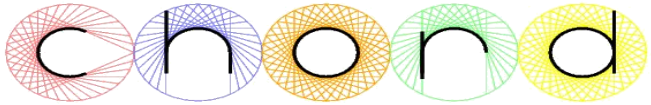
Hálózatba kapcsolt erőforrás platformok és alkalmazásaik

Simon Csaba
TMIT
2018

Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications

Ion Stoica, Robert Morris, David Karger,
M. Frans Kaashoek, Hari Balakrishnan - MIT



Web szerver terhelésének elosztása

- Mirroring
  - A teljes tartalom duplikálva több helyen
    - Nagy forgalom, nagy adattárolási kényszer
    - Nagy terhelés minden mirror oldalon
      - **Az összes európai ugyanahhoz az európai mirror szerverhez fordul amikor az USA-ban alszanak, az amerikai mirror szerver pedig kihasználatlan**
- Caching
  - A tartalmat részekre vágjuk, és elosztottan tároljuk
    - A népszerű tartalmakat mindenhol tárolják
    - Ha nincs nálam, tudom kitől kell kérni
  - Hagyományos hash függvény
    - C = hash (o) mod N

      C – cache, o – object, N – cache szerverek száma
  - Probléma: Ha egy cache eltűnik, a teljes tartalmat újra kell osztani
    - C’ = hash (o) mod (N-1)

Consistent hashing

- D. Karger, E. Lehman, T. Leighton, M. Levine, D. Lewin, R. Panigrahy,
- „Consistent hashing and random trees: distributed caching protocols for
- relieving hot spots on the World Wide Web”, Proceedings of ACM Symposium
- on Theory of Computing, El Paso, Texas, 1997.
- http://theory.lcs.mit.edu/~karger/Papers/web.ps.gz

- Tim-Berners Lee és T. Leighton eredeti ötlete
-

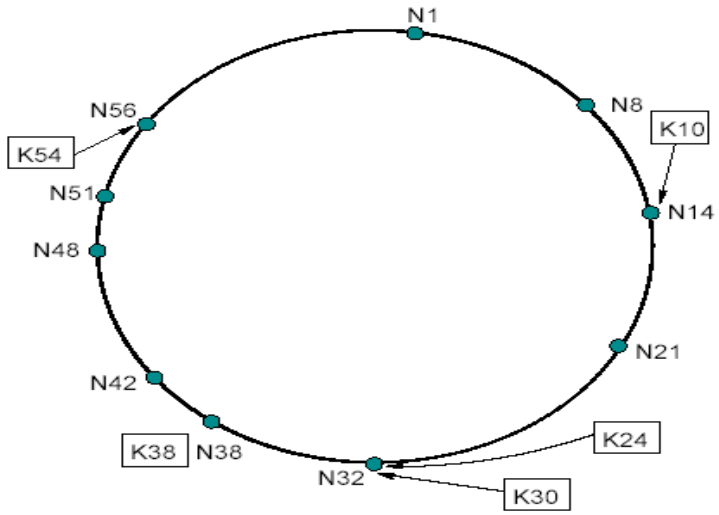
Consistent Hashing

- *m* bites azonosító tér a fájloknak és a csomópontoknak
  - *m* tetszőleges szám, elég nagy, hogy az ütközés valószínűsége kicsi legyen
- Fájl azonosító = H(Kulcs)
  - Kulcs = “Let_It_Be”, H(Kulcs) = 60
- Csomópont azonosító = H(IP cím)
  - IP = “198.10.10.1”, H(IP cím) = 123
- Egyenletes eloszlással

- Hogyan lehet a fájl azonosítókat a csomópont azonosítókhoz rendelni?

Consistent Hashing

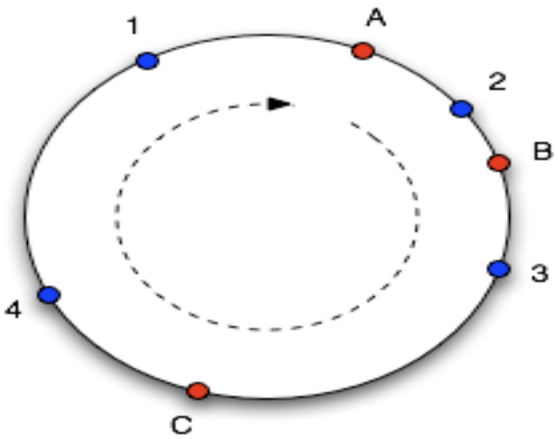
- Azonosítók egy *azonosító gyűrű* mentén
- elhelyezve modulo 2^m
  - Példa: m = 6

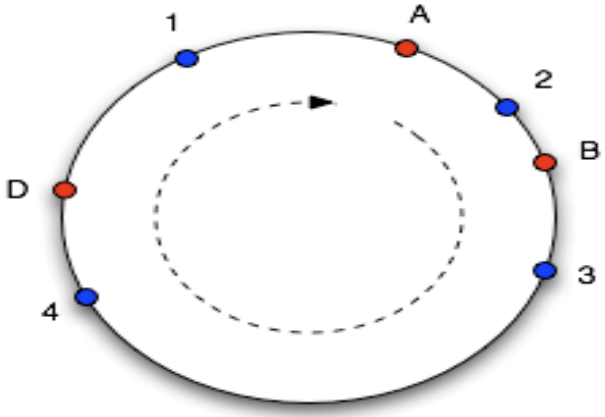


- Minden K kulcs az őt követő legközelebbi
- N csomópontnál kerül tárolásra
  - N = successor(k)

