Elosztott Hash Táblák

Jelasity Márk

Motiváció

- Nagyméretű hálózatos elosztott rendszerek az Interneten egyre fontosabbak
  - Fájlcserélő rendszerek (BitTorrent, stb), Grid, Felhő, Gigantikus adatközpontok, Botnetek
- Ezek számára speciális algoritmusokat kell kifejleszteni!
  - Koordináció, terheléselosztás, adattárolás és visszakeresés, adatelemzés, stb.

Rendszermodell

- Elvonatkoztatunk az előbbi konkrét rendszerektől, és az esszenciájukra koncentrálunk
- „hagyományos” algoritmus kurzusokon is van rendszermodell
  - Központi CPU, RAM, stb.
  - Még itt is figyelni kell: pl. virtuális memória esetén más a praktikus időkomplexitás
- Esetünkben a rendszermodell kulcsfontosságú

Rendszermodell

- Nagyon sok CPU, amelyek saját memóriával rendelkeznek (komplett számítógépek)
  - Ezentúl csomópontnak hívjuk őket (node)
- Csomagkapcsolt hálózat segítségével kommunikálnak
  - Minden csomópont hálózati címmel rendelkezik
  - A cím ismeretében a csomópont számára üzenet küldhető bármely másik csomópontból
- Az üzenetek a hálózatban késhetnek, és el is veszhetnek, sorrendjük sem garantált

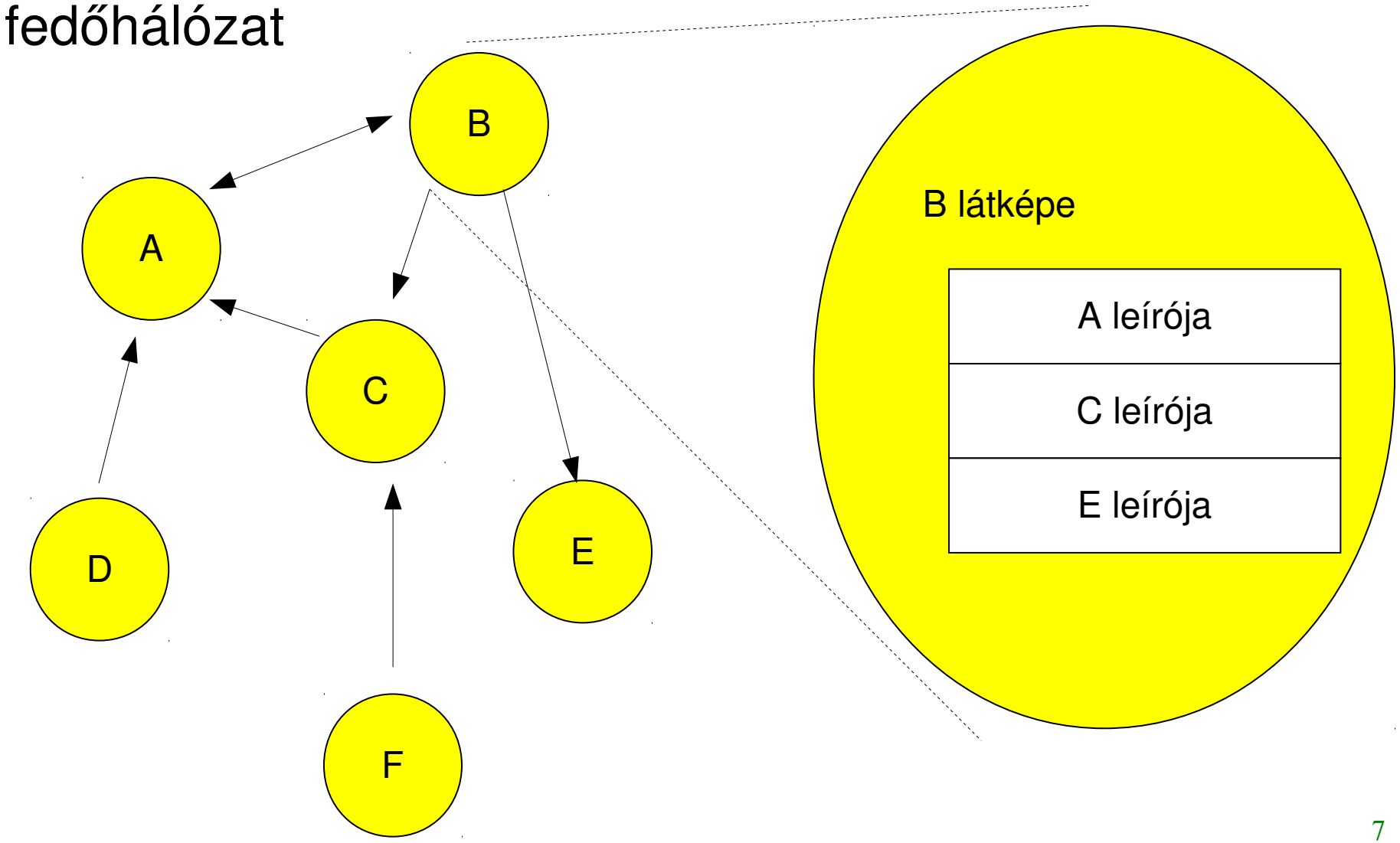
Rendszermodell

- A csomópontok bármikor távozhatnak a rendszerből figyelmeztetés nélkül
- Bármikor csatlakozhat új csomópont a rendszerhez
- A csomópontok száma akár milliós nagyságrendű is lehet
- Egyszóval, óriási, extrém módon dinamikus és elég megbízhatatlan komponensekből álló rendszer
- Viszont óriási számítási-, tároló-, és kommunikációs kapacitással

Fedőhálózatok (overlay networks)

