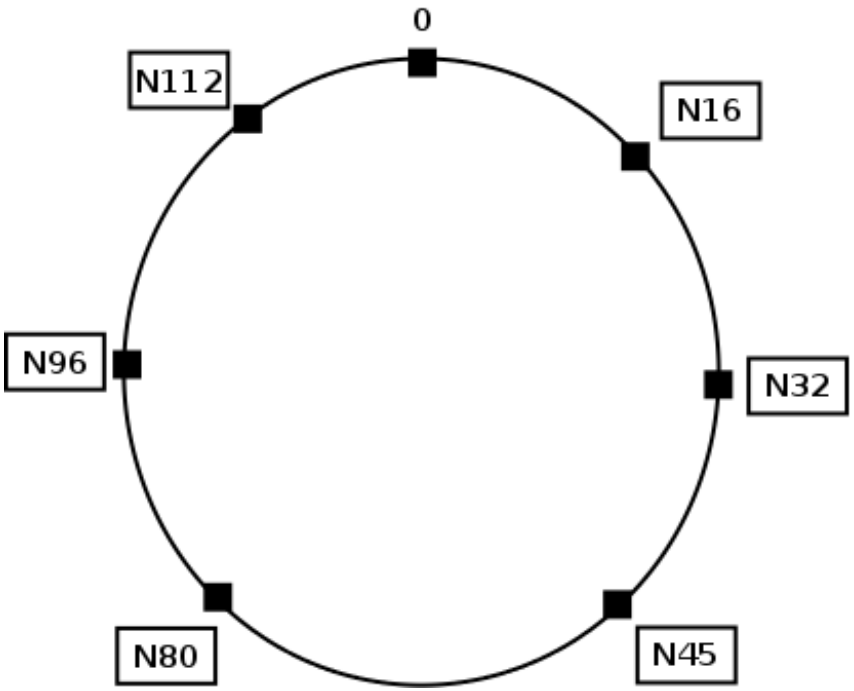
Cloud Computing

Cap 5. Chord. Pastry. Kelips.