Consistent Hashing

- A csomópont felelős az 1. és 4. kulcsokért
- Ha C kilép, A lesz felelős a 3. kulcsért is
- Ha D belép, átveszi a felelősséget a 3. és 4. kulcsért
  - A többi megfeleltetés változatlan

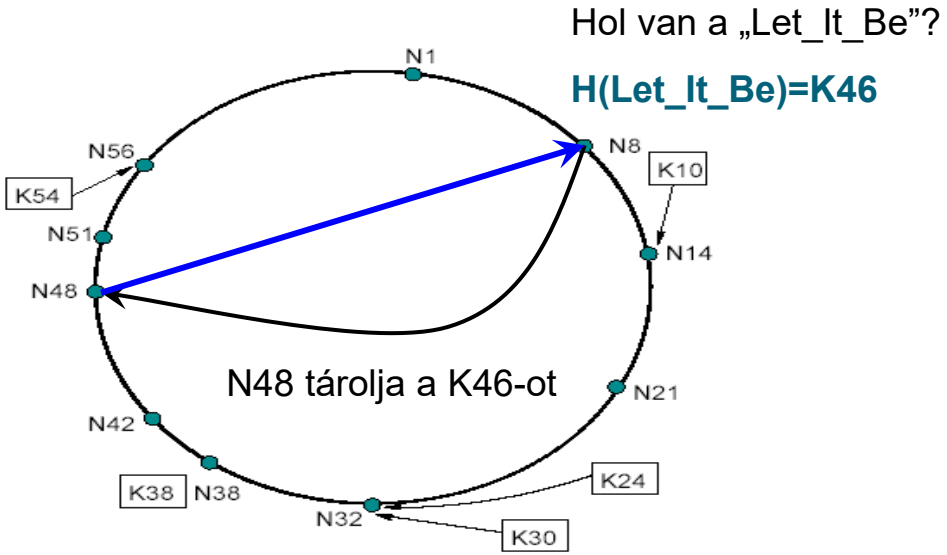




Consistent hashing - keresés

Minden csomópont ismeri *az összes többi* csomópontot

- **Nagy „útválasztó" táblák - O(N)**
- ***Gyors* keresés O(1)**

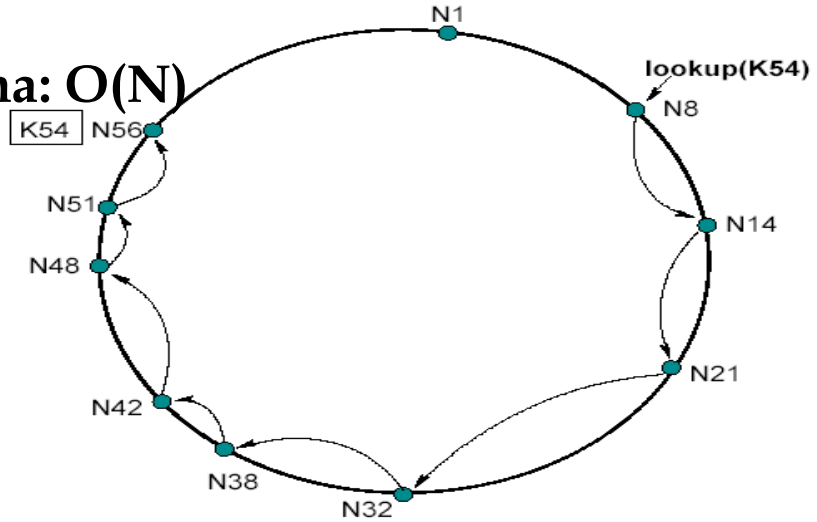


Chord: alap keresés

Minden csomópont ismeri az őt követőt a gyűrűn

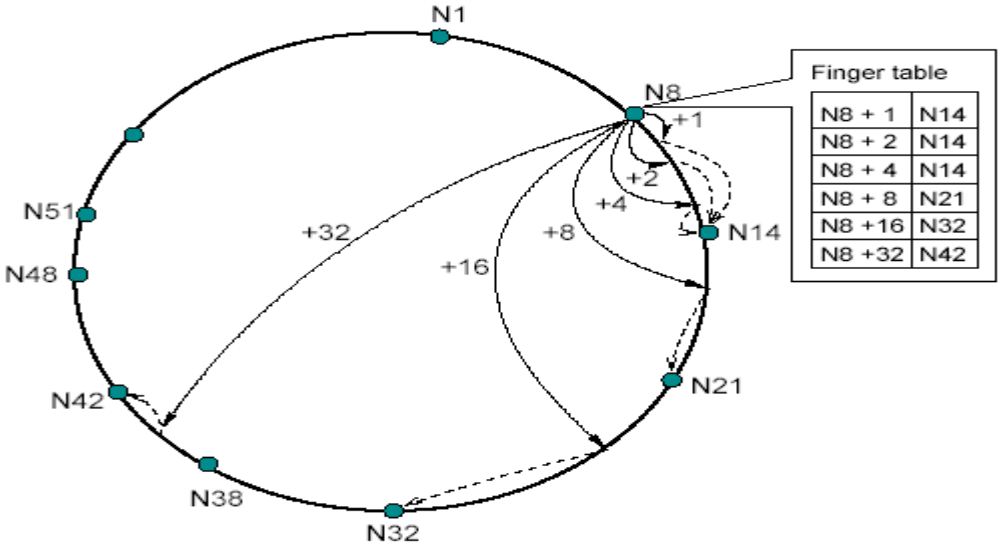
- **Az öt megelőzőt is hasznos ismernie**