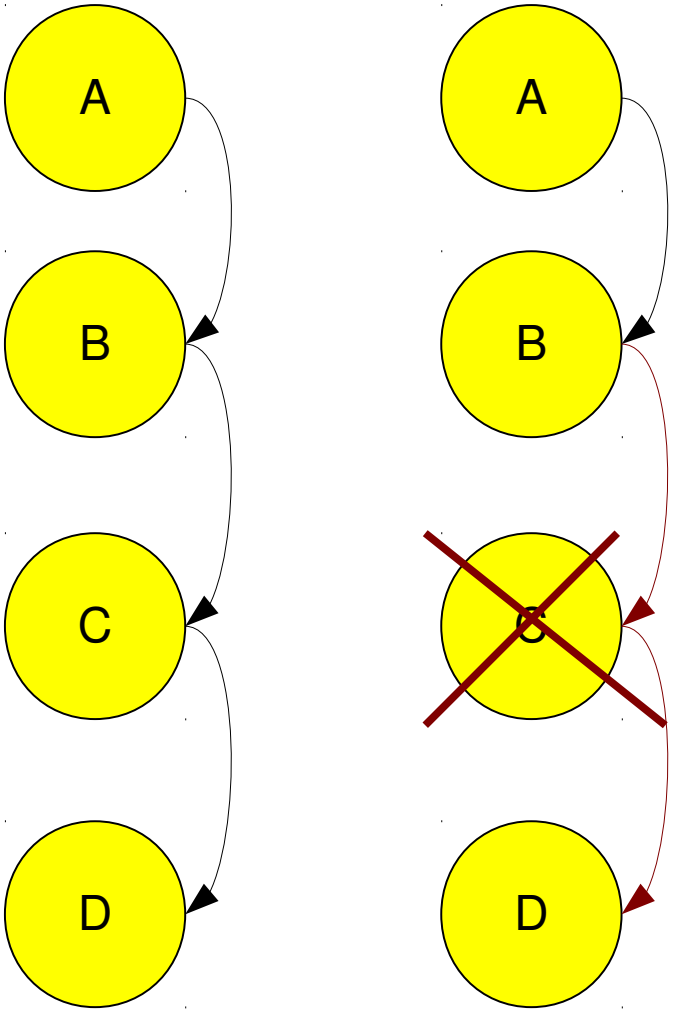
- Szeretnénk egységes, megbízható, hatékony rendszerré kovácsolni ezt a kaotikus rendszert
- Egy megoldás: az alkalmazási rétegben lakó protokollokkal un. fedőhálózatot alakítunk ki
  - Virtuális hálózat, amit az ismeretségi reláció definiál
  - Független a fizikai hálózattól
- Ennek a segítségével pl. elosztott adatszerkezeteket építhetünk, amik egy egyszerű interfészen keresztül kezelhetők; pl. hash tábla.

Fedőhálózat



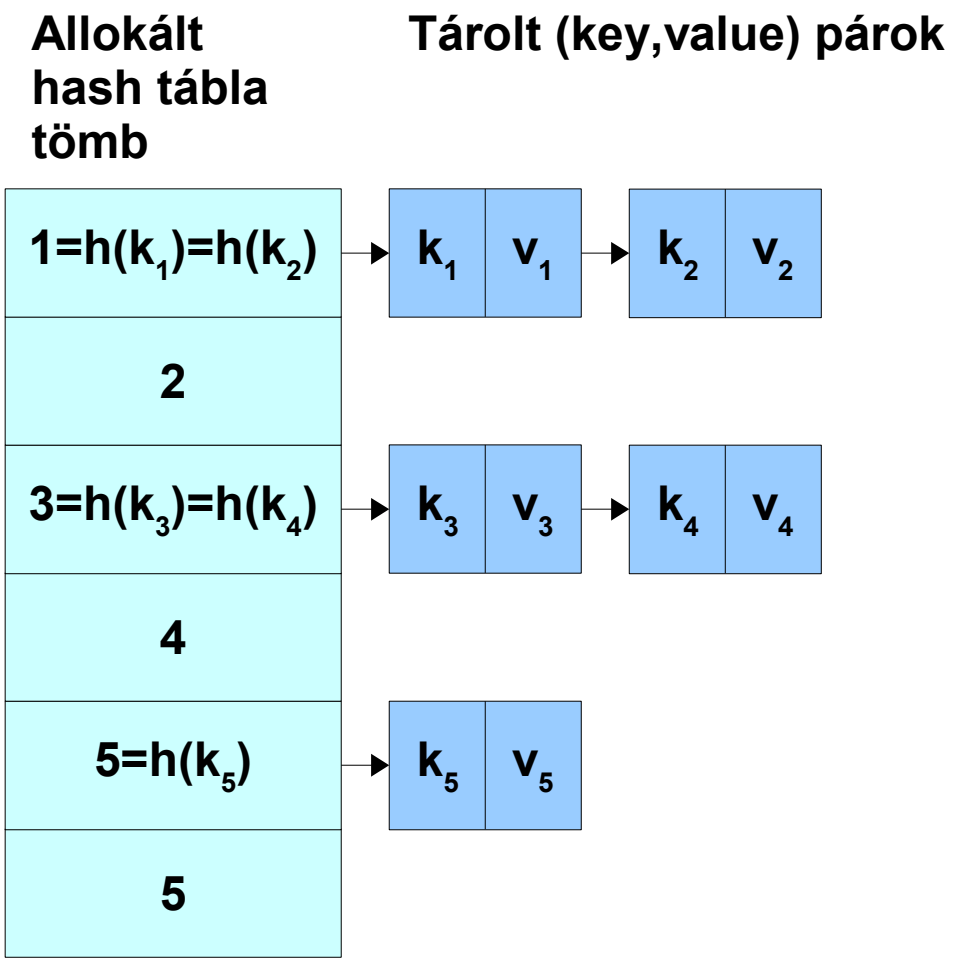
Elosztott adatszerkezetek

- Kb. mint a linkelt adatszerkezetek, de speciális problémák
- Dinamizmust kezelni kell, és nem minden szerkezet valósítható meg egyszerűen, pl. láncok, fák problémásak (de nem lehetetlen!)



