeket felvenni, a többi lépés belőlük meghatározható.

Üzenetek konzisztens naplázása



Úgy szeretnénk naplózni, hogy elkerüljük árva folyamatok kialakulását.

- Q folyamat fogadta m_1 -et és m_2 -t, és elküldte m_3 -at.
- Tegyük fel, hogy m_2 üzenetet se R, se Q nem naplózta
- Q összeomlása után a visszajátszás során ekkor **senkinek sem tűnik fel**, hogy m_2 kimaradt
- Mivel Q kihagyta m_2 -t, ezért a tőle (lehet, hogy) függő m_3 -at sem küldi el

Üzenetek konzisztens naplázása

Jelölések

HDR[m]: Az m üzenet fejléce, tartalmazza a küldő és a folyamat azonosítóját és az üzenet sorszámát.

Egy üzenet **stabil**, ha a fejléce már biztosan nem veszhet el (pl. mert stabil tárolóra írták).

COPY[m]: Azok a folyamatok, amelyekhez $HDR[m]$ már megérkezett, de még nem tárolták el. Ezek a folyamatok képesek $HDR[m]$ reprodukálására.

DEP[m]: Azok a folyamatok, amelyekhez megérkezett $HDR[m']$, ahol m' okozatilag függ m -től.

Ha C összeomlott folyamatok halmaza, akkor $Q \notin C$ árva, ha függ olyan m üzenettől, amelyet csak az összeomlottak tudnának előállítani:

$$Q \in DEP[m] \text{ és } COPY[m] \subseteq C.$$

Ha $DEP[m] \subseteq COPY[m]$, akkor nem lehetnek árva folyamataink.

Üzenetek konzisztens naplázása

Pesszimista naplázóprotokoll

Ha m nem stabil, akkor megköveteljük, hogy legfeljebb egy folyamat függön tőle: $|DEP[m]| \leq 1$.

Implementáció: minden nem-stabil üzenetet stabilizálni kell továbbküldés előtt.

Optimistika naplázóprotokoll

Tegyük fel, hogy C a hibás folyamatok halmaza. A nem-stabil m üzenetekre, ha $COPY[m] \subseteq C$, akkor azt szeretnénk elérni, hogy egy idő után $DEP[m] \subseteq C$ is teljesüljön.

Implementáció: minden árva folyamatot visszagörgetünk olyan ellenőrzőpontig, ahol a függőség már nem áll fenn.

Elosztott rendszerek: Alapelvek és paradigmák

Distributed Systems: Principles and Paradigms

Maarten van Steen¹ Kitlei Róbert²

¹VU Amsterdam, Dept. Computer Science

²ELTE Informatikai Kar

10. rész: Objektumalapú elosztott rendszerek

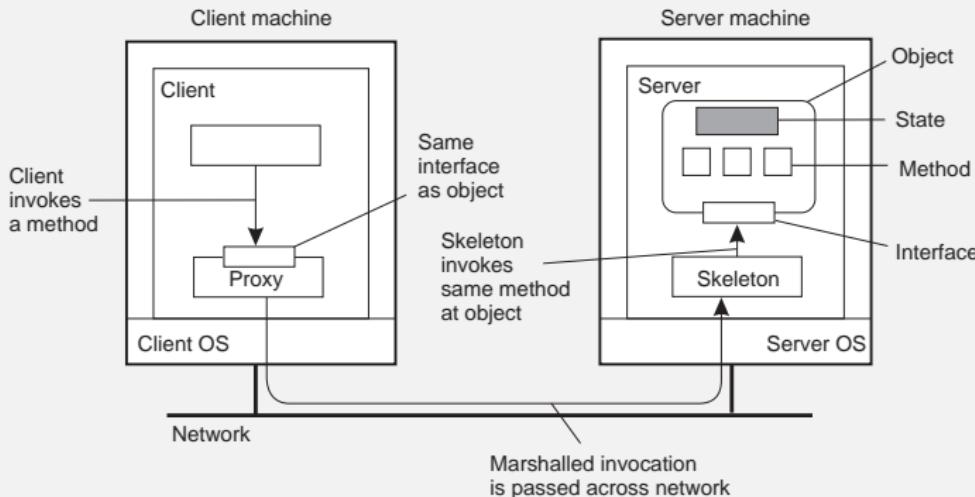
2015. május 24.

Tartalomjegyzék

Fejezet
01: Bevezetés
02: Architektúrák
03: Folyamatok
04: Kommunikáció
05: Elnevezési rendszerek
06: Szinkronizáció
07: Konzisztencia & replikáció
08: Hibatűrés
10: Objektumalapú elosztott rendszerek
11: Elosztott fájlrendszerek
12: Elosztott webalapú rendszerek

Távoli elosztott objektumok

- Objektum: műveleteket és adatokat **zár egységebe** (enkapszuláció)
- A műveleteket **metódusok** implementálják, ezeket **interfészekbe** csoportosítjuk
- Az objektumokat csak az **interfészükön** keresztül érhetik el a kliensek
- Az objektumokat **objektumszerverek** tárolják
- A kliensoldali **helyettes** (proxy) megvalósítja az interfészt
- A szerveroldalon a **váz** (skeleton) kezeli a beérkező kéréseket



Távoli elosztott objektumok

Objektumok létrehozás ideje alapján

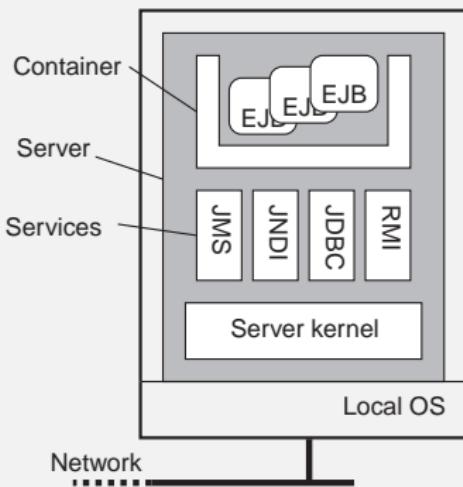
- **Fordítási időben létrejövő objektumok:** A helyettesítő és a vázat a fordítóprogram készíti el, összeszerkeszti a kliens és a szerver kódjával. Nem cserélhető le, miután létrejött, és a klienssel/szerverrel azonos a programozási nyelve.
- **Futási időben létrejövő objektumok:** Tetszőleges nyelven valósítható meg, de [objektumadapterre](#) van szükség a szerveroldalon a használatához.

Objektumok élettartamuk alapján

- **Átmeneti** (transzient) objektum: Élettartama csak addig tart, amíg be van töltve a szerverbe. Ha a szerver kilép, az objektum is megsemmisül.
- **Tartós** (persistent) objektum: Az objektum állapotát és kódját lemezre írjuk, így a szerver kilépése után is megmarad. Ha a szerver nem fut, az objektum passzív; amikor a szerver elindul, betöltéssel aktivizálható.

Példa: Enterprise Java Beans (EJB)

Az objektumokat alkalmazásszerverek (pl. GlassFish) tárolják, amelyek lehetővé teszik az objektumok különböző módokon való elérését.



Példa: Enterprise Java Beans (EJB)

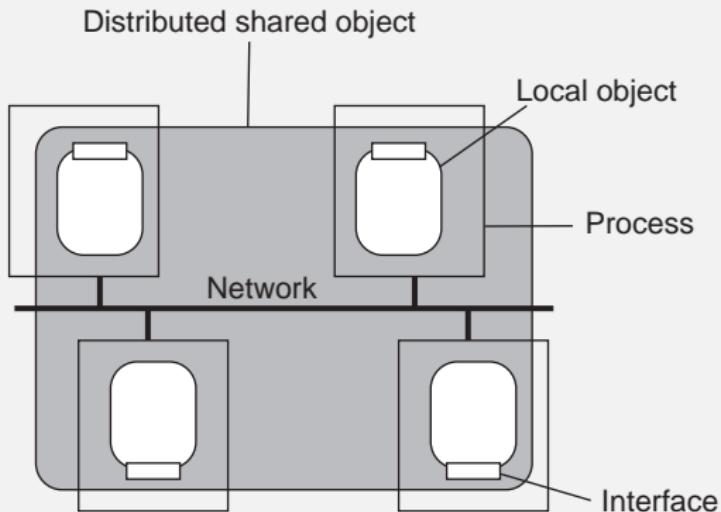
EJB-k fajtái

- **Stateless session bean:** Tranziens objektum, egyszer hívják meg, miután elvégezte a feladatát, megszűnik. **Példa:** egy SQL lekérdezés végrehajtása, és az eredmény átadása a kliensnek.
- **Stateful session bean:** Tranziens objektum, de a klienssel egy munkameneten (session) keresztül tartja a kapcsolatot, ezáltal állapotot is tart fenn. **Példa:** bevásárlókosár.
- **Entity bean:** Perzisztens, állapottal rendelkező objektum, amely több munkamenetet is ki tud szolgálni. **Példa:** olyan objektum, amely az utolsó néhány kapcsolódó kliensről tárol adatokat.
- **Message-driven bean:** Különböző fajta üzenetekre reagálni képes objektum. A **publish/subscribe** kommunikációs modell szerint működik.

Globe elosztott objektumok

Általában a távoli objektumok nem elosztottak: az állapotukat egy gép tárolja.

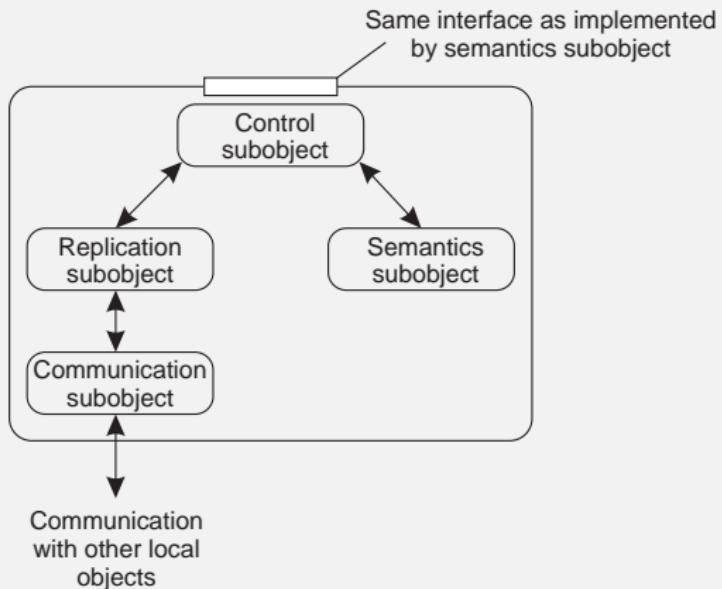
A Globe rendszerben az objektumok fizikailag több gépen helyezkednek el: elosztott közös objektum (distributed shared object, DSO).