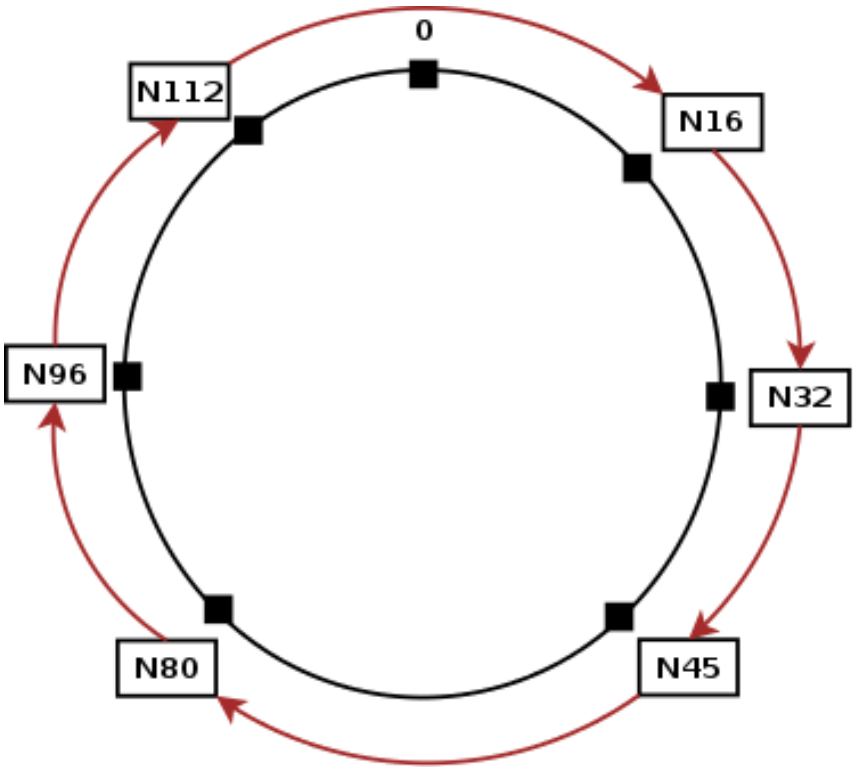
November 7, 2022

Inel de peer-uri



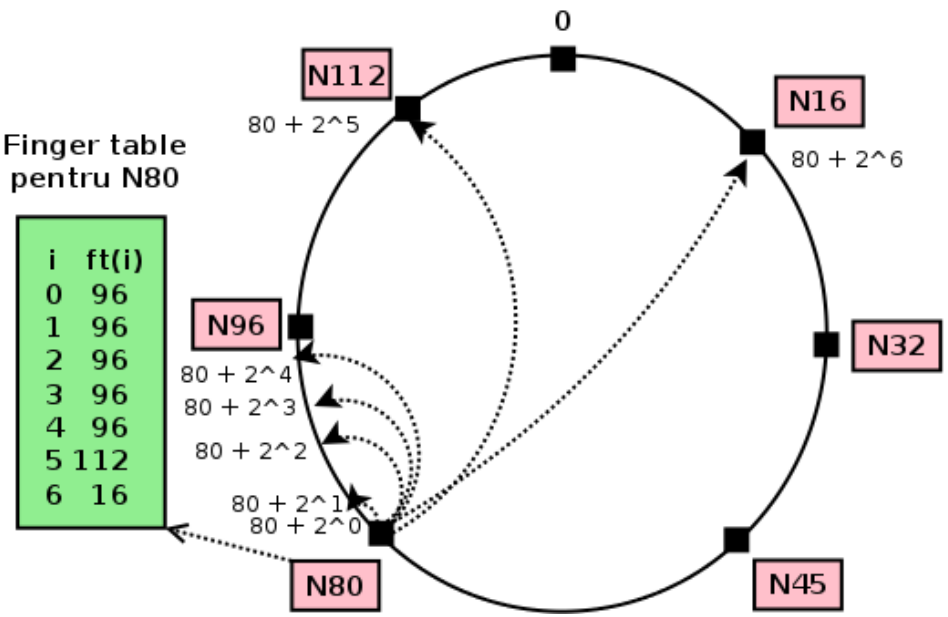
- m = 7 (127 poziții posibile), 6 noduri
- se stochează peer ID-urile
- fiecare peer își cunoaște peer-ul succesor:

Inel de peer-uri



- m = 7 (127 poziții posibile), 6 noduri
- se stochează peer ID-urile
- fiecare peer își cunoaște peer-ul succesor
- în mod similar pentru predecesori (mai rar folosit în practică)
- nodul **succesor** este primul tip de peer pointer

Finger tables



- m = 7
- pentru nodul cu ID-ul n, $ft(i)_n$ = primul peer cu ID-ul $\geq (n + 2^i)$ **modulo** 2^m
- folosit pentru routing-ul unui query foarte rapid

  n = 80 (N80)

  $n + 2^0 \rightarrow N96$

  . . .