Hash táblák

- Kulcsok és szatellit adatok tárolása
  - **put(key,value)**
  - **value = get(key)**
  - **remove(key)**
- A h() hash függvény a cella indexét adja meg
- A (key,value) pár pl. az adott cellából linkelt listába kerülhet



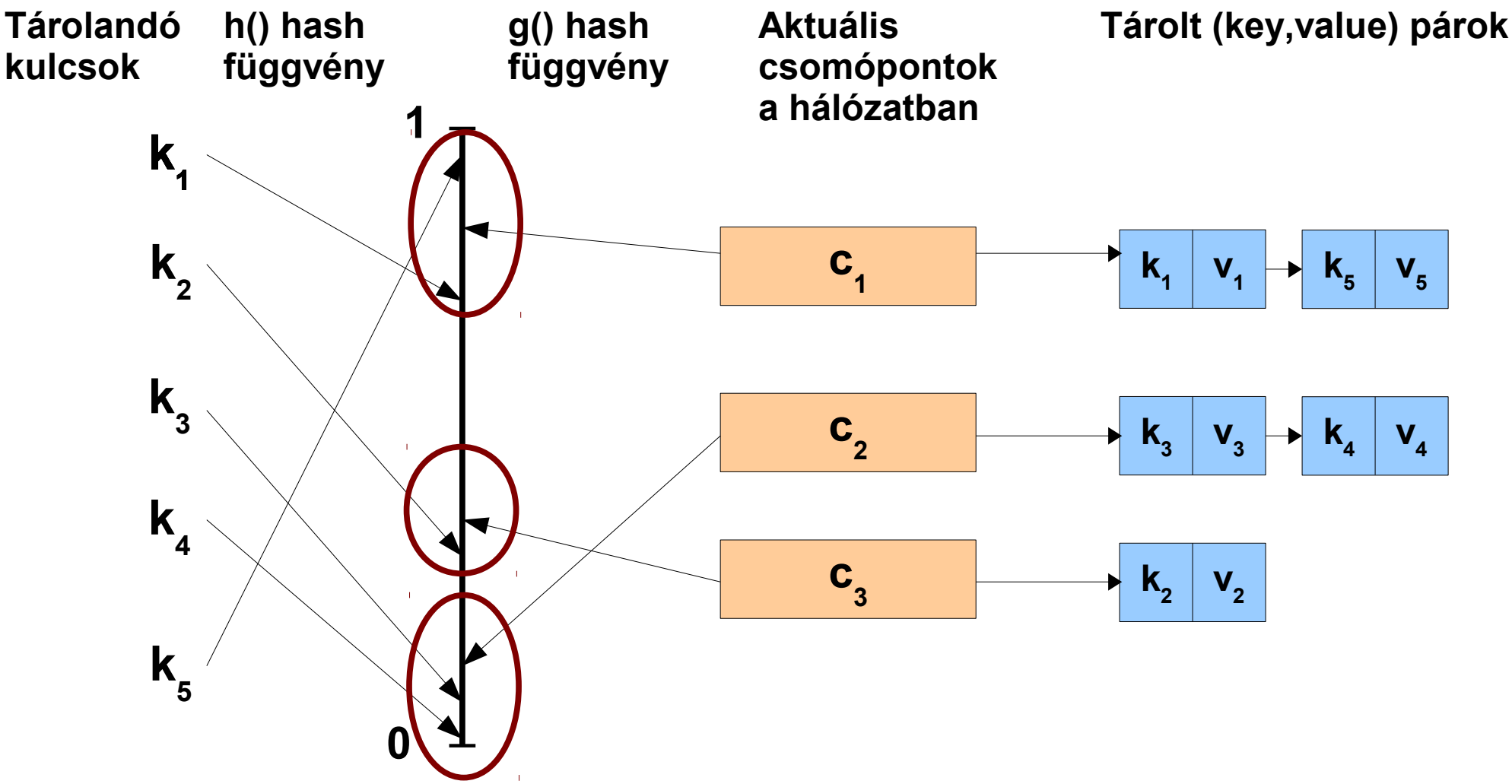
Miért kell hash tábla?

- Ideális esetben N kulcsot O(N) időben tudunk O(N) méretű helyre eltárolni, és O(1) időben elérni minden kulcsot.
- Tehát gyorsan tudunk keresni tetszőleges kulcsokra (bár a szatellit adatok attribútumaira nem tudunk gyorsan keresni!)
  - A kulcs lehet pl. fájlnév, az adat maga a fájl, vagy esetleg egy link a fájlra.
- A cél ugyanezt elérni az elosztott rendszerünkben

Első lépés: konzisztens hash-elés

- A hash táblákban a tömb átméretezése nagyon költséges, mert majdnem minden kulcs új cellába kerül
- Konzisztens hash-elés: olyan hash tábla implementáció, ahol egyedi cellák hozzáadása vagy törlése olcsó művelet
  - Törlés esetén csak a törölt cella elemeit kell áthelyezni
  - Hozzáadás esetén régi cellák között nincs adatforgalom (monotonitás)