Globe elosztott objektumok

Az elosztottság támogatásához belső architektúra szükséges, és célszerű, ha ez független attól, hogy a DSO külső felülete milyen szolgáltatásokat nyújt.

A replikációkezelő alobjektum vezérli, hogy **hogyan** és **mikor** kell a lokális szemantikus alobjektumot meghívni.



Folyamatok: Objektumszerverek

A rendszer részei a **kiszolgálók**, a **vázak** és az **adaptek**.

A kiszolgálót (servant), amely az objektum működését biztosítja, több paradigma szerint lehet implementálni:

- Függvények gyűjteménye, amelyek adatbázistáblákat, rekordokat stb. manipulálnak (pl. C vagy COBOL nyelven)
- Osztályok (pl. Java vagy C++ nyelven)

A váz (skeleton) a szerveroldali hálózati kapcsolatokat kezeli:

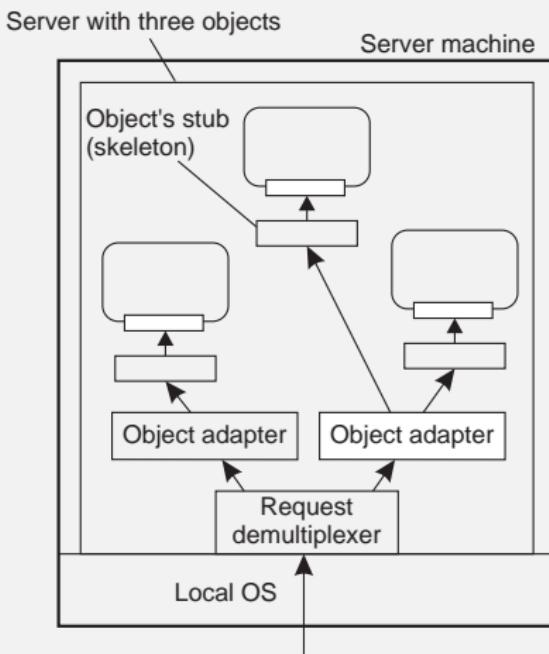
- Kicsomagolja a beérkező kéréseket, lokálisan meghívja az objektumot, becsomagolja és visszaküldi a választ
- Az interfész specifikációja alapján hozzák létre

Az objektumadapter feladata objektumok egy csoportjának kezelése:

- Elsőként fogadja a kéréseket, és azonosítja a releváns kiszolgálót
- **Aktivációs házirend** (policy) szerint aktiválja a megfelelő vázat
- Az adapter generálja az **objektumhivatkozásokat**

Folyamatok: Objektumszerverek

Az objektumszerverek vezérlik a tartalmazott objektumok létrehozását.



Példa: Ice

Az Ice (Internet Communications Engine) objektumorientált köztesréteg, szolgáltatásai elérhetőek a legnépszerűbb nyelveken. Az aktivációs házirend megjelenik a köztesréteg szintjén: az adapter properties adattagján át lehet módosítani. Ez segíti a rendszer egyszerű kezelhetőségét.

```
main(int argc, char* argv[]) {
    Ice::Communicator ic = Ice::initialize(argc, argv);
    Ice::ObjectAdapter adapter =
        ic->createObjectAdapterWithEndPoints("a", "tcp -p 2000");
    Ice::Object object = new MyObject;

    adapter->add(object, objectID);
    adapter->activate();

    ic->waitForShutdown();
}
```

Kliens csatlakoztatása objektumhoz

Objektumhivatkozás

Ha egy kliens birtokol egy referenciát egy objektumra, képes hozzá csatlakozni (**bind**):

- A hivatkozás előírja, melyik szerveren, melyik objektumot, milyen kommunikációs protokoll szerint lehet rajta keresztül elérni
- A hivatkozáshoz kód tartozik, ezt a konkrét objektum eléréséhez felparaméterezve kapjuk a helyetteset

Kétfajta csatlakozás

- **Implicit:** Magán a hivatkozott objektumon hívjuk meg a műveleteket
- **Explicit:** A kliens kódjában a csatlakozás explicit megjelenik

Kliens-objektum csatlakozás: implicit/explicit

```
// implicit                                // explicit
Distr_object* obj_ref;                      Distr_object* obj_ref;
obj_ref = ...;                             Local_object* obj_ptr;
obj_ref->do_something();                  obj_ref = ...;
                                            obj_ptr = bind(obj_ref);
                                            obj_ptr->do_something();
```

- A hivatkozás tartalmazhat egy URL-t, ahonnan az implementáció letölthető
- Protokollban rögzíthető, hogyan kell betölteni és példányosítani a letöltött kódot
- A szerver és az objektum ismerete elegendő lehet a távoli metódushívás megkezdéséhez
- Objektumhivatkozások paraméterként is átadhatóak, amik RPC esetében bonyodalmakat okoznak

Távoli metódushívás (Remote Method Invocation, RMI)

Tegyük fel, hogy a helyettes és a váz rendelkezésre áll a kliensnél/szervernél.

- 1 A kliens meghívja a helyettest
- 2 A helyettes becsomagolja a hívás adatait, és elküldi a szervernek
- 3 A szerver biztosítja, hogy a hivatkozott objektum aktív:
 - Külön folyamatot hozhat létre, amely tárolja az objektumot
 - Betöltheti az objektumot a szerverfolyamatba
 - ...
- 4 Az objektum váza kicsomagolja a kérést, és a metódus meghívódik
- 5 Ha paraméterként objektumhivatkozást kaptunk, ezt szintén távoli metódushívással éri el a szerver; ebben a szerver kliensként vesz részt
- 6 A választ hasonló úton küldjük vissza, a helyettes kicsomagolja, és visszatér vele a klienshez

RMI: Paraméterátadás

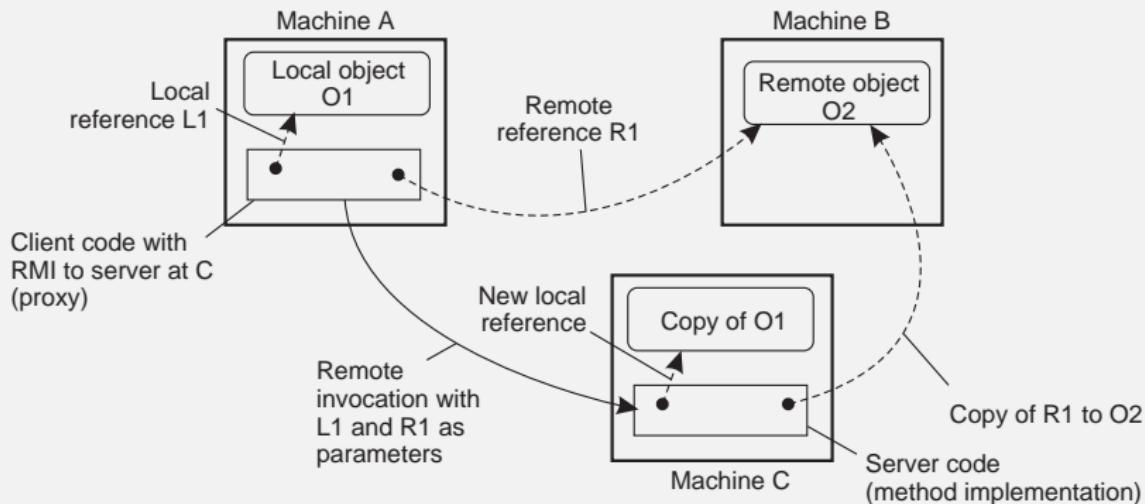
Hivatkozás szerinti paraméterátadás: sokkal egyszerűbb, mint RPC esetén.

- A szerver egyszerűen távoli metódushívással éri el az objektumot
- Ha már nincsen szüksége rá, megszünteti a csatolást (unbind)

Érték szerinti paraméterátadás: RMI esetén ez kevésbé kényelmes.

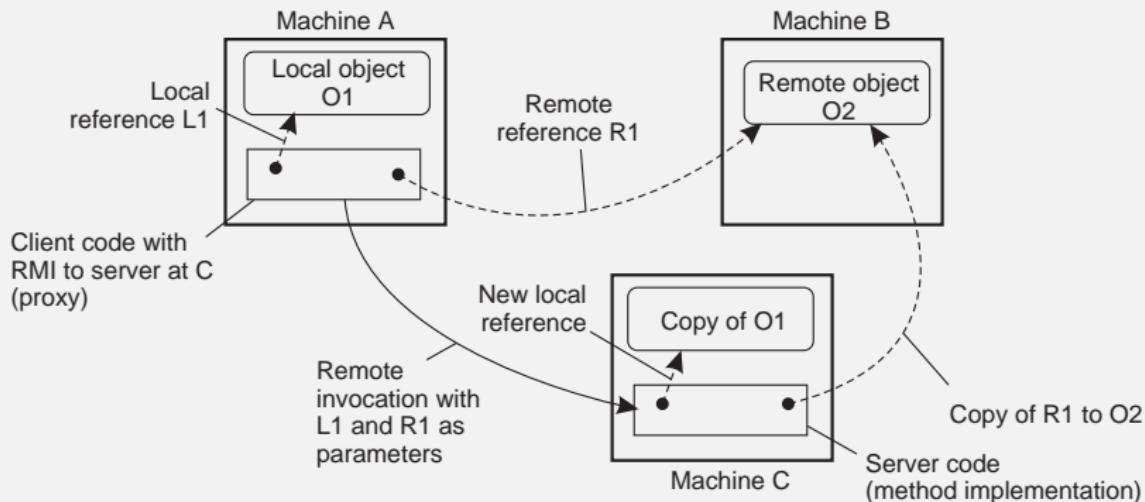
- Szerializálni kell az objektumot
 - Az állapotát
 - A metódusait, vagy hivatkozást olyan helyre, ahol elérhető az implementációjuk
- Amikor a szerver kicsomagolja az objektumot, ezzel **másolat készül az eredetiről**
- Ez az automatikus másolódás többféle problémát okoz, pl. néha „túl átlátszó”: könnyen több objektumról készíthetünk másolatot, mint amennyiről szeretnénk

RMI: Paraméterátadás



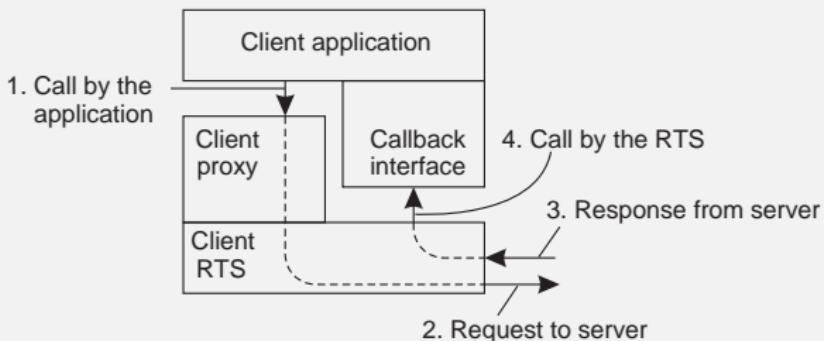
A rendszerszintű objektumhívatkozások általában a szerver címét, adapterének portját és az objektum lokális azonosítóját tartalmazzák. Néha ezekhez további információk is járulnak, pl. a kliens és szerver között használt protokoll (TCP, UDP, SOAP stb.)