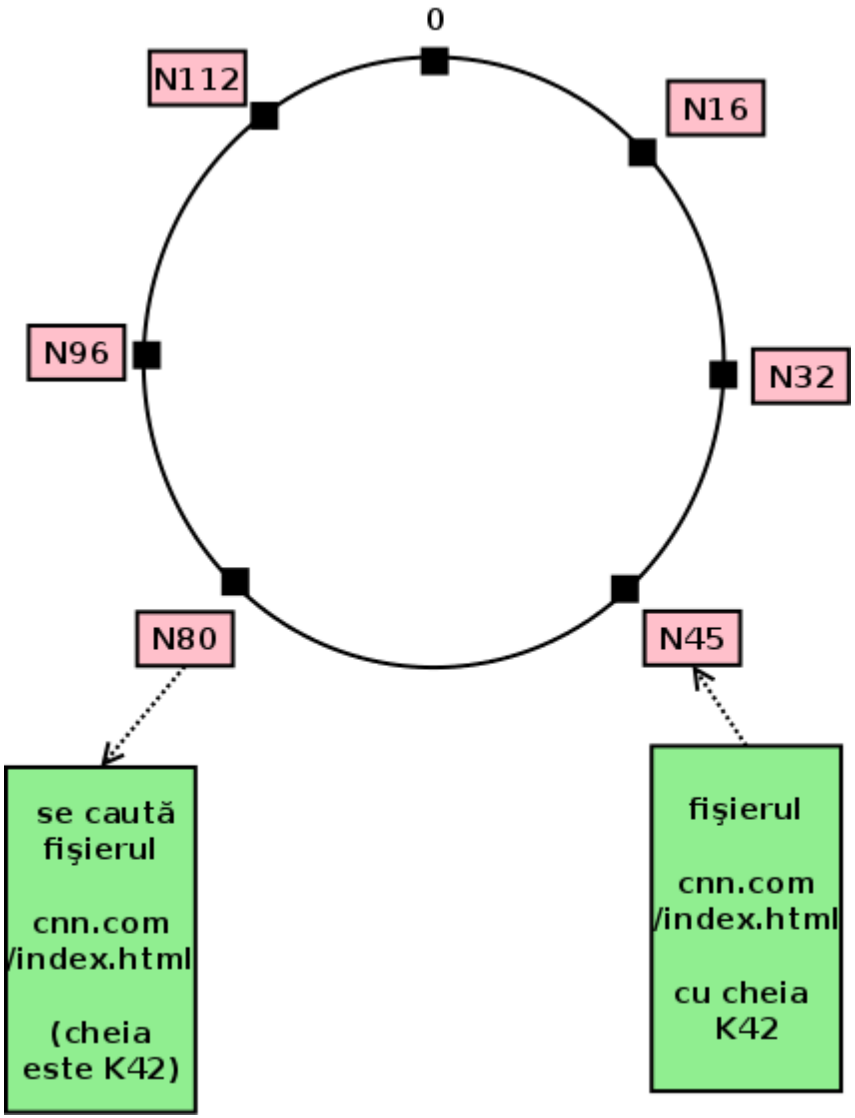
  $n + 2^5 = 80 + 32 \rightarrow N112$

  $n + 2^6 = 80 + 64 \rightarrow N16$
- i crește cu 1, dar distanța se dublează

Distribuirea fișierelor cu Chord

- fișierele sunt mapped folosind aceeași funcție de hashing
  - SHA-1(nume fișier) $\rightarrow$ string de 160 biți
  - fișierul e **stocat în primul peer cu ID-ul mai mare sau egal cu cheia sa (modulo 2^m)**
- fișierul cnn.com/index.html mapează în cheia K42, stocat în primul peer cu ID-ul ≥ 42
  - aplicația se numește **cooperative web caching**
  - se poate generaliza pentru orice sistem de file sharing (e.g. mp3)
- consistent hashing - cu K chei și N peer-uri, fiecare peer stochează O(K/N) chei (load balancing bun)