Konzisztens hash-elés: egy példa

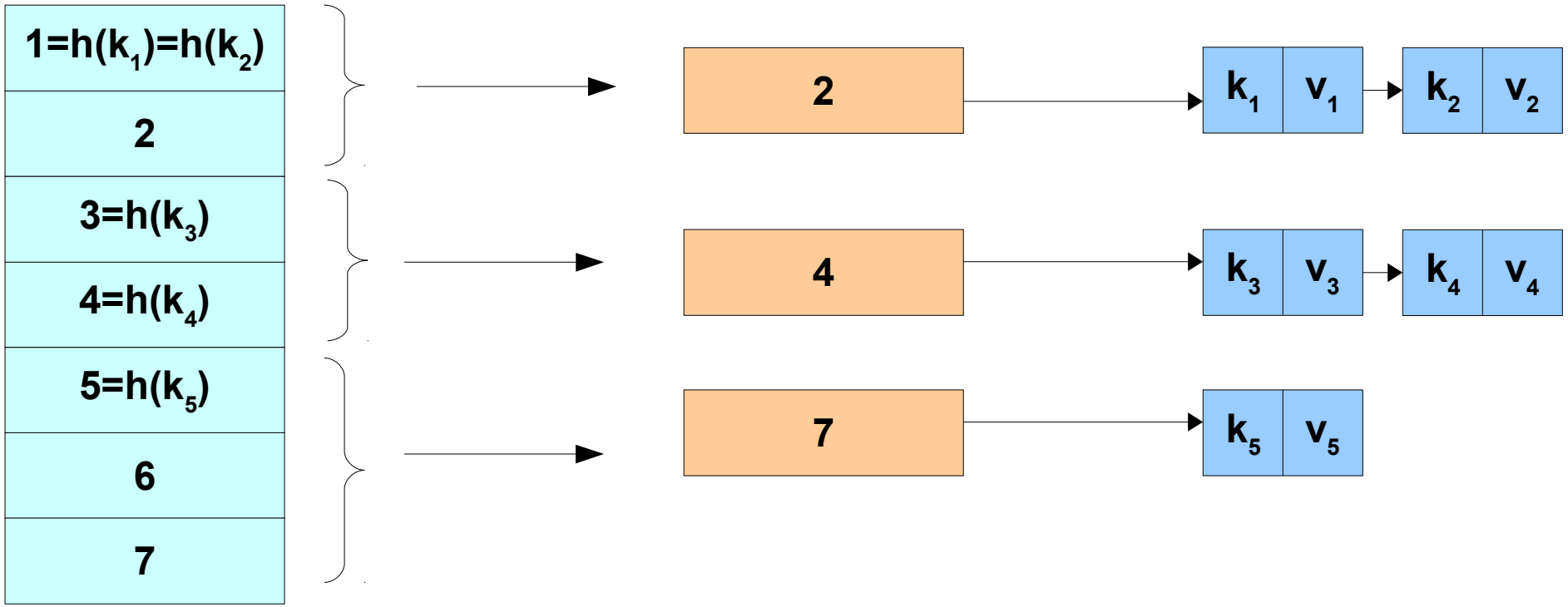


Konzisztens hash-elés még egyszer

Absztrakt „allokált tömb”, más néven azonosító (ID) tér

Aktuális csomópontok a hálózatban (g() szerint rendezve)

Tárolt (key,value) párok



Konzisztens hash-elés és fedőhálózatok

- Tehát, minden esetben
  - Egy k kulcshoz kiszámoljuk a h(k) hash értéket
  - A konzisztens hash tábla definíciója ehhez hozzárendel egy éppen aktív csomópontot
  - Ezt a csomópontot meg kell találni!
- Ezt pedig egy speciális fedőhálózat segítségével csináljuk, amelyben egy alkalmas útvonalválasztó algoritmus a megfelelő csomóponthoz vezet