RMI: Paraméterátadás

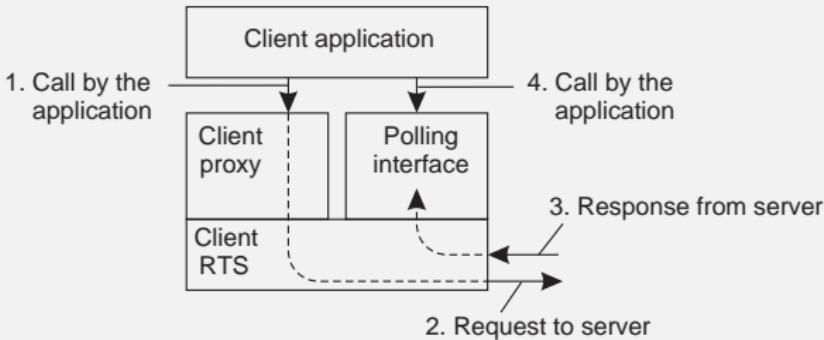


Mivel a helyettesek mindenféleképpen ismernie kell a hivatkozáshoz szükséges adatokat (cím, port, lokális ID), felhasználhatjuk a **helyettes magát mint távoli hivatkozást**. Ez különösen előnyös, ha a helyettes letölthető (pl. a Java esetében igen).

Objektumalapú üzenetküldés



A kliens az
üzenetekre várás
közben lehet aktív
vagy passzív.

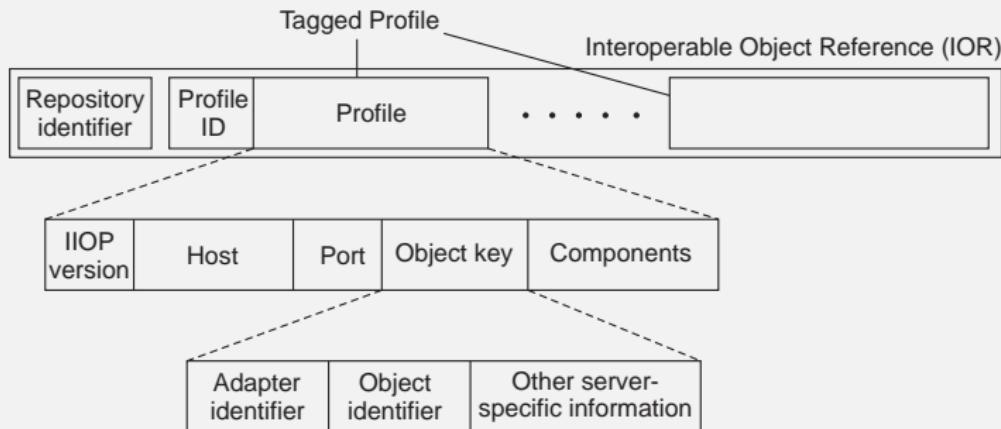


Objektumhivatkozások

IIOP¹: távoli objektumhivatkozásokat kezelő protokoll

CORBA²: Objektumalapú köztesréteg, IIOP-t használ

Az alábbi ábrán a CORBA objektumhivatkozásának szerkezete látható.



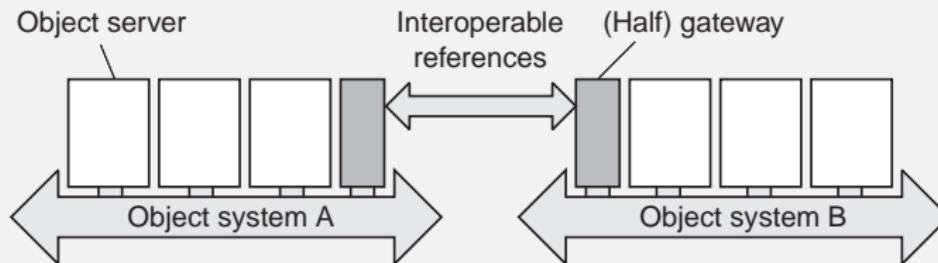
¹Internet Inter-ORB Protocol

²Common Object Request Broker Architecture

Objektumhivatkozások

Különböző objektumkezelő rendszerekben a hivatkozások szerkezete nagymértékben eltérhet.

A rendszerek között átjárók (gateway) biztosíthatják a hivatkozások konvertálását.



Replikáció és konzisztencia

Az objektumok a **belépő konzisztencia** megvalósításának természetesen adódó eszközei:

- Az adatok egysége vannak zárva, és **szinkronizációs** változóval (**zárral**) védjük őket
- A szinkronizációs változókat **soros konzisztencia** szerint érjük el (az értékek beállítása atomi lépés)
- Az adatokat kezelő műveletek összessége pont az objektum interfésze lesz

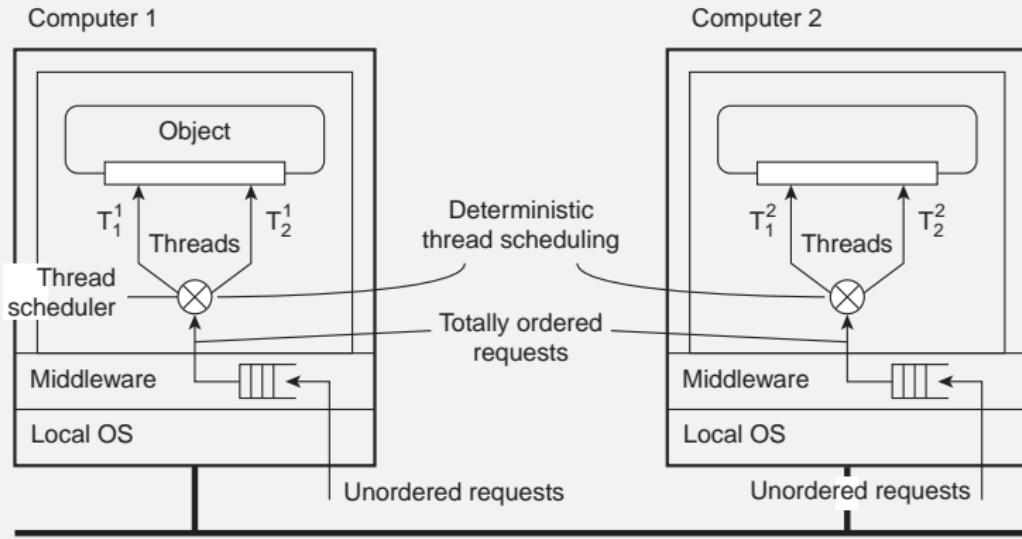
Replikáció

Mit tegyünk, ha az objektumot replikálni kell? A replikált objektumokon a műveletek végrehajtásának sorrendjének azonosnak kell lennie.

Replikált objektumok

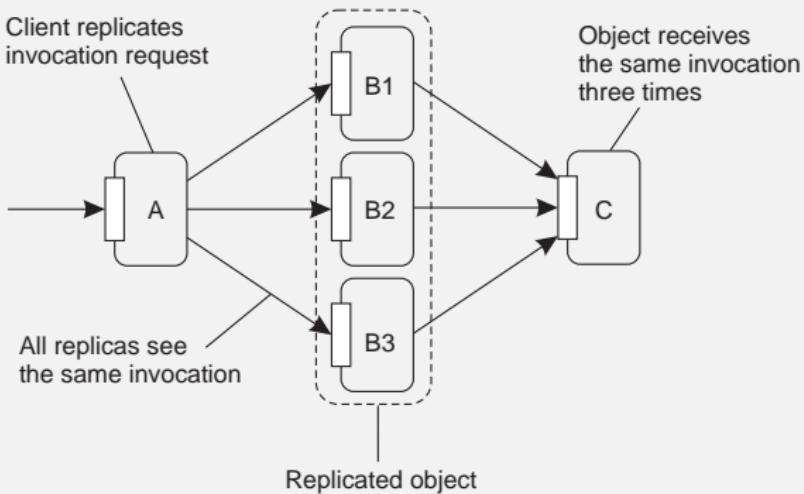
Nemcsak a kéréseknek kell sorrendben beérkezniük a replikátumokhoz; a vonatkozó szálak ütemezésének determinisztikusnak kell lennie.

Egyszerű megoldás lehetne, ha teljesen sorosítva (egyetlen szálon) hajtanánk végre a kéréseket, de ez túl költséges.