Keresési idő ~ üzenetek száma: O(N)



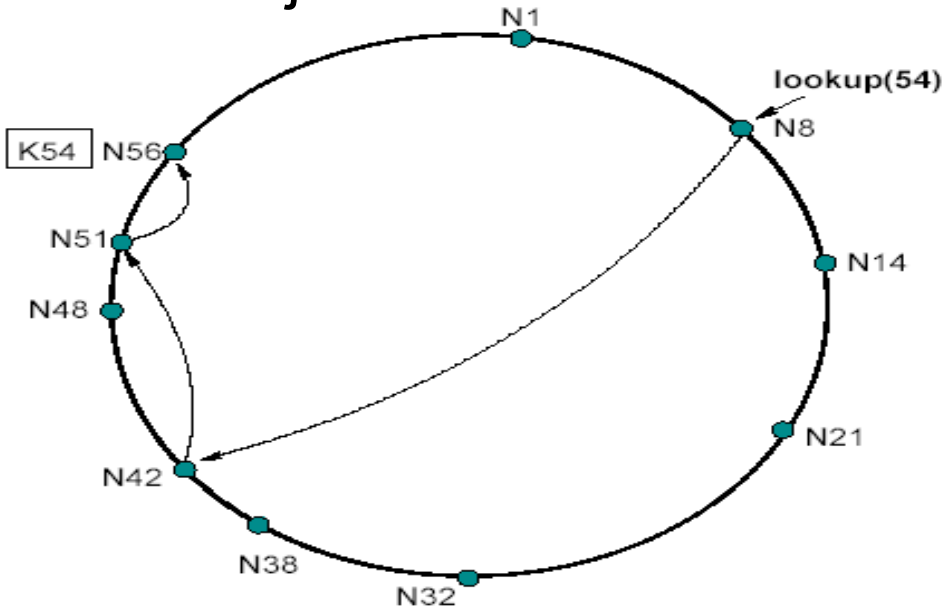
„Mutató táblák” (Finger tables)

- Minden egyes csomópont *m* számú további csomópontot tart nyilván
- Az előre mutató távolság exponenciálisan növekszik
  - *finger[i] = successor (n + 2^{i-1})*



Chord: gyors/skálázódó keresés

- A mutató táblák segítségével a keresésnek O(log N) csomópontot kell bejárnia

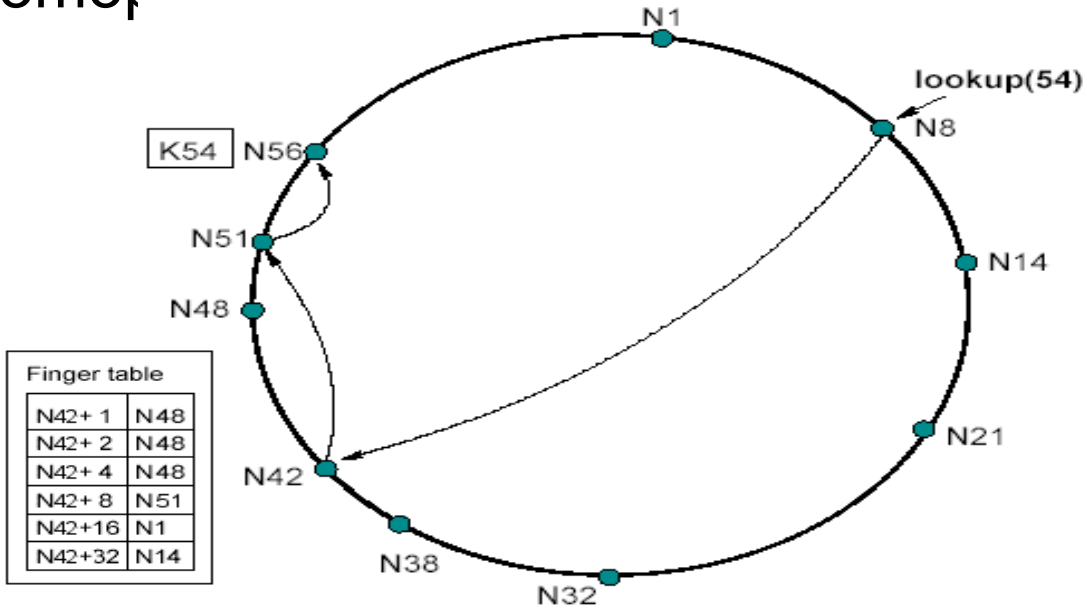


Finger table

N8 + 1	N14
N8 + 2	N14
N8 + 4	N14
N8 + 8	N21
N8 +16	N32
N8 +32	N42

Chord: gyors/skálázódó keresés

- A mutató táblák segítségével a keresésnek O(log N) csomóponton kell beiárnia



Chord: gyors/skálázódó keresés

- Minden csomópont *m* további bejegyzést tartalmaz

- Minél közelebbi a kulcs, annál részletesebb
- információval rendelkezik róla a csomópont