Algoritmul Search

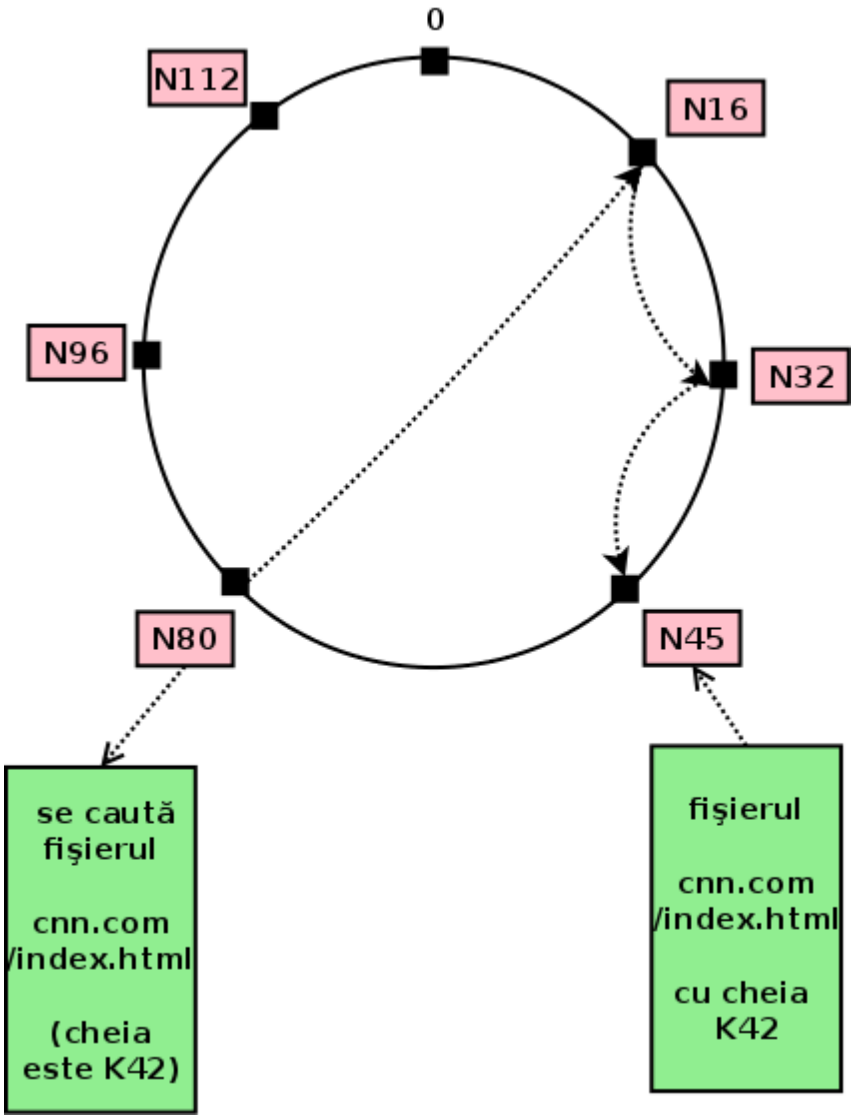


- la nodul n, trimite un search query pentru cheia K,
- către cea mai mare intrare în finger table $\leq K$
- dacă nu există nici una, trimite query-ul către succesor

  se consideră sensul orar

  N80 trimite la N16

Algoritmul Search



- la nodul n, trimite un search query pentru cheia K,
- către cea mai mare intrare în finger table $\leq K$
- dacă nu există nici una, trimite query-ul către succesor

* se consideră sensul orar

* N80 trimite la N16

* N16 poate trimite la N32 sau N80 (cum?)

* N32 trimite către succesor (nu are vecin ≤ 42)