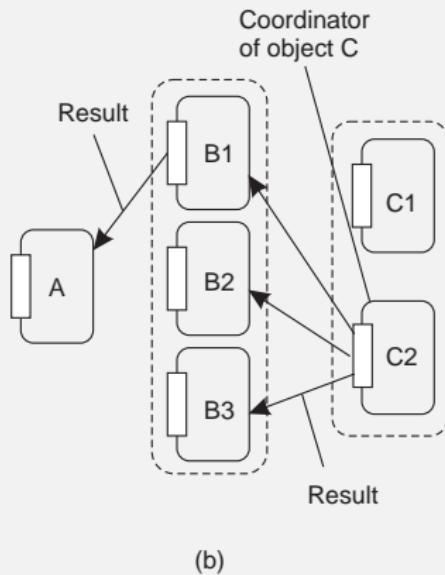
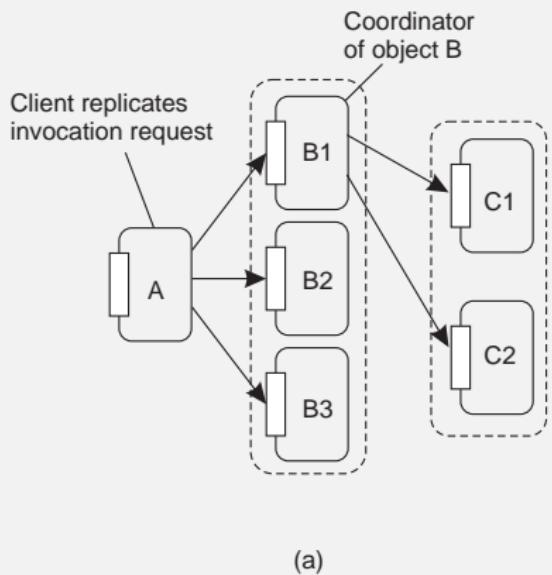
Replikált hívások

Aktív replikáció: ha a replikált objektum hívás során maga is meghív más objektumot, az a kérést többszörörzve kapná meg.



Replikált hívások

Megoldás: mind a szerver-, mind a kliensobjektumon válasszunk koordinátort, és csak a koordinátorok küldhessenek kéréseket és válaszokat.



Elosztott rendszerek: Alapelvek és paradigmák

Distributed Systems: Principles and Paradigms

Maarten van Steen¹ Kitlei Róbert²

¹VU Amsterdam, Dept. Computer Science

²ELTE Informatikai Kar

11. rész: Elosztott fájlrendszerök

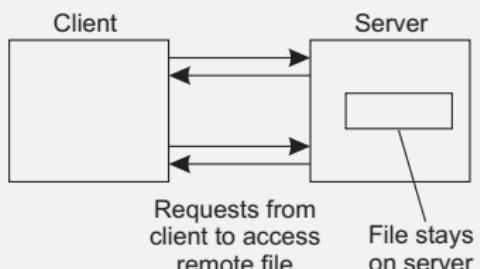
2015. május 24.

Tartalomjegyzék

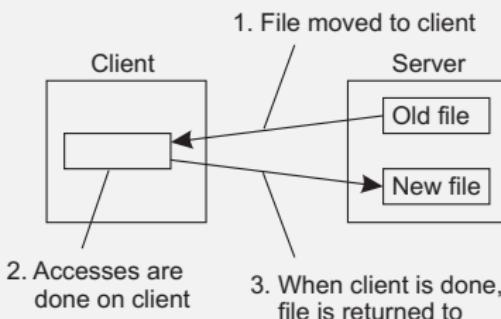
Fejezet
01: Bevezetés
02: Architektúrák
03: Folyamatok
04: Kommunikáció
05: Elnevezési rendszerek
06: Szinkronizáció
07: Konzisztencia & replikáció
08: Hibatűrés
10: Objektumalapú elosztott rendszerek
11: Elosztott fájlrendszerek
12: Elosztott webalapú rendszerek

Elosztott fájlrendszerek

Cél: a fájlrendszer átlátszó elérését biztosítani távoli kliensek számára.



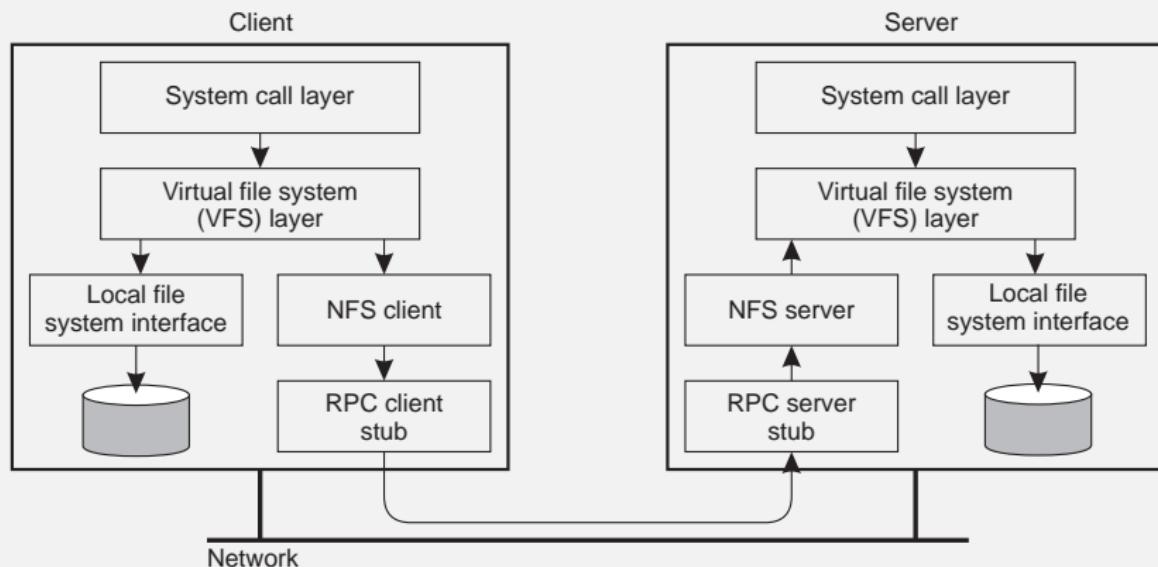
Távoli hozzáférési modell



Feltöltés/letöltés modell

Példa: NFS architektúra

Az NFS (Network File System) elosztott fájlok távoli elérését teszi lehetővé. Az alkalmazások a helyi VFS (Virtual File System) réteget érik el, ez biztosítja a távoli elérés átlátszóságát.

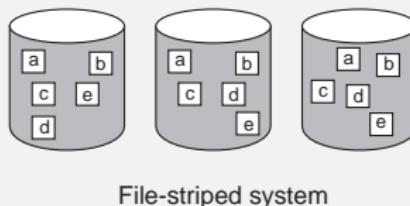
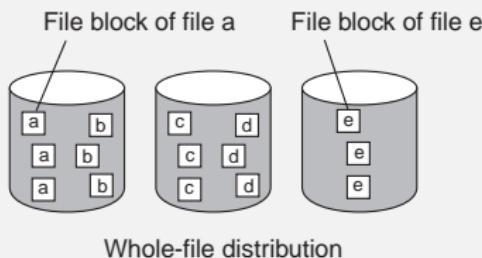


NFS fájlműveletek

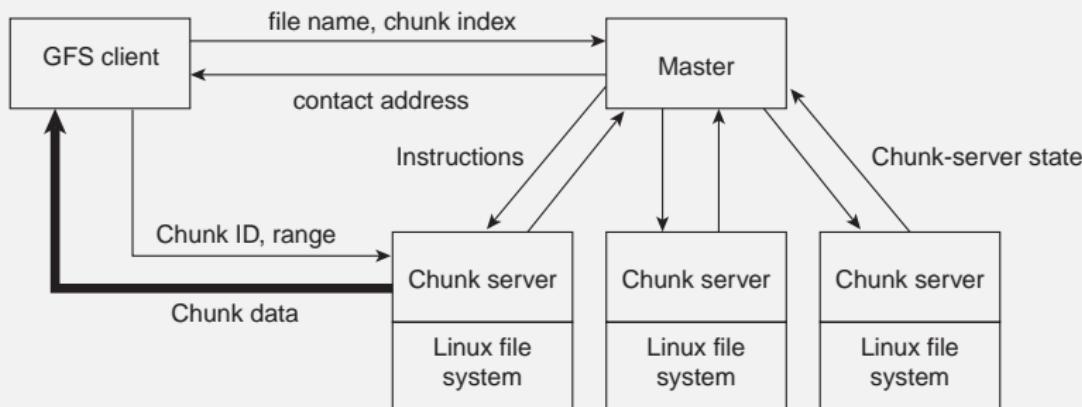
Művelet	Verzió	Leírás
Create	v3	Hagyományos fájl létrehozása
Create	v4	Nem-hagyományos fájl létrehozása (hagyományos: Open)
Link	v3 és v4	Valódi csatolás (hard link) létrehozása fájlról
Symlink	v3	Szimbolikus csatolás (soft link, symlink) létrehozása fájlról
Mkdir	v3	Alkönyvtár létrehozása
Mknod	v3	Különleges fájl létrehozása
Rename	v3 és v4	Fájl átnevezése
Remove	v3 és v4	Fájl eltávolítása a fájlrendszerből
Rmdir	v3	Üres könyvtár eltávolítása
Open	v4	Fájl megnyitása
Close	v4	Fájl bezárása
Lookup	v3 és v4	Fájl megkeresése név szerint
Readdir	v3 és v4	Könyvtár tartalmának lekérése
Readlink	v3 és v4	Szimbolikus csatolás elérési útvonalának olvasása
Getattr	v3 és v4	Fájl tulajdonságainak lekérdezése
Setattr	v3 és v4	Egy vagy több tulajdonság írása
Read	v3 és v4	Fájl tartalmának olvasása
Write	v3 és v4	Fájl adatainak írása

Fürt (cluster) alapú fájlrendszerek

Nagy fájlrendszerek esetén a kliens-szerver alapú megközelítés nem elég jó ⇒ a fájlokat csíkokra bontjuk (**striping**), így a részeiket párhuzamosan érjük el. Ennek célja a rendszer **gyorsítása** és **biztonságosabbá tétele**.