Elosztott hash táblák

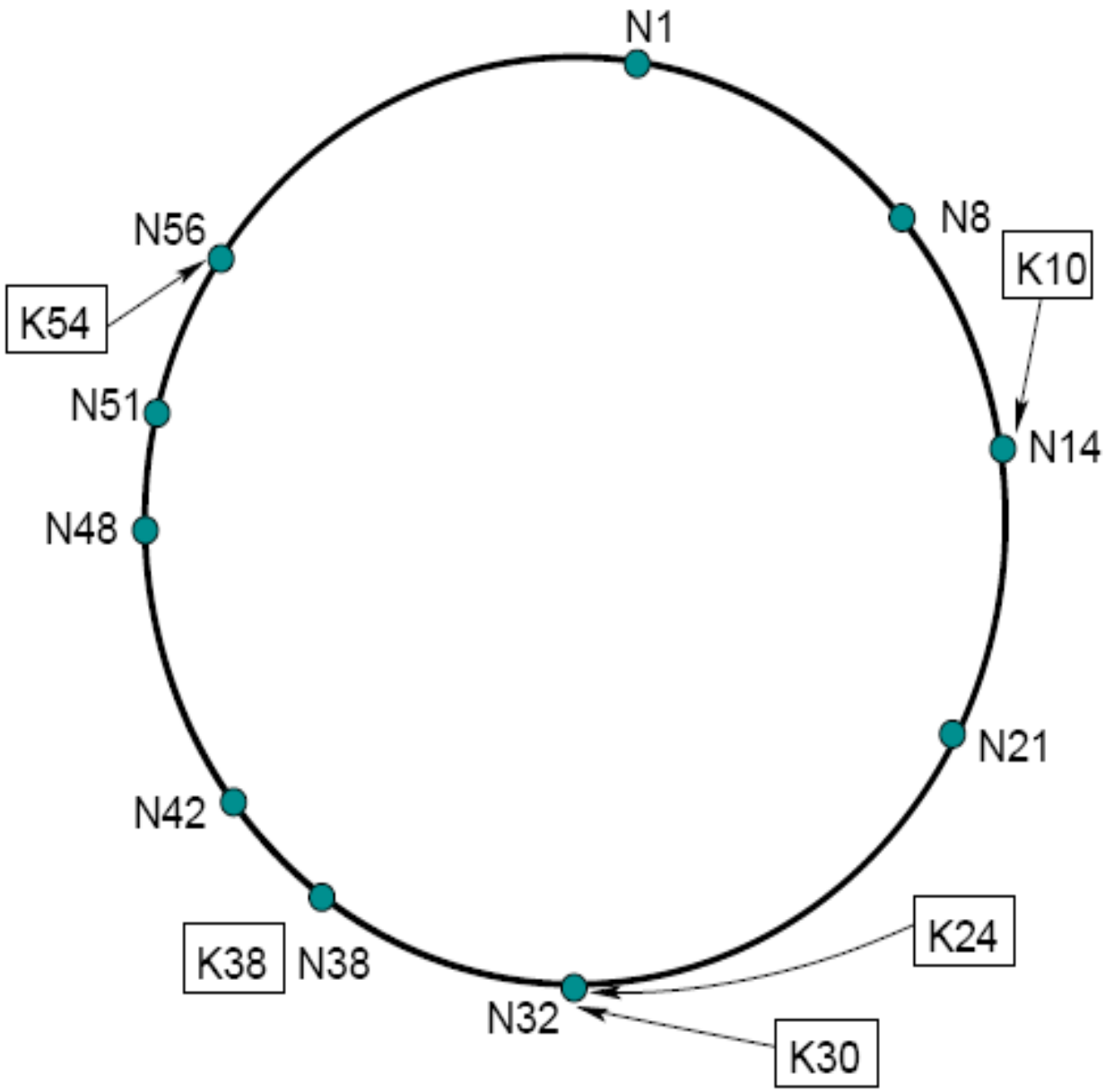
- Rengeteg variáció létezik: sokfajta
  - Konzisztens hash-elés
  - Fedőhálózat
    - **Ennek konkrét felépítő és fenntartó algoritmusai**
  - Útvonalválasztás
- Ezek különböznek több szempontból
  - Útvonalválasztás átlagos üzenetkomplexitása
  - Fedőhálózat egy csomópontjának tárkomplexitása
  - Robosztusság, megbízhatóság, egyszerűség, stb

Chord

- Az egyik első, és egyben legidézettebb elosztott hash tábla: első cikkre 8500+ idézet a google scholar szerint!
- Előnyei:
  - Egyszerű
  - Üzenet- és tárkomplexitás egyaránt O(log N), ahol N a hálózat mérete
- A konzisztens hash-elést egy rendezett gyűrűt formáló fedőhálózat implementálja

A Chord gyűrű

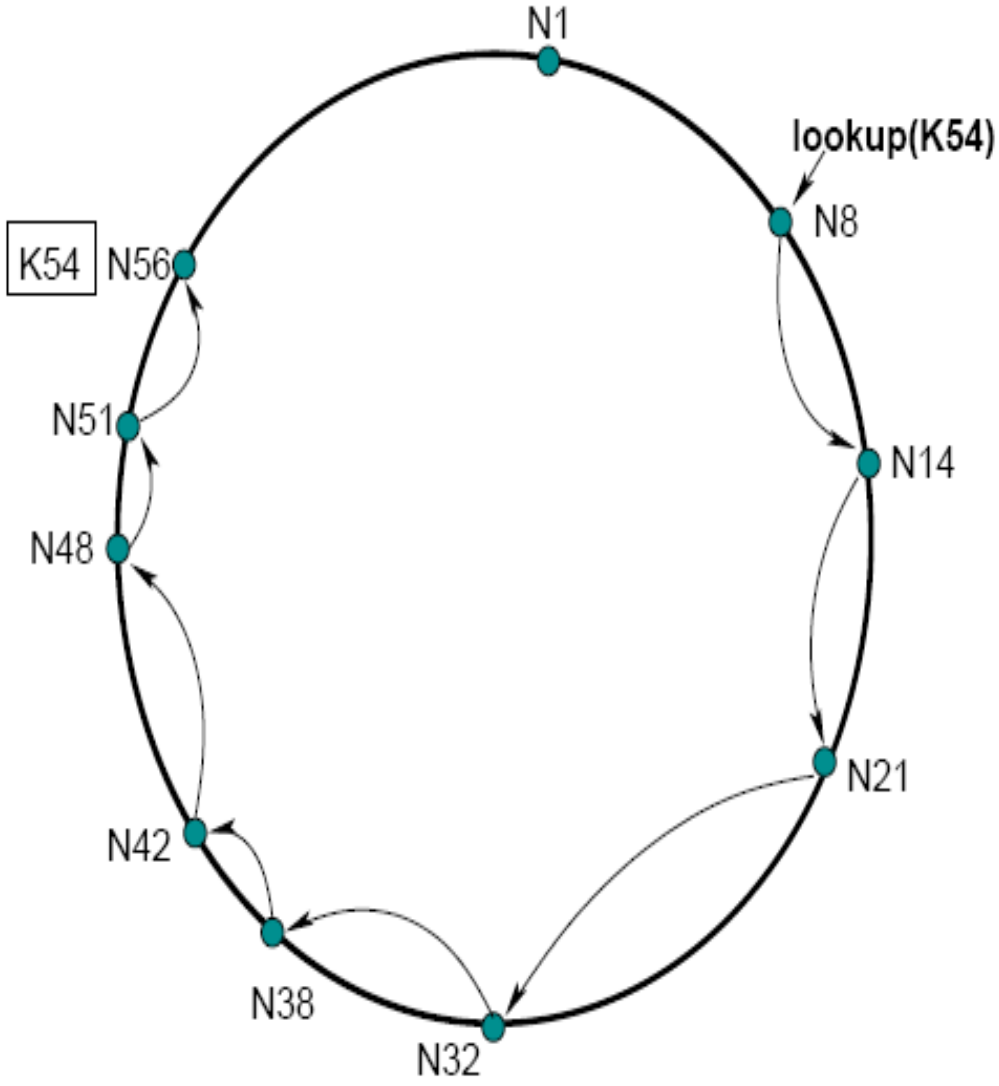
- Azonosító gyűrű
  - 10 csomópont
  - 5 kulcs
- (Az azonosító egy 160 bites szám, az SHA-1 hash függvényt használja)
- Adott hash-kódhoz a rákövetkező csomópontot rendeljük