* se folosește plain RPC

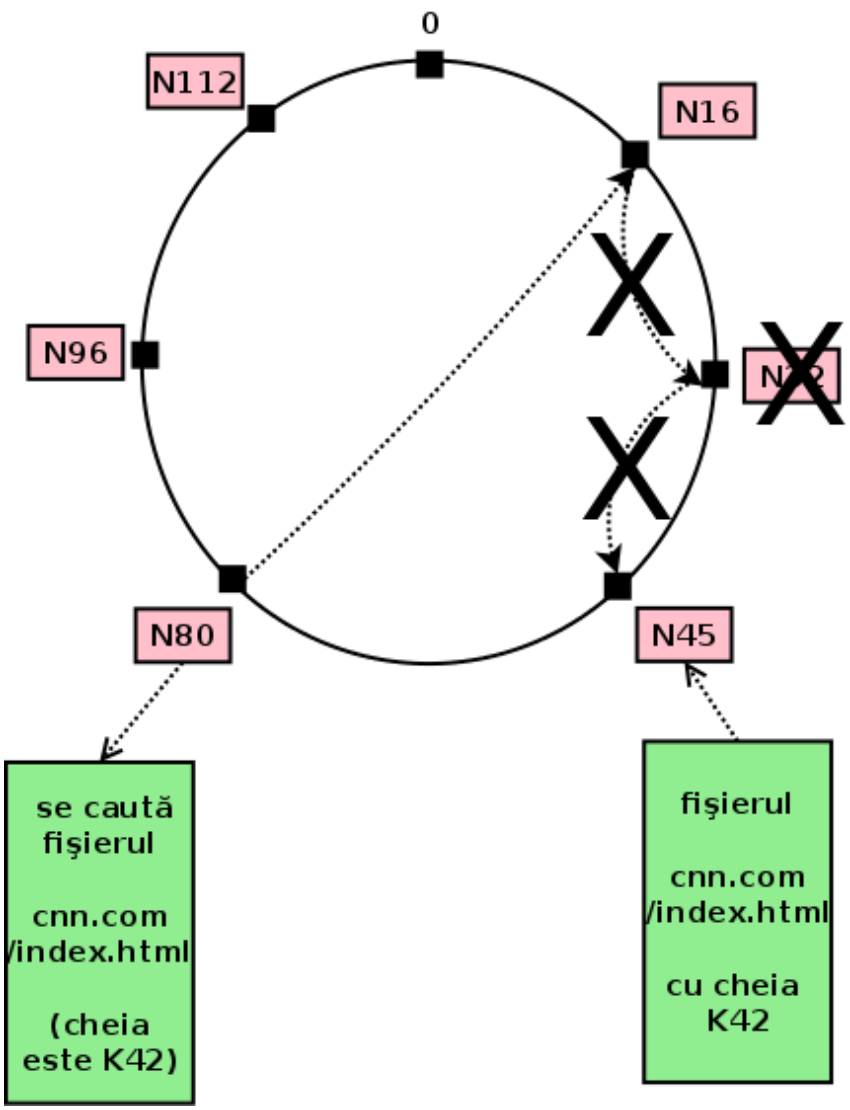
Analiza de timp

- la fiecare pas, distanța dintre nodul curent și target se înjumătățește
  - reducere la absurd: query-ul nu sare în a doua jumătate, ci doar în prima
  - dar intrările în finger table sunt noduri obținute prin dublare
  - vom avea cel puțin una (următoarea) în finger table, contradicție
  - deci nodul va avea cel puțin un vecin (FT-entry) în a doua jumătate
- după log(N) forwardings, distanța către cheie este cel mult $2^m/2^{log(N)} = 2^m/N$
- dar SHA-1 face, cu probabilitate ridicată, ca pe un segment de cerc de $2^m/N$ să fie cel mult un succesor
- cu probabilitate redusă, succesorii vor fi cel mult în O(log(N))

Analiza de timp

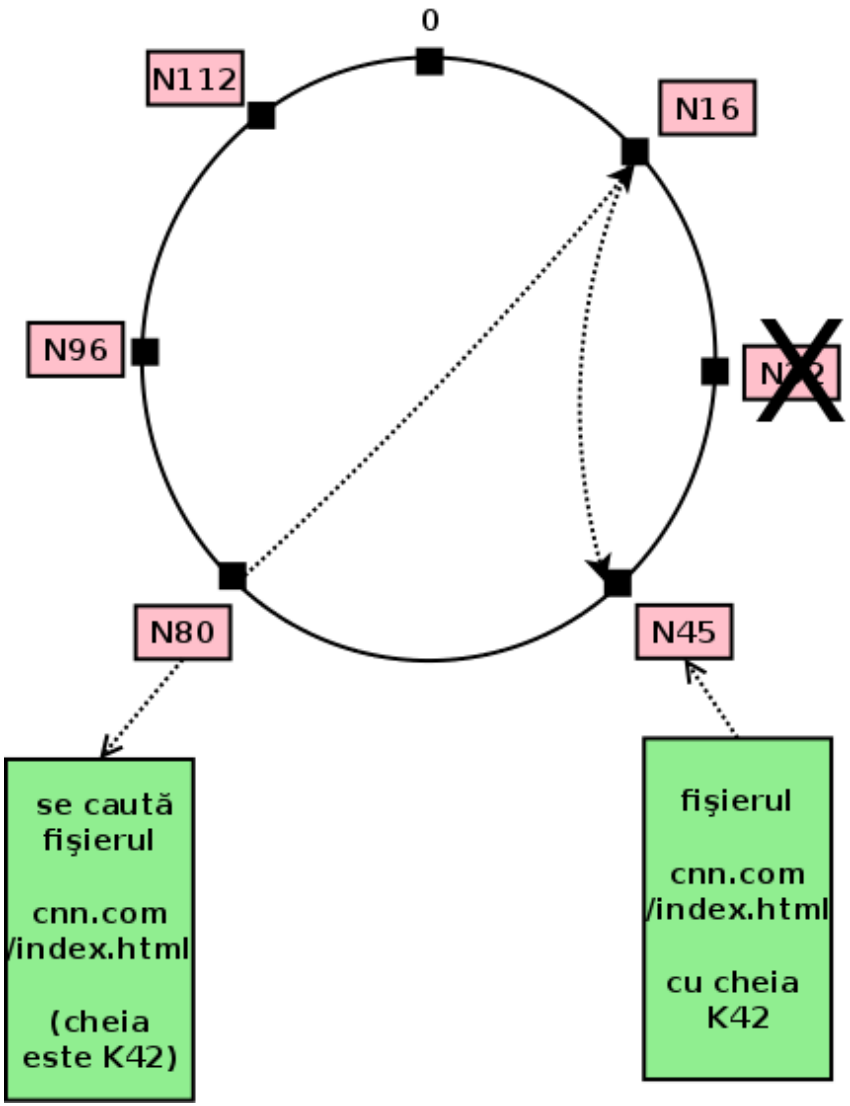
- timpul de O(log(N)) se păstrează și pentru inserții de fișiere sau orice routing
  - routing-ul e folosit ca **building block** pentru operațiile de insert, lookup, delete
- timpul O(log(N)) este valabil doar dacă finger table și succesorii sunt activi
- ce se întâmplă în cazul căderii acestora?