Példa: Google File System



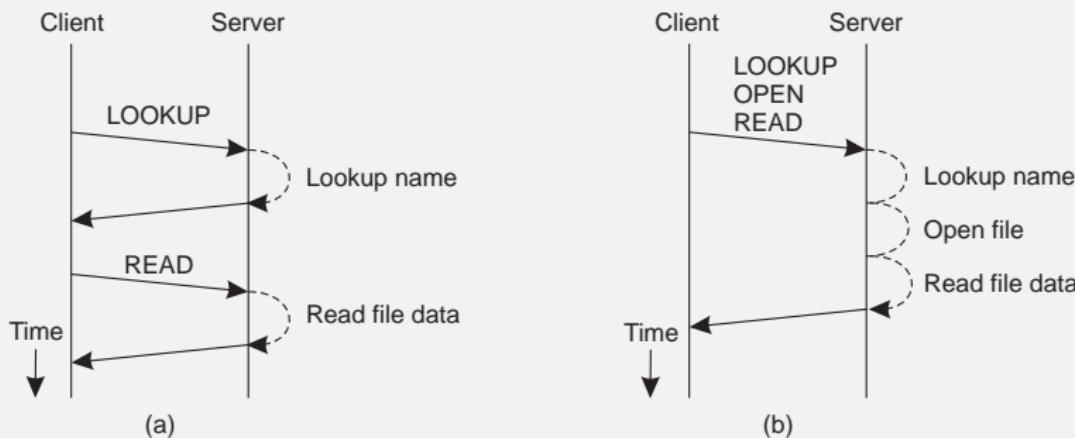
A fájlt 64 MB méretű részekre (chunk) bontjuk, és több szerveren elosztva, replikálva tároljuk.

- A központ (master) csak azt tárolja, melyik szerver melyik részek felelőse \Rightarrow I/O terhelése alacsony
- A szerverek **elsődleges másolaton alapuló** replikációs protokollt használnak; a központot ez **nem** terheli

RPC fájlrendszerben

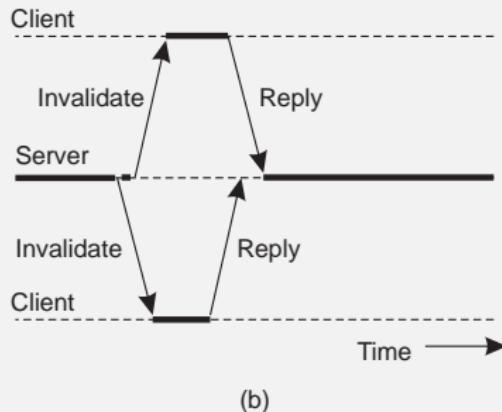
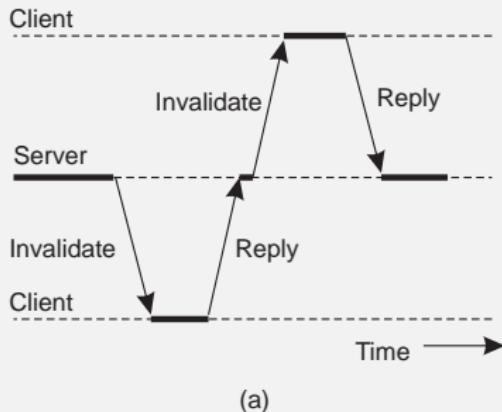
Egy lehetőség távoli fájlrendszer megvalósítására, ha távoli eljáráshívások segítségével végezzük a fájlműveleteket.

Ha a távoli gép elérése költséges, alternatív megoldásokra van szükség, pl. az NFSv4 támogatja több művelet összekombinálását egy hívásba.



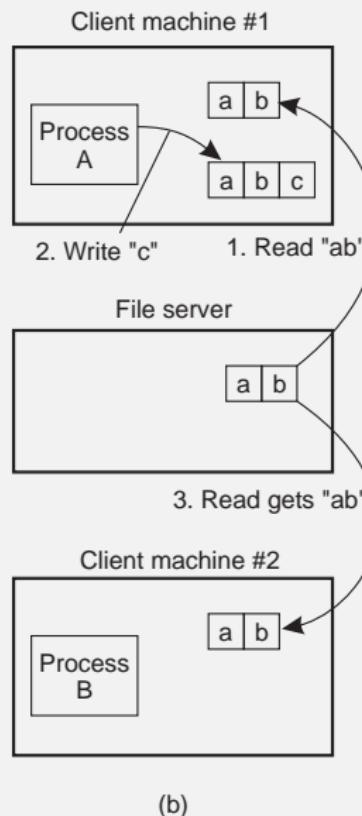
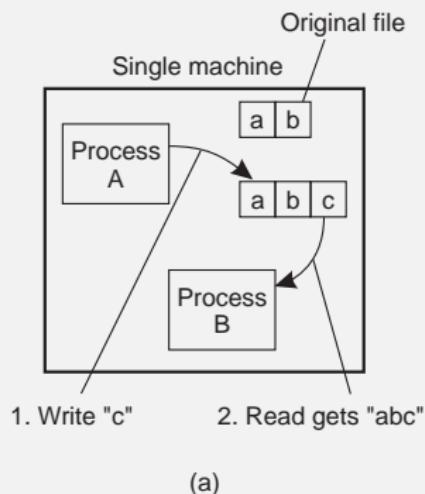
Példa: RPC a Coda fájlrendszerben

Ha replikált fájlokkal dolgozunk (pl. a Coda kliensei cache-elhetik a fájlokat), akkor a kérések sorrendjének kikényszerítése ((a) ábra) túlságosan költséges lehet, mert ha összeomlik egy kliens, akkor csak hosszú timeout után jöhet a következő.



A fájlmegosztás szemantikája

Ha egyszerre több kliens is hozzáférhet egy fájlhoz, a konkurens írási és olvasási műveletek lehetséges végrehajtási sorrendjeit és a kijöhető eredményeket (összefoglalva: a rendszer szemantikáját) rögzíteni kell.



A fájlmegosztás szemantikája

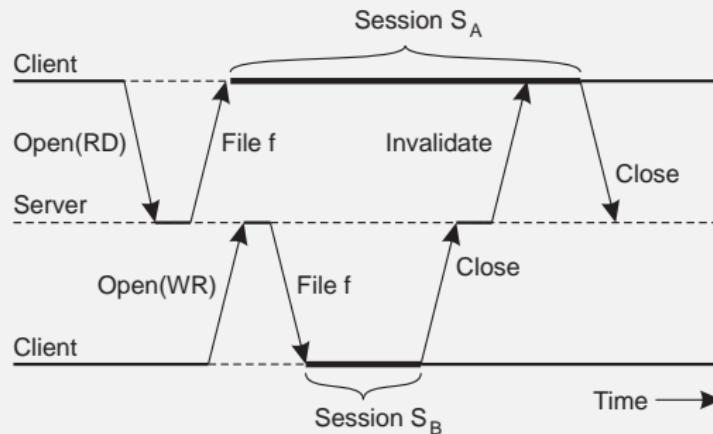
Sokfajta szemantika jöhet szóba.

- **Megváltoztathatatlan fájlok:** a fájlok tartalmát nem lehet módosítani létrehozás után; ez a modell csak ritkán használható
- **UNIX szemantika:** az olvasási műveletek mindig a legutolsó írási művelet eredményét adják ⇒ a fájlból csak egy példányunk lehet az elosztott rendszerben (az előző (a) ábra)
- **Tranzakciós szemantika:** a rendszer minden fájlra külön biztosít tranzakciókat
- **Munkamenet szemantika:** onnantól, hogy a kliens megnyitja a fájlt, odáig, amíg vissza nem írja, az írási és olvasási műveletei csak saját maga számára látszanak (gyakori választás a rendszerekben; az előző (b) ábra)

Példa: fájlmegosztás Coda rendszerben

A Coda fájlrendszer munkamenetei tranzakciós szemantikát valósítanak meg. A megnyitott fájl tartalma átmásolódik a kliensre; ha más módosítja a fájlt, arról a kliens értesítést kap.

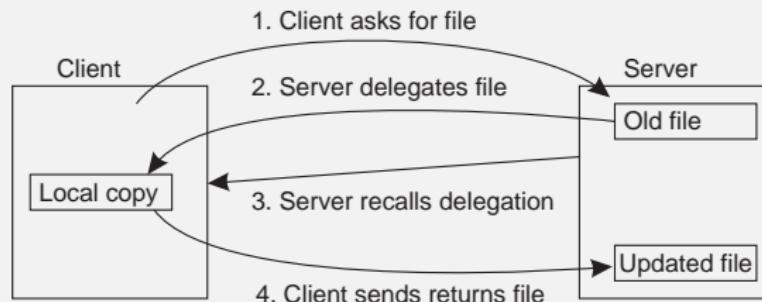
Ha a kliens csak olvas (pl. az ábrán S_A), akkor folytathatja a működését – úgy tekintjük, hogy a tranzakció, amelyet végez, már korábban lezárult.



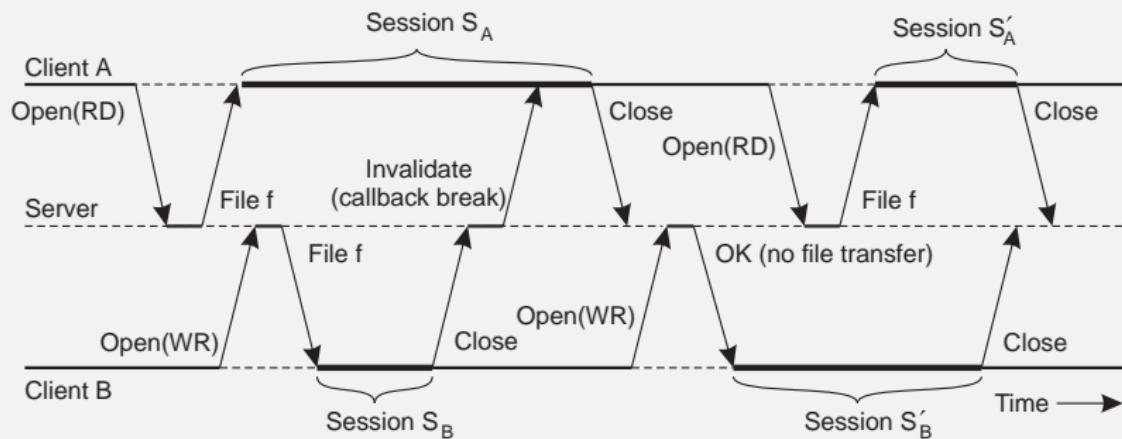
Konzisztencia és replikáció

A modern elosztott fájlrendszerben a **kliensoldali gyorsítótárazás** szerepe főleg a teljesítmény növelése, a **szerveroldali replikáció** célja a hibatűrés biztosítása.

A kliensek tárolhatják a fájlokat (vagy részeket belőlük), és a szerver kiértesíti őket, ha ezt a jogot visszavonja tőlük ⇒ a szerverek általában **állapotteljesek**.