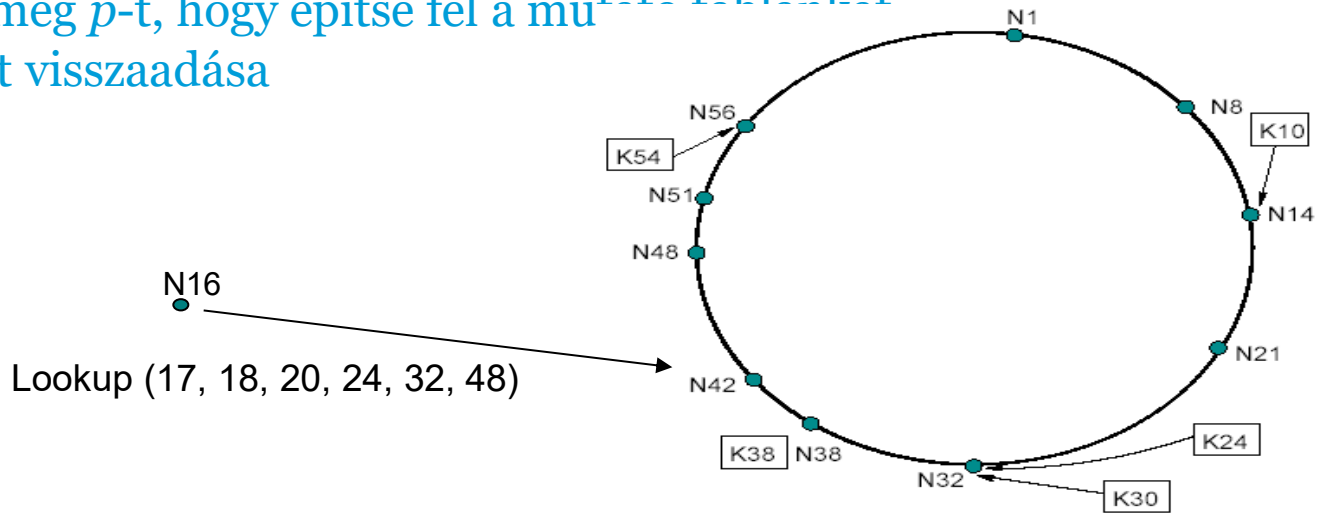
- Általában nem biztosítja az azonnali célba jutást

Új érkező

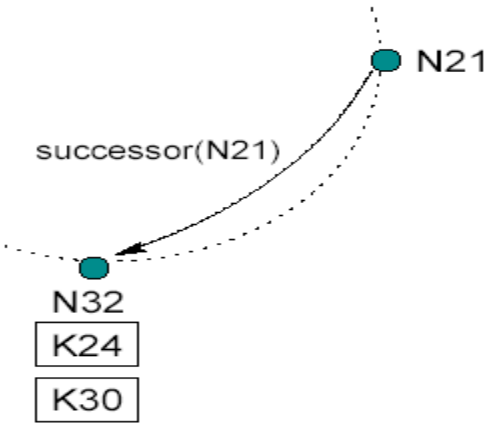
- Három lépésben (alap működés)
  - Újonnan érkező *mutató táblájának* feltöltése
  - Gyűrű csomópontok *mutató táblájának* frissítése
  - Kulcsok cseréje
- „Lusta” vagy kevésbé agresszív működés
  - Csak a *követő* csomópont beállítása
  - Periodikus *követő (successor), megelőző (predecessor)* ellenőrzés
  - Periodikus *mutató tábla* frissítés

Új érkező (mutató táblák)

- Kiindulás: bármely *p* ismert csomópontból
  - Kérjük meg *p*-t, hogy építse fel a mutató táblázatot
  - Táblázat visszaadása