Search și failures



- nodul intermediar N32 cade
- nodul N16 nu știe de existența lui N45
- query-ul se pierde

Search și failures



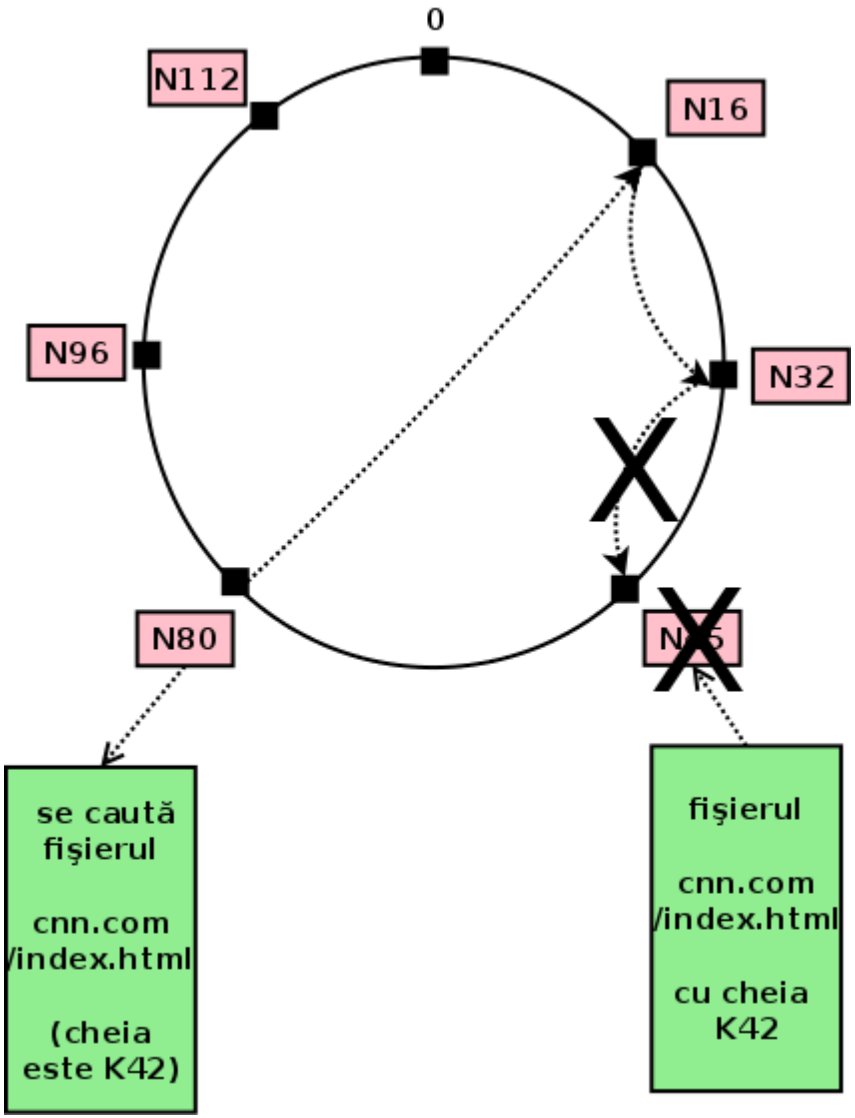
- rezolvare: nodul va păstra nu doar un succesor, ci