Példa: kliensoldali gyorsítótárazás Coda rendszerben

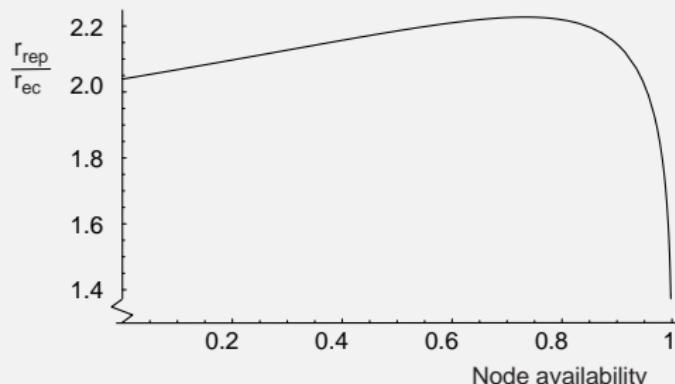


A tranzakciós szemantika segítségével a teljesítmény tovább növelhető.

Rendelkezésre állás növelése P2P rendszerekben

Sok P2P alapú, decentralizált fájlrendszer létezik. Ezekben probléma lehet, ha túl gyorsan változik a tagság (churn), mert kiléphet akár egy fájlt tartalmazó összes csúcs. Ennek kivédéséhez replikálhatjuk a fájljainkat (arányát jelölje r_{rep}).

Másik megközelítés: **erasure coding**: az F fájlt bontsuk m részre, és minden szerverre tegyük n részt, ahol $n > m$. A replikációs arány ekkor $r_{ec} = n/m$. Ez az arány általában sokkal kisebb, mint r_{rep} , ha a rendszerünk változékony.



Elosztott rendszerek: Alapelvek és paradigmák

Distributed Systems: Principles and Paradigms

Maarten van Steen¹ Kitlei Róbert²

¹VU Amsterdam, Dept. Computer Science

²ELTE Informatikai Kar

12. rész: Elosztott webalapú rendszerek

2015. május 24.

Tartalomjegyzék

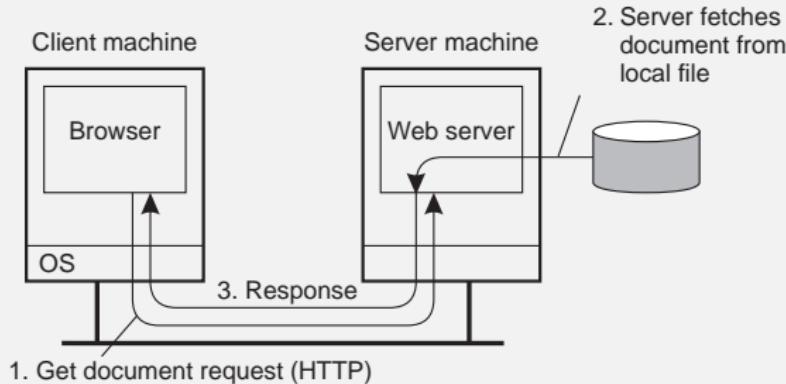
Fejezet
01: Bevezetés
02: Architektúrák
03: Folyamatok
04: Kommunikáció
05: Elnevezési rendszerek
06: Szinkronizáció
07: Konzisztencia & replikáció
08: Hibatűrés
10: Objektumalapú elosztott rendszerek
11: Elosztott fájlrendszerek
12: Elosztott webalapú rendszerek

Elosztott webalapú rendszerek

A WWW (világháló, World Wide Web) olyan szerverek összessége, amelyek HTTP protokollon keresztül különféle tartalmakat szolgálnak ki. A dokumentumokat **hiperhivatkozások** kapcsolják össze.

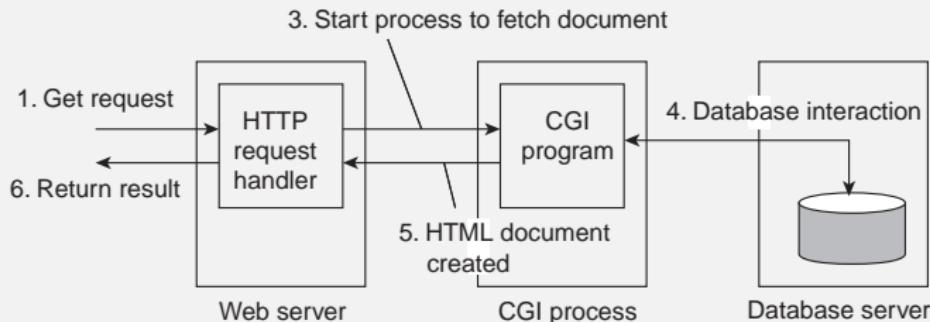
- Sok dokumentum szövegalapú: szövegfájl, HTML, XML
- Egyéb fajták: képek, audio, video, dokumentum (PDF, PS)
- A tartalmak lehetnek a kliensoldalon végrehajthatóak (Javascript)

Jelenleg **kb. 1 milliárd weboldal** létezik, amelyek közül kevesebb mint 200 millió aktív; **egy felmérés szerint** a szerverek népszerűsége: Apache (40%), IIS (29%), nginx (15%), egyéb (16%).



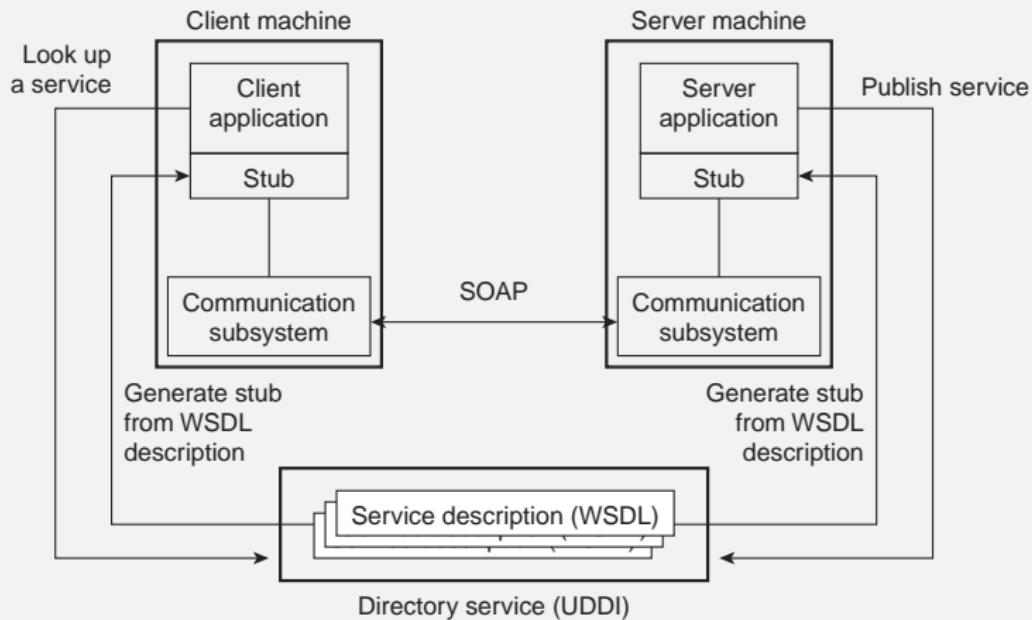
Többrétegű architektúrák

Már a kezdeti kiszolgálók is gyakran három rétegbe tagozódtak.



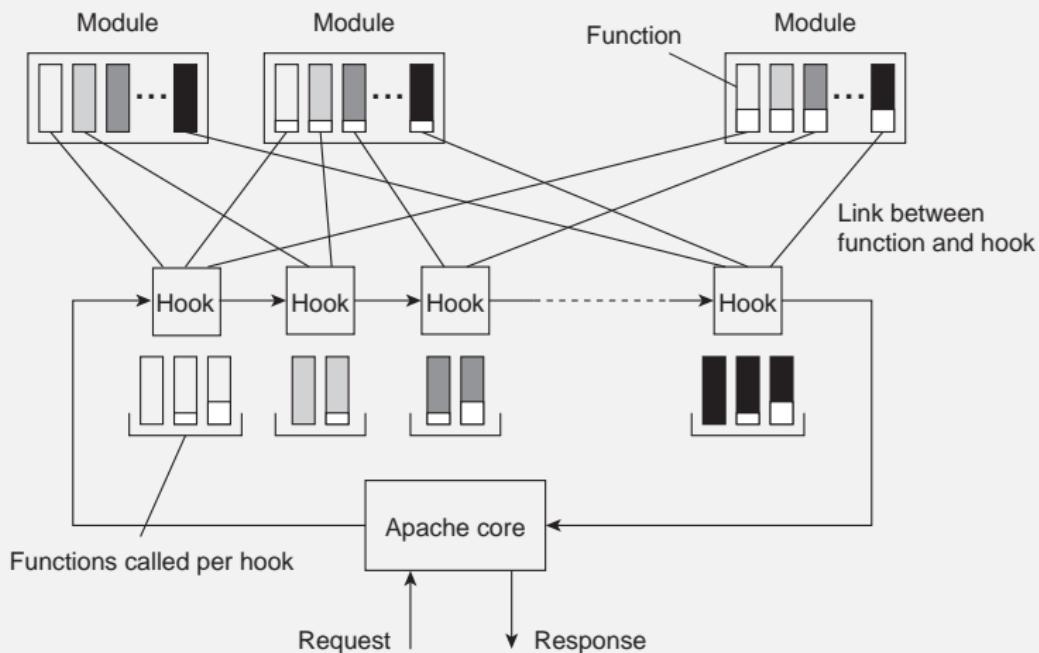
Webszolgáltatások

Felmerült az is, hogy a **felhasználó ↔ weboldal** interakció mellett az oldalak is igénybe vehetnek **szolgáltatásokat** más oldalakról ⇒ fontos, hogy a szolgáltatások **szabványosak** legyenek.