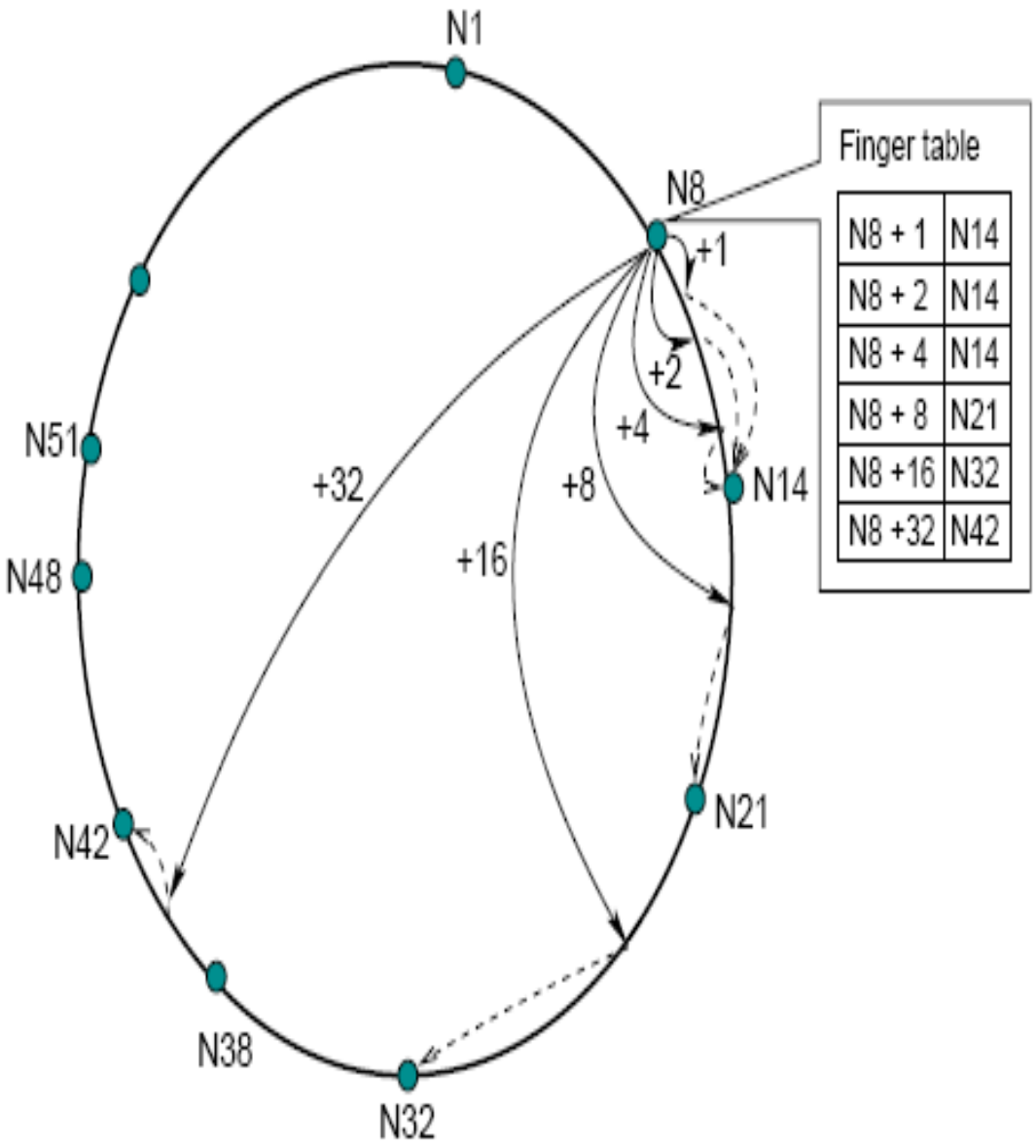
Egy primitív visszakereső algoritmus

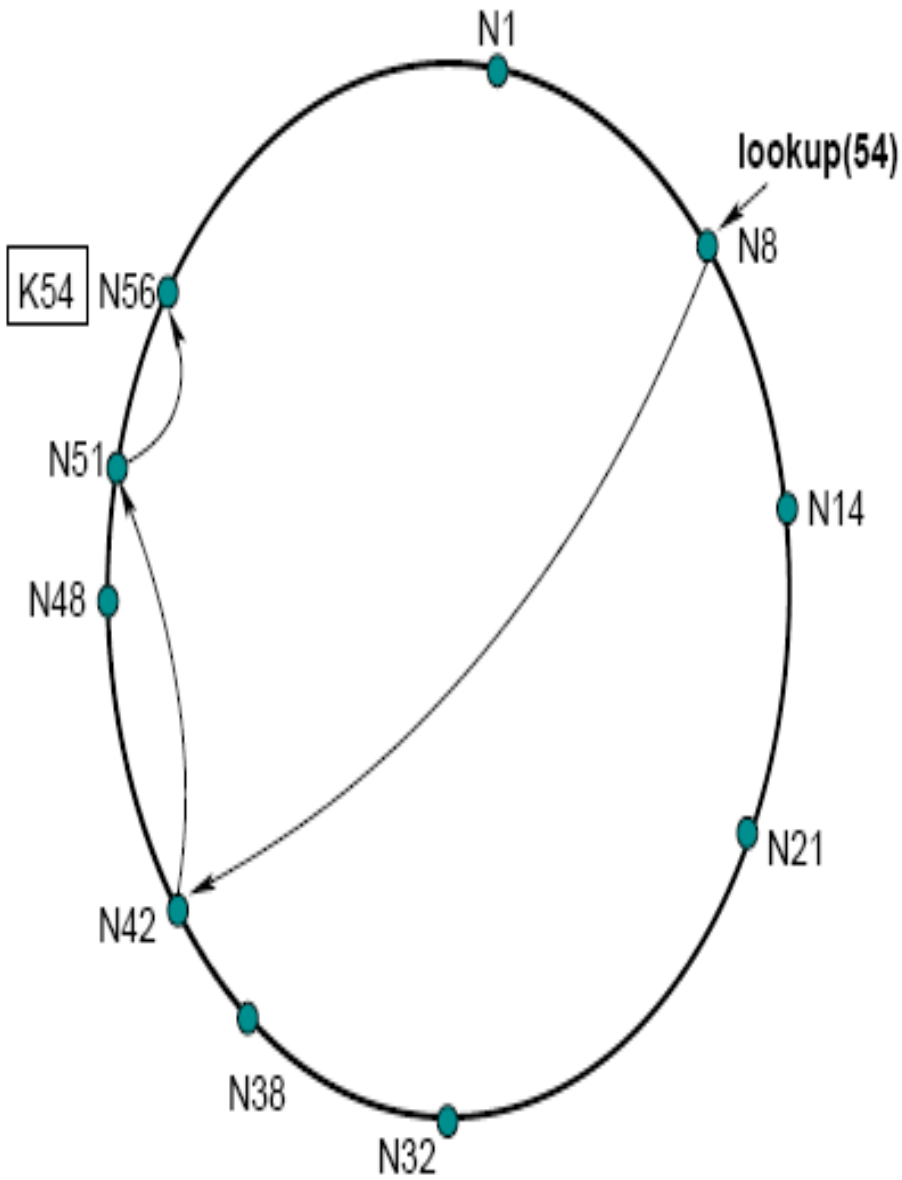
- A feladat tehát a rákövetkező pont megtalálása adott kulcshoz
- Pl. mehetünk körbe...
- Ez O(N) üzenet: túl sok

```
// ask node n to find the successor of id
n.find_successor(id)
  if (id ∈ (n, successor])
    return successor;
  else
    // forward the query around the circle
    return successor.find_successor(id);
```



Egy hatékony visszakereső algoritmus





Egy hatékony visszakereső algoritmus

```
// ask node n to find the successor of id
n.find_successor(id)
  if (id ∈ (n, successor])
    return successor;
  else
    n'=closest_preceeding_node(id)
    return n'.find_successor(id)

// ask node n to find the predecessor of id
n.closest_preceeding_node(id)
  for i = m downto 1
    if (finger[i] ∈ (n,id)) return finger[i]
  return n'
```

- A leközelebbi megelőző fingerre ugrunk, amíg elérjük a célt
- Itt feltesszük, hogy távoli eljáráshívással működik, de lehet mobil ágens szerű is

Komplexitás

- Várhatóan O(log N) szomszéd kell minden csomóponton:
  - A finger táblában definíció szerint $\log_2 M$ sor van, ahol M az azonosító tér mérete (pl 160 bites azonosítóknál a tábla max 160 sorból áll)
  - ezen kívül O(1) szomszéd a gyűrűben
  - Ez azt jelenti, hogy várhatóan O(log N) másik csomópontról tárolunk információt, hiszen az első log M/N finger várhatóan csak O(1) különböző csomópontra mutathat

Komplexitás

- O(log N) üzenettovábbítás minden visszakeresésnél nagy valószínűséggel (with high probability, w.h.p.):
  - A távolság a céltól a gyűrű mentén minden lépésben legalább feleződik
  - tehát $\log_2$ N lépés után a távolság maximum M/N (ahol M a gyűrű kerülete), de egy M/N hosszú intervallumban már csak
    - **O(1) csomópont van várható értékben**
    - **Max O(log N) csomópont van w.h.p.**
  - mivel a csomópontok azonosítója véletlen

Csomópontok belépése és hibák

- Egy új csomópontnak
  - Meg kell találnia a megelőző, rákövetkező, és finger szomszédait
  - Értesítenie kell azon csomópontokat, amelyek számára ő megelőző, rákövetkező, vagy finger szomszéd lehet
- Lézezik protokoll, ami ezt O(log N) időben megoldja, de ez elég komplikált
- Létezik egy „nyugis”, egyszerű protokoll, ami folyamatosan fut, és a hibajavításért is felelős egyúttal