  r **intrări succesor multiple**

Analiza conectivității

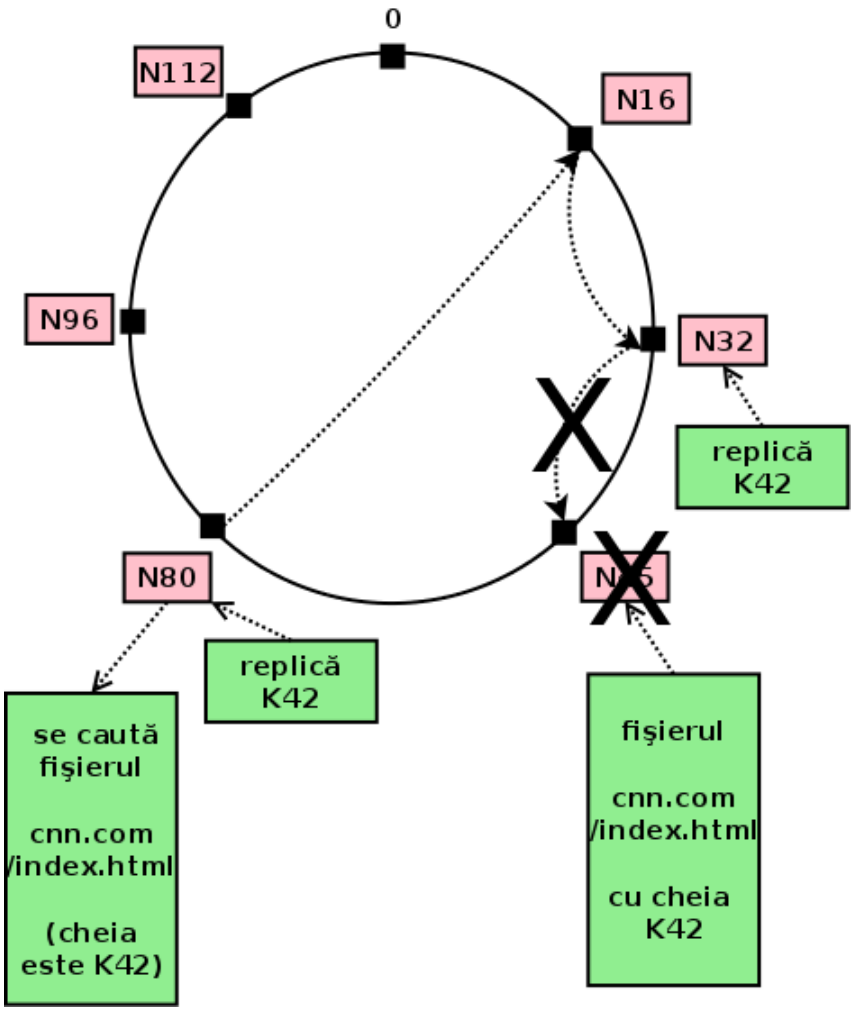
- alegerea $r = 2 \cdot log(N)$ este suficientă pentru menținerea corectitudinii de lookup cu probabilitate ridicată
- presupunem că 50% din noduri cad (se inactivează)
- Pr(în nodul n, ca cel puțin un succesor să fie activ) = $1 - \left(\frac{1}{2}\right)^{2log(N)} = 1 - \frac{1}{N^2}$
- Pr(să se întâmple asta în toate nodurile active) = $\left(1 - \frac{1}{N^2}\right)^{\frac{N}{2}} = e^{-\frac{1}{2N}} \approx 1$, pentru valori mari ale lui N
- probabilitatea ca inelul să fie conectat e aproape evenimentul sigur