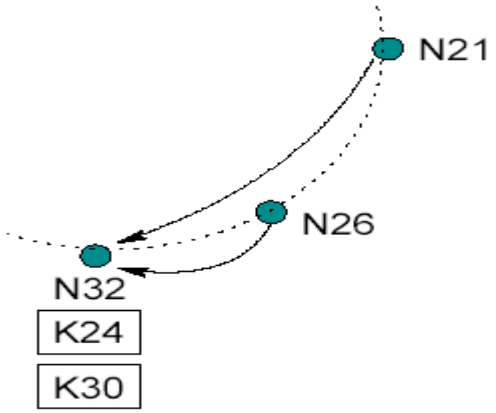
Új érkező



A gyűrű csomópontok mutató tábláinak frissítése

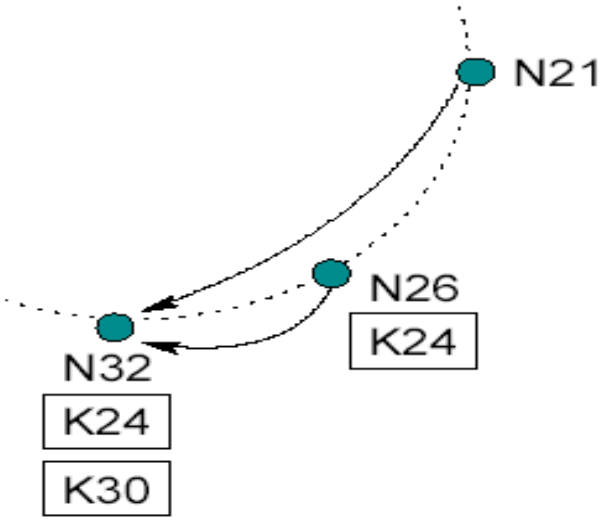
- **új érkező a *frissítés* funkciót kelti életre a szomszédos csomópontokban**
- **csomópontok rekurzívan frissíttetik a további csomópontok mutató tábláit**

Új érkező



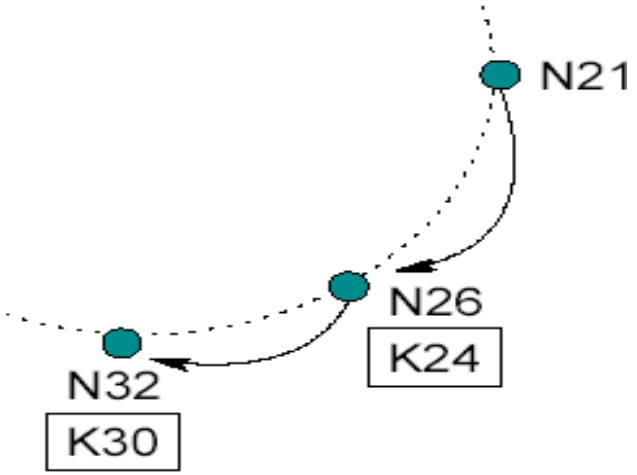
- N26 belép a rendszerbe
- N26.successor = N32
- N26 értesíti N32-t
- N32.predecessor = N26

Új érkező



- N26 átmásolja a ráeső kulcsokat
- N21.frissítés:
  - lekéri N32-től a predecessor-t, aki N26

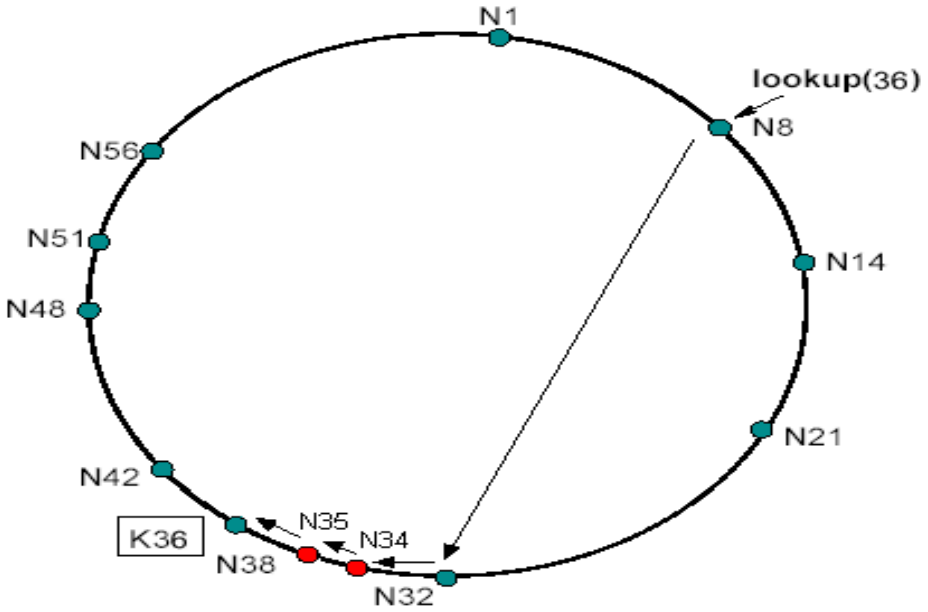
Új érkező



- N21.successor = N26
- N21 értesíti N26-ot a létezéséről
- N26.predecessor = N21

Új érkező: keresés

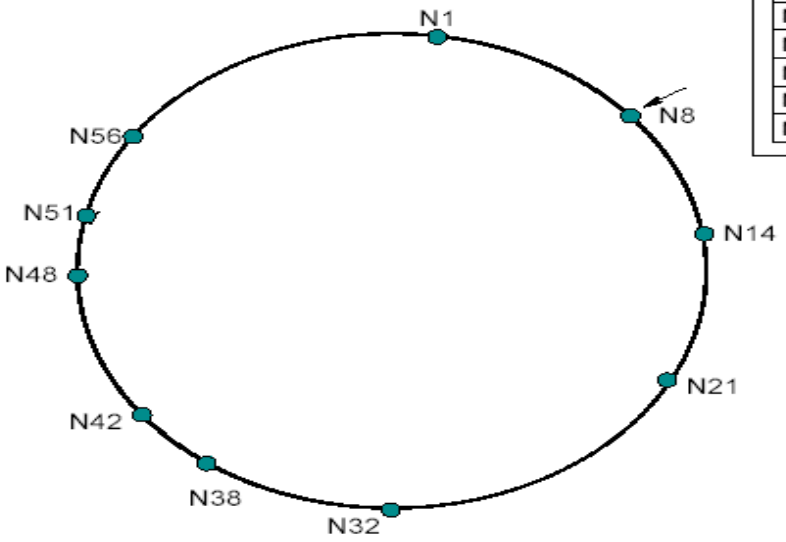
- Korrekt *mutató táblák* esetén O(log N)
- Ha csak a *követő* lánc helyes, akkor is korrekt, de lassabb működés