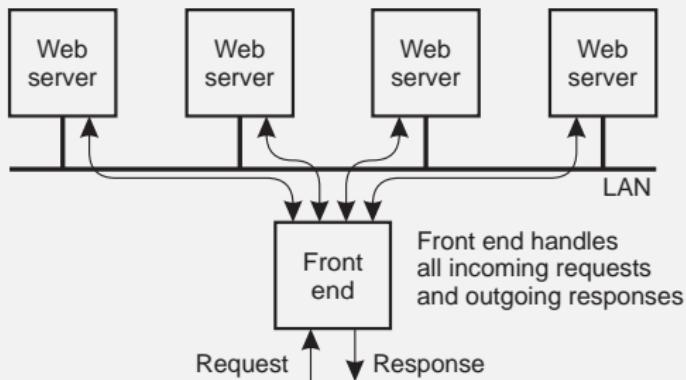
Webszerverek

A szerver szerkezetét a tartalmak kiszolgálásának menete szabja meg. A szerverekbe beépülő modulok telepíthetők, amelyek a kiszolgálás egyes fázisaiban aktivizálódnak.



Szerverfürtök

A teljesítmény és a rendelkezésre állás növelésének érdekében a szerverek sokszor (a felhasználó számára átlátszó módon) többszörözve vannak.

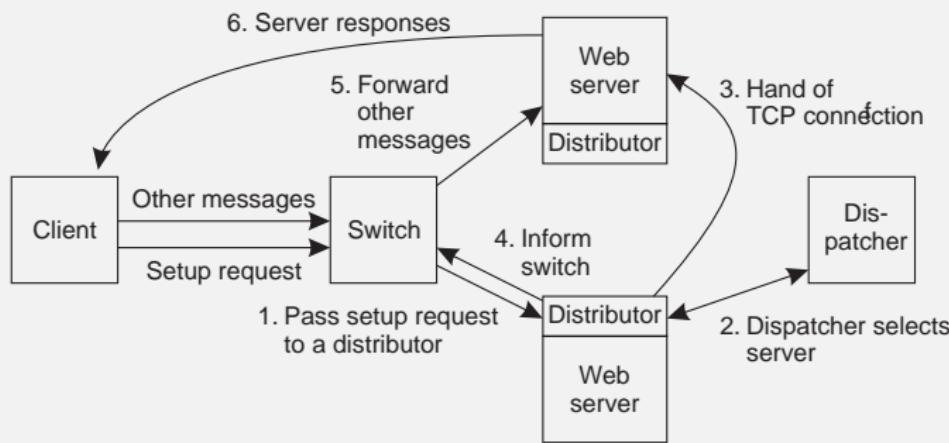


A kapcsolattartó (front end) szűk keresztmetszetté válhat, ennek elkerülésére több lehetőség van.

- **TCP átadás:** Valamilyen metrika alapján kiválasztunk egy szervert, és a kliens kiszolgálását az a szerver folytatja.
- **Tartalomérzékeny kéréselosztás (content aware distribution):** Lásd következő oldal.

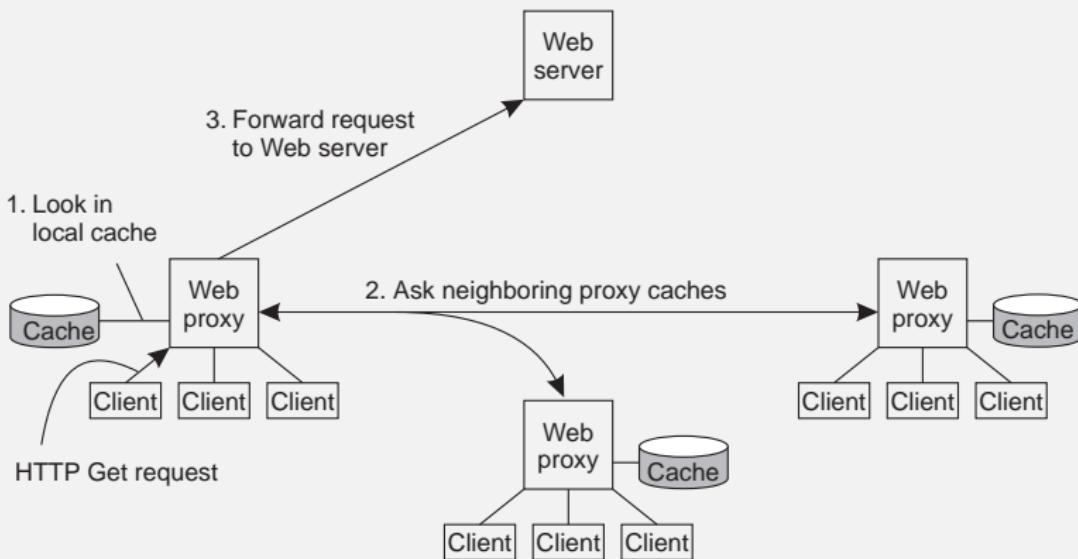
Szerverfürtök

Tartalomérzékeny kérésselosztás (content aware distribution): A HTTP kérés tartalmát is figyelembe vesszük a szerver kiválasztásánál. Ez megnöveli a kapcsolattartó terhelését, de sok előnye van: segítségével hatékonyabb lehet a szerveroldali cache-elés, és lehetnek bizonyos feladatokra dedikált szervereink.



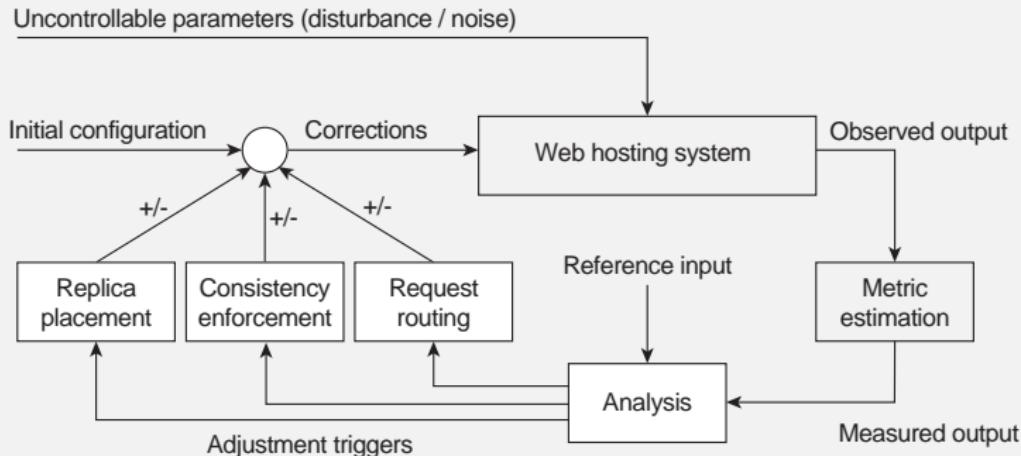
Webhelyettes

A kimenő kapcsolatok kezelésére **webhelyetteseket** (web proxy) telepíthetünk. Ezek cache-lik a kiszolgált tartalmakat; csak akkor fordulnak a szerverekhez, ha sem náluk, sem a többi helyettesnél nincsen meg a kért tartalom.



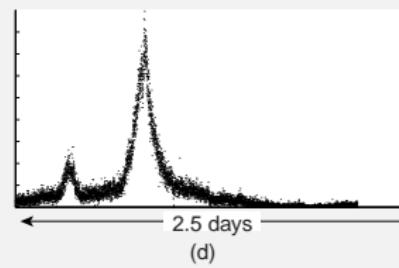
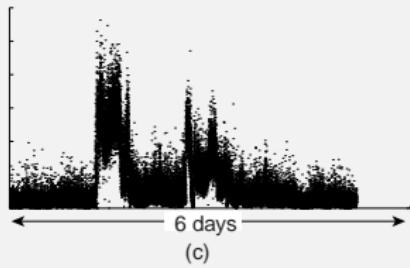
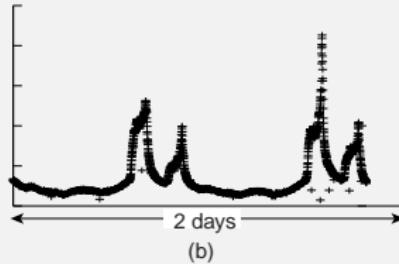
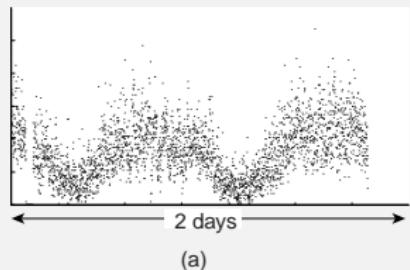
Replikáció webkiszolgálókban

A replikáció célja a teljesítmény növelése. A rendszer paraméterei (hová célszerű a replikátumokat elhelyezni, konzisztencia megkövetelt erőssége, kérések útvonalválasztása) változóak lehetnek, ezeket célszerű **önszabályozással** beállítani.



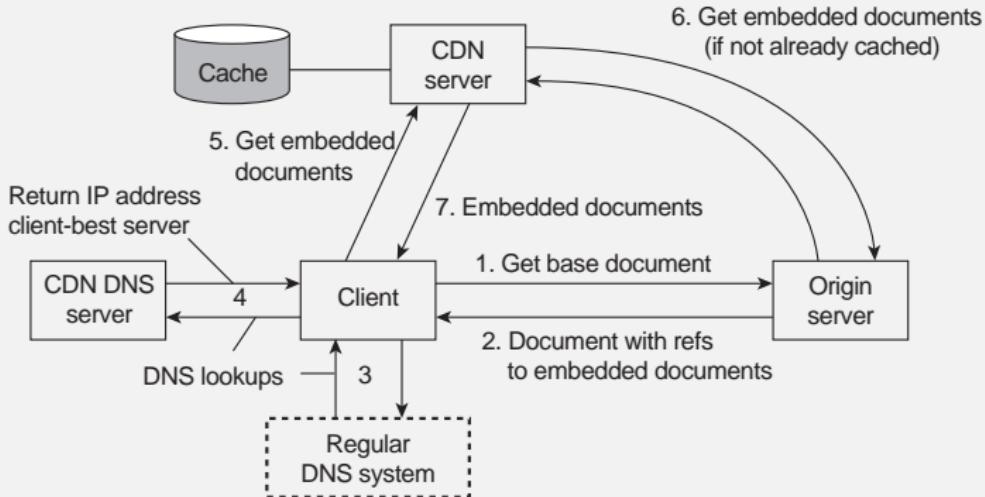
Hirtelen terhelés

A terhelés néha ugrásszerűen megemelkedik (flash crowd, flash mob), ezt még akkor sem könnyű kezelni, ha az erőforráskezelés dinamikus.