Csatlakozás: egy nyugis megoldás

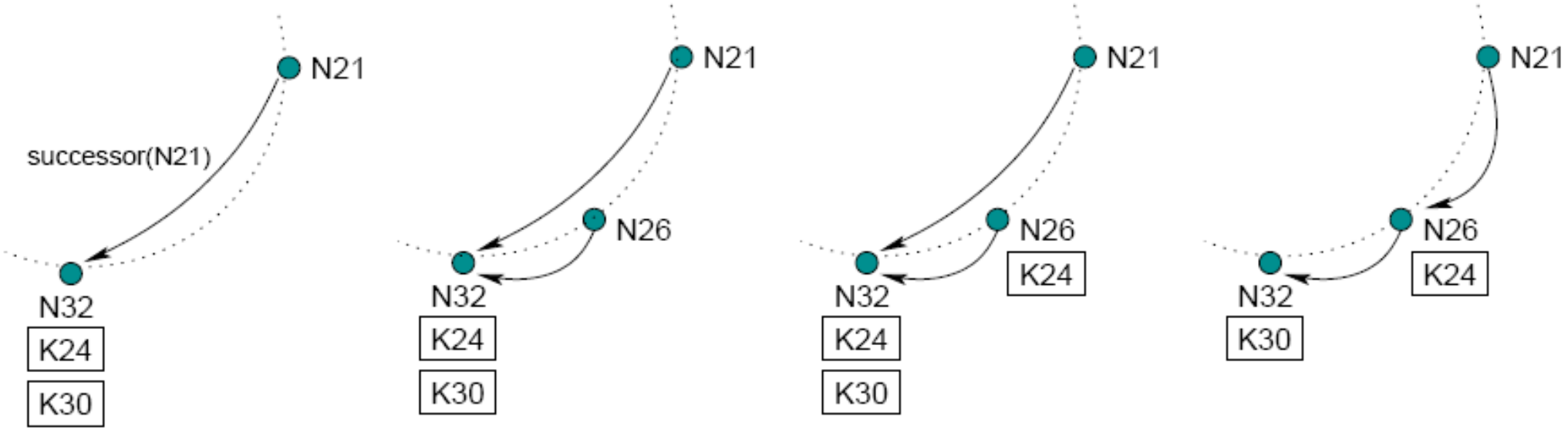
- Ha a gyűrű helyes, akkor a működés már megfelelő: a fingerek „csak” a sebességhez kellenek
- Stabilizáció
  - Minden csomópont szabályos időközönként futtatja
  - Minen csomópont szabályos időközönként meghívja a find_successor($n+2^{i-1}$) függvényt is egy véletlen i-re
  - Minden hívás költsége O(log N) csomópontonként

```
n.join(n')
  predecessor = nil;
  sucessor =
     n'.find_successor(n);
```

```
n.stabilize()
  x = sucessor.predecessor;
  if (x ∈ (n, successor) )
     successor = x;
     successor.notify(n);
```

Csatlakozás: egy nyugis megoldás

- Csatlakozás: találd meg a rákövetkezőt, aztán stabilizálj
  - Mivel a gyűrű gyorsan kiegészül, az útvonalválasztás működik
  - Az útvonalválasztás O(log N) idejű marad, ha a fingerek megtalálása gyorsabban végbemegy, mint ahogy a hálózat mérete megduplázódik



Hibák kezelése

- A hibás (kieső) csomópontok kezelése
  - Replikáció: egy rákövetkező helyett helyett r rákövetkező szomszédot tárolunk
    - **Robusztus a hibákra, mert ha a rákövetkező szomszéd kiesik, könnyen megtaláljuk az újat**
  - Altenratív útvonalak
    - **Ha egy finger nem válaszol, válasszuk az előző fingert, vagy a replikált rákövetkező szomszédot, ha már közel vagyunk**
- Az adattárolás robosztussá tételéhez replikálhatjuk a kulcsokat is
  - A tárolt adatok hasonló robosztusságra tesznek szert

Számítási komplexitás

- A Chord, és még sok más megoldás O(logN) tár- és úthossz komplexitással rendelkezik
- O(1) tárkomplexitás is elérhető O(logN) úthossz komplexitással (pl. Viceroy) vagy $O(\log^2 N)$ (Symphony) úthossz komplexitással
- A Kelips algoritmus $O(N^{1/2})$ tárkapacitású és O(1) útvonalhosszt ér el, de ehhez kell kulcsreplikáció.

Főbb pontok

- A fedőhálózatok a nagyméretű, megbízhatatlan, elosztott rendszerek egyik fő eszköze
- Az elosztott hashtáblák fő összetevői
  - Konzisztens hash-elés
  - Egy fedőhálózat, ami implementálja az útvonalválasztó algoritmust egy adott kulcs megtalálásához
- A Chord egy gyűrű hálózatot használ, „shortcut” élekkel, és O(log N) tár- és ótvonalhossz komplexitást ér el
- Az elméleti határ O(log N) útvonalhosszhoz O(1) tárkomplexitás

References

- Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM), pages 149–160, San Diego, CA, 2001. ACM, ACM Press.
- David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the World Wide Web. In Proceedings of the twenty-ninth annual ACM symposium on Theory of computing (STOC '97), pages 654–663, New York, NY, USA, 1997. ACM.
- Gurmeet Singh Manku, Mayank Bawa, and Prabhakar Raghavan. Symphony: Distributed hashing in a small world. In Proceedings of the 4th USENIX Symposium on Internet Technologies and Systems (USITS'03), 2003.
- Dahlia Malkhi, Moni Naor, and David Ratajczak. Viceroy: A scalable and dynamic emulation of the butterfly. In Proceedings of the 21st ACM Symposium on Principles of Distributed Computing (PODC'02), pages 183–192, New York, NY, USA, 2002. ACM.
- Indranil Gupta, Ken Birman, Prakash Linga, Al Demers, and Robbert Renesse. Kelips: Building an efficient and stable P2P DHT through increased memory and background overhead. In Peer-to-Peer Systems II (IPTPS'03), volume 2735 of Lecture Notes in Computer Science, pages 160–169. Springer Berlin / Heidelberg, 2003.
- Eng Keong Lua, Jon Crowcroft, Marcelo Pias, Ravi Sharma, and Steven Lim. A survey and comparison of peer-to-peer overlay network schemes. IEEE Communications Surveys and Tutorials, 7(2):72–93, 2005.