Finger table

N8 + 1	N14
N8 + 2	N14
N8 + 4	N14
N8 + 8	N21
N8 +16	N32
N8 +32	N42

Hibák kezelése

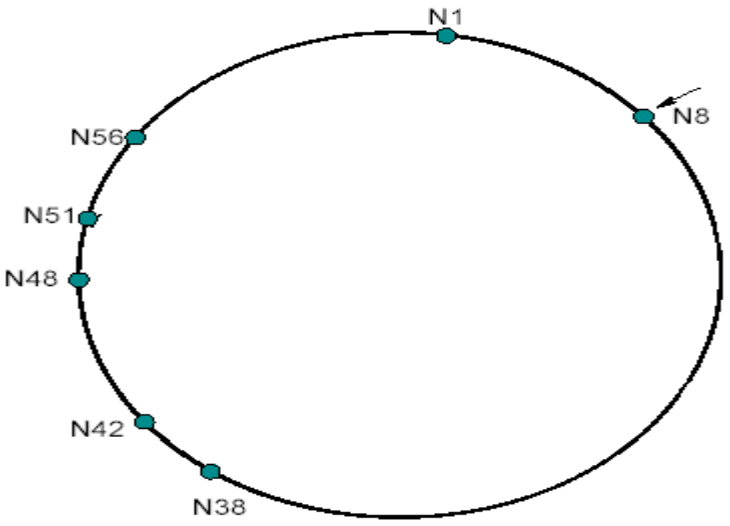


Finger table

N8 + 1	N14
N8 + 2	N14
N8 + 4	N14
N8 + 8	N21
N8 +16	N32
N8 +32	N42

- Csomópontok kiesése (hiba) helytelen keresést eredményezhet
- Mi van ha az N14, N21 és N32 egyszerre meghibásodik, vagy kilép?
- Hogyan tud az N8 tudomást szerezni az N38-ról?

Hibák kezelése



Finger table

N8 + 1	N14
+ 2	N14
+ 4	N14
+ 8	N21
+16	N32
+32	N42

- Csomópontok kiesése (hiba) helytelen keresést eredményezhet
- Mi van ha az N14, N21 és N32 egyszerre meghibásodik, vagy kilép?
- Hogyan tud az N8 tudomást szerezni az N38-ról?

Hibák kezelése (II)

- Követő *lista*
  - Az egyetlen *követő* helyett *r* soron követő csomópont regisztrációja
  - Hiba esetén ismeri a soron következő (élő) csomópontot
    - helyes keresés
- Valószínűségi garancia
  - *r* megválasztása, hogy a keresési hiba valószínűsége megfelelően alacsony legyen
    - r ~ O(log N)

Chord előnyök

- Hatékony: O(log N) üzenet keresésenként
  - ahol N a kiszolgálók (csomópontok) száma
- Alacsony szórás a keresési időben
  - A CAN-ben nagy a szórás
- Skálázódik: O(log N) állapot csomópontonként
- Robosztus: megbirkózik jelentős résztvevő változással

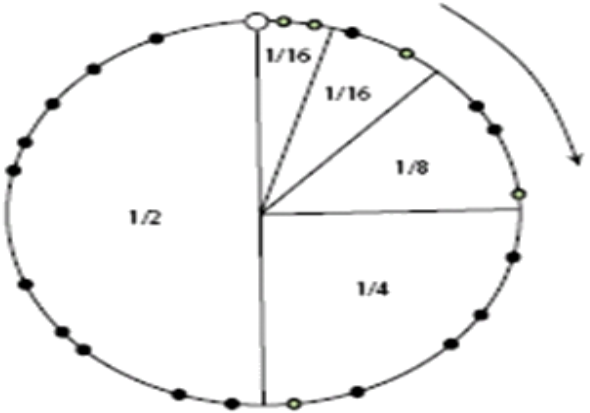
- Állítások bizonyításai [tech_report]
  - http://www.pdos.lcs.mit.edu/chord/papers/chord-tn.ps
- Feltételezés: nincs rosszakaratú résztvevő

Chord előnyök

- Minden csomópont *K/N kulcsot* kezel (N - „node”, K - „key”)
- Be/kilépés egy N-tagú hálózatba: csak *O(K/N)* kulcsot kell átmozgatni
  - Csak a dinamikus csomópont számára
- Egy keresés O(log N) üzenetet generál
- Be/kilépés után a mutató táblák frissítéséhez $O(\log^2 N)$ üzenet szükséges