Szerveroldali replikáció

A tartalomkézbesítő hálózatok (Content Delivery Network, CDN) nagy teljesítményű és rendelkezésre állású elosztott rendszerek, amelyeknek célja dokumentumok hatékony kiszolgálása.



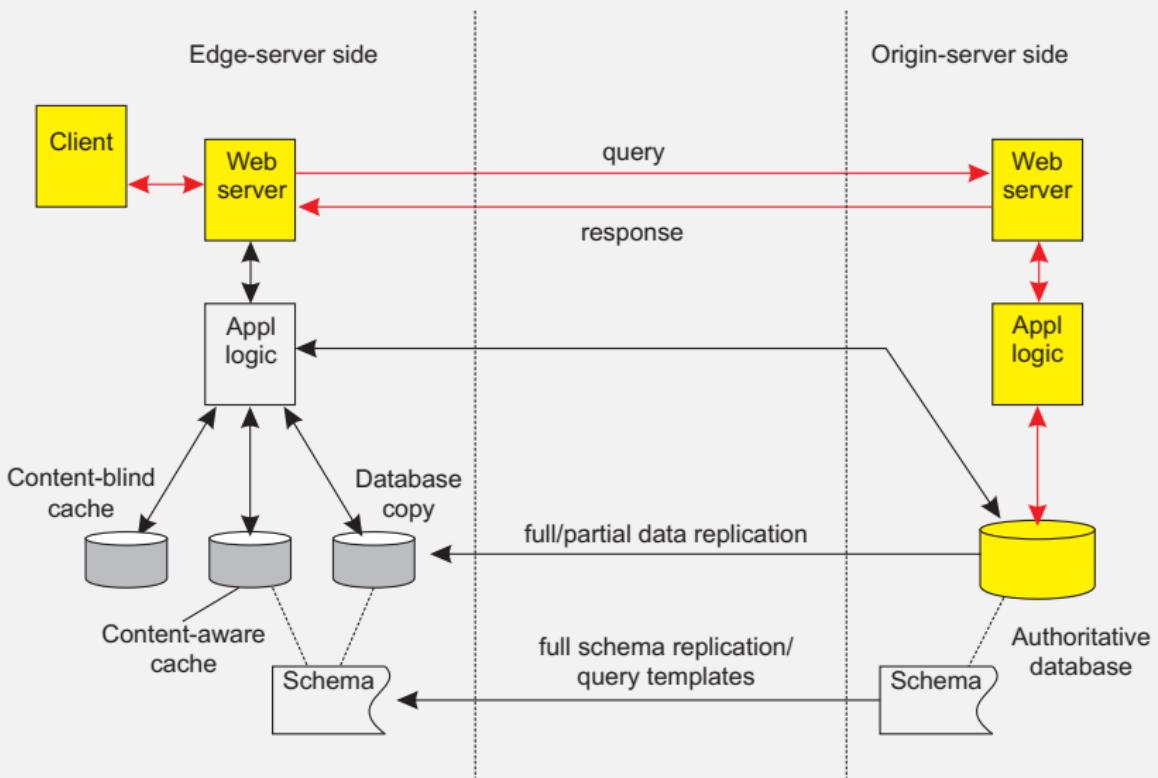
Replikáció webalkalmazásokban

Ha a CDN tárolt adataiban változás következik be, ez először az eredetszerveren jelenik meg. A változásokat el kell juttatni a CDN szerverekhez; ennek a célszerű módja a rendszer jellegétől függ.

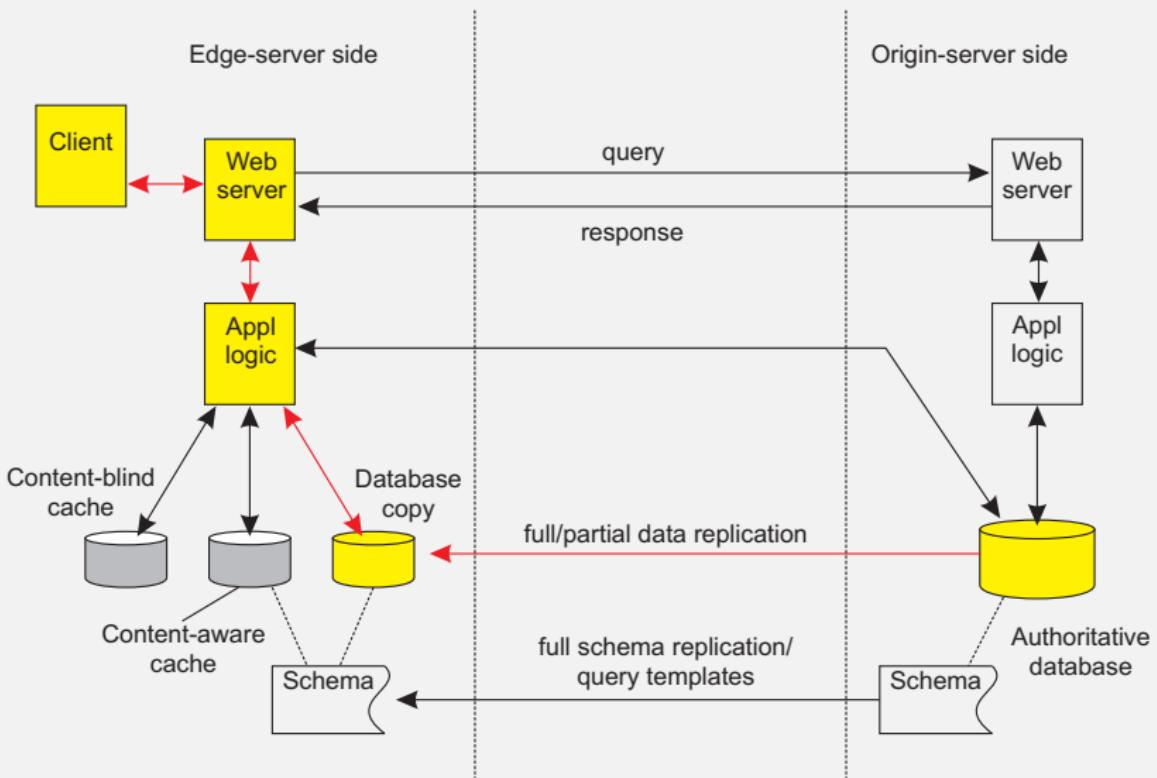
- **Teljes replikáció:** sok olvasás, kevés írás, **összetett** lekérdezések
- **Részleges replikáció:** sok olv., kevés írás, **egyszerű** lekérdezések
- **Tartalom szerinti gyorsítótárazás:** Az adatbázist az edge szerver módosított, a lekérdezésekhez illeszkedő alakban tárolja helyben, és feliratkozik a szerveren a frissítésekre. Jól működik **intervallumokra vonatkozó, összetett** lekérdezésekre.
- **Eredmények gyorsítótárazása:** Az edge szerver a korábbi lekérdezések eredményeit tárolja el. Jól működik **egyszerű** lekérdezésekre, amelyek **egyedi adatokra** (nem intervallumokra) vonatkoznak.

Ha az írások számaránya megnő, akkor a replikáció akár ronthatja is a rendszer teljesítményét.

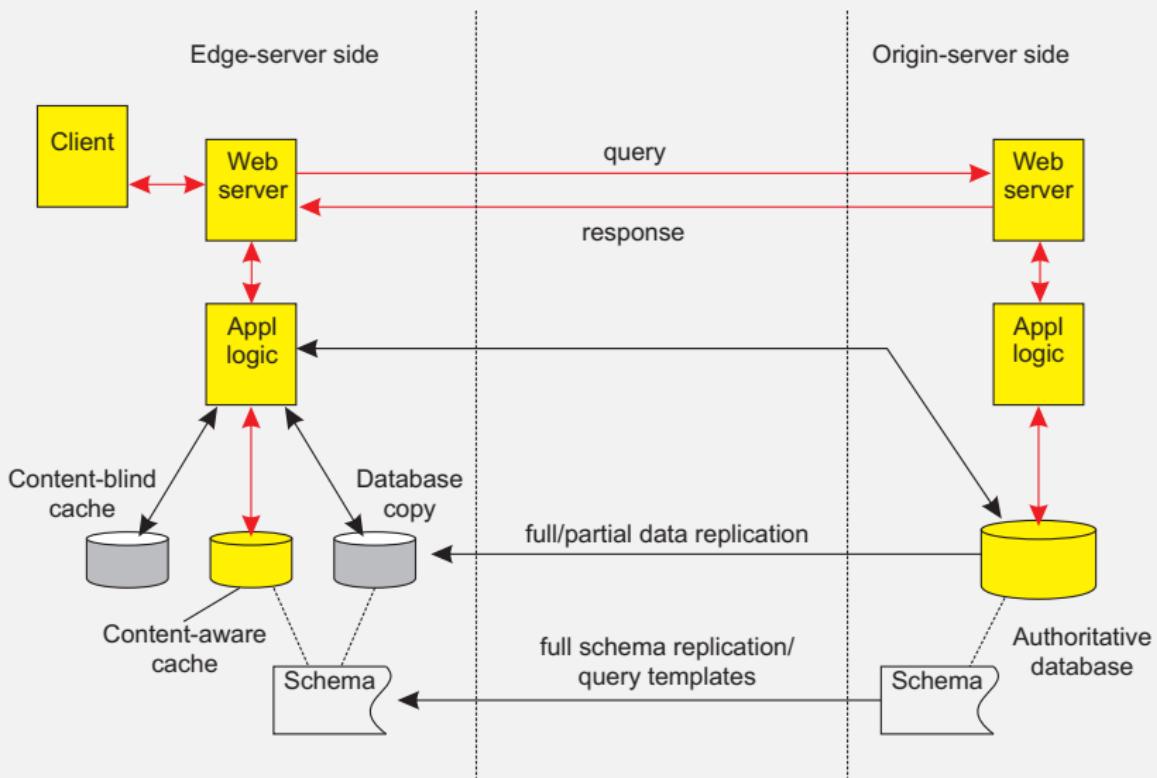
Replikáció webalkalmazásokban: nincsen replikáció



Webalk. replikációja: részleges/teljes replikáció



Webalk. replikációja: tartalom szerinti gyorsítótárazás



Webalk. replikációja: eredmények gyorsítótárazása