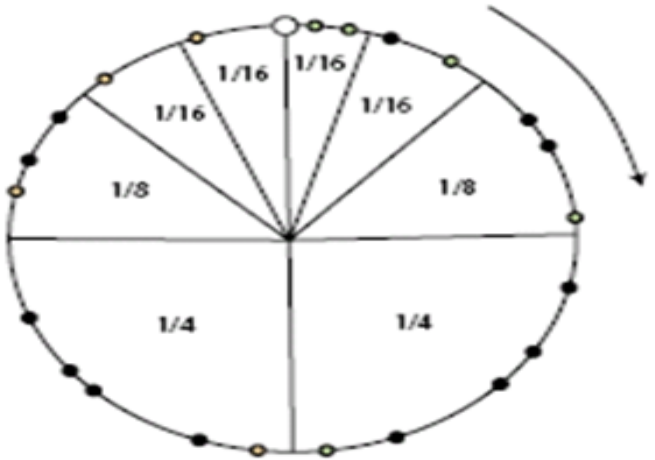
Hátrányok

- Merev finger (routing) tábla!
  - Nehezíti a tábla helyreállítását node-ok kiesése után
  - Lehetetlenné teszi a közelségi információ felhasználását
- A bejövő és kimenő irányú üzenetek eloszlása éppen
- ellentétes
  - Nem lehet a bejövő forgalmat a routing tábla frissítésére használni



Bidirectionnal Chord

- Kétirányú routing tábla
  - A tábla kétszer akkorára nő
  - Kétszer annyi vezérlő üzenetre van szükség



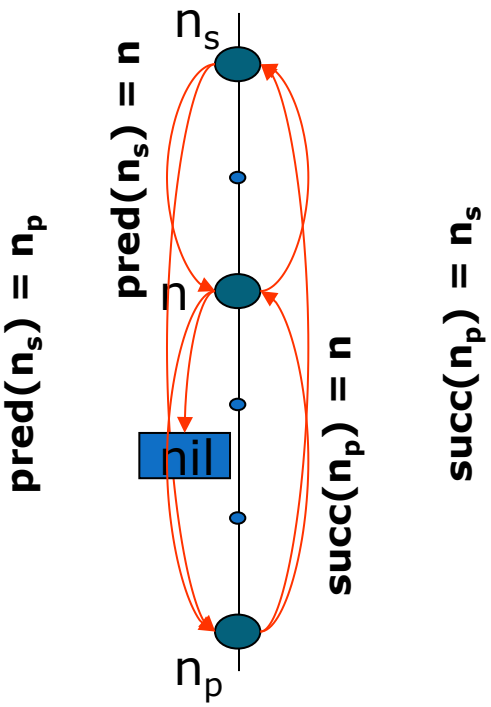
Chord implementáció

- 3000 soros C++ kód
- Library amely tetszőleges alkalmazáshoz linkelhető

- UsenetDHT
  - A teljes replikálás helyett elosztott tárolás

- OverCite
  - A CiteSeer elosztott változata

stabilize()



- **New *n* joins the DHT**
  - predecessor = nil
  - n acquires n_s as successor via some n'
  - n notifies n_s being the new predecessor
  - n_s acquires n as its predecessor
- **n_p runs stabilize**
  - n_p asks n_s for its predecessor (now n)
  - n_p acquires n as its successor
  - n_p notifies n
  - n will acquire n_p as its predecessor
- **all predecessor and successor pointers are now correct**
- **fingers still need to be fixed, but old fingers will still work**

fix_finger()

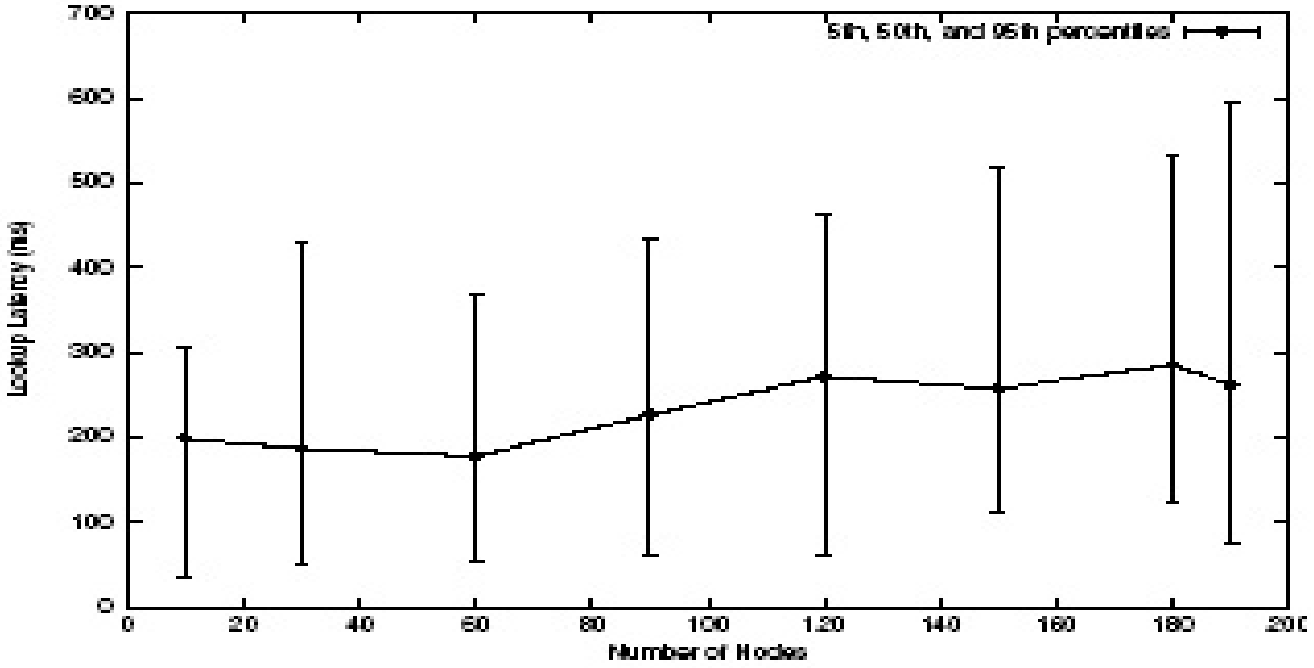
- Why do weo have to fix fingers?
  - If we are operating the Chord by the numbers?

fix_finger()

- Why do we have to fix fingers?
  - If we are operating the Chord by the numbers?
- Who notifies the finger table-owners about a leaving member?
  - Predecessors + stabilize() are not handling this issue
  - „take the french leave”
  - Non-transitive connectivity.
  - Bidirectional Chord is not enough (stochastic)
- Refresh the entries of a finger table
  - Otherwise the time of a search increases (greedy solutions for failover)

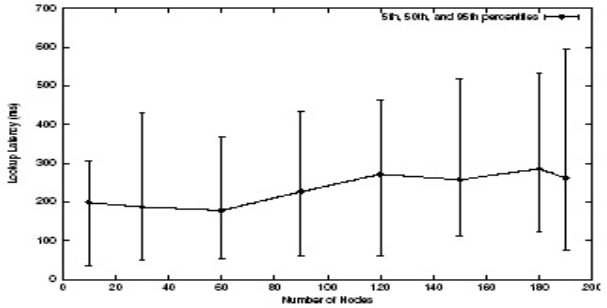
Mérési eredmények

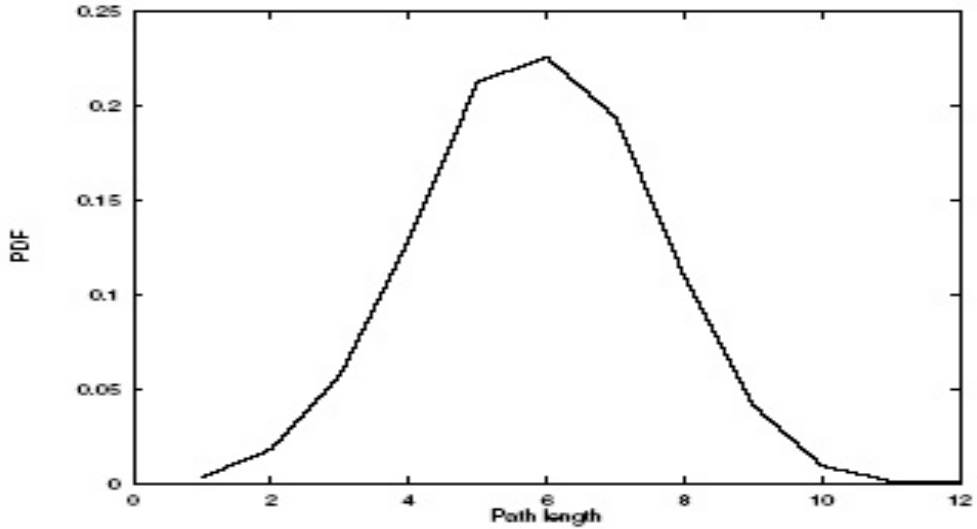
- A késleltetés csak lassan növekszik, ha növeljük a csomópontokat



Mérési eredmények

- A késleltetés csak lassan növekszik, ha növeljük a csomópontokat
- Átlagos úthossz: ½ $\log_2 N$
- A chord egy robosztus protokoll





Irodalom

- I. Stoica, R. Morris, D. Karger, F. Kaashoek, H. Balakrishnan, "Chord: A
- Scalable Peer-To-Peer Lookup Service for Internet Applications," ACM
- Sigcomm 2001.
- http://www.acm.org/sigcomm/sigcomm2001/p12.html

- The Chord Project
- http://pdos.csail.mit.edu/chord/

Feladattervezés

Mit tudtok bizonyítani a Chord szimulátorral?

Addressed Difficult Problems (1)

- **Load balance**: distributed hash function, spreading keys evenly over nodes
- **Decentralization**: chord is fully distributed, no node more important than other, improves robustness
- **Scalability**: logarithmic growth of lookup costs with number of nodes in network, even very large systems are feasible

Addressed Difficult Problems (2)

- **Availability**: chord automatically adjusts its internal tables to ensure that the node responsible for a key can always be found

- **Flexible naming**: no constraints on the structure of the keys – key-space is flat, flexibility in how to map names to Chord keys

- http://www.tmit.bme.hu/hakap

Feladat: skálázható-e a Chord?

- 20 tartalom, 10 node
- 10 random keresés indítása
- Keresés során: ugrások száma

- Screenshot: node-ok
- Excelben: hisztogram(ugrások száma)