Failures și replicare



- dacă N45 cade, nu există o copie a fișierului

Failures și replicare

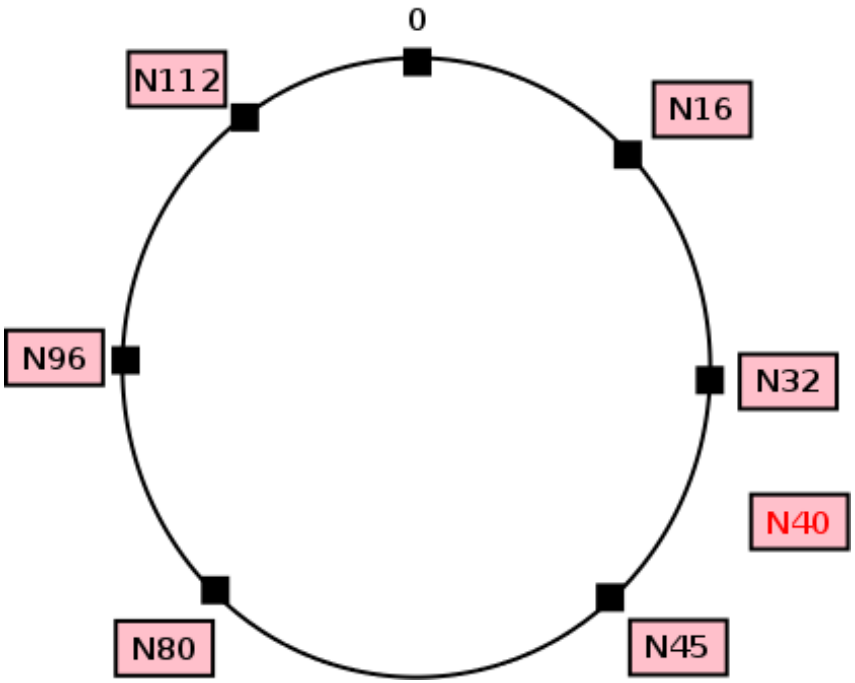


- dacă N45 cade, nu există o copie a fișierului
- se replică fișierul, și în succesor și în predecesor
- o cheie foarte căutată va avea factor de replicare mai mare

Schimbări dinamice

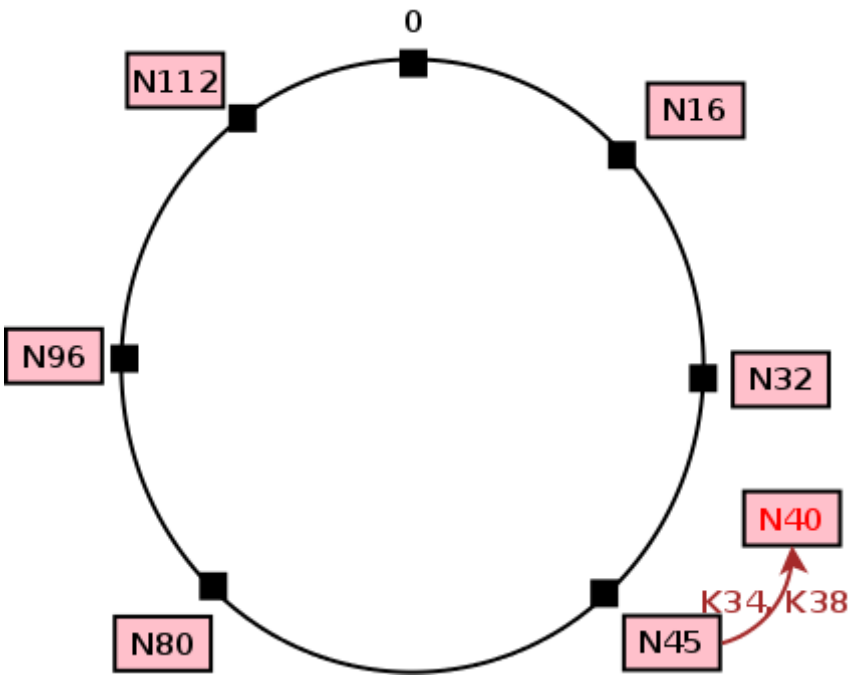
- peer-urile pot cădea (deveni inactive)
- peer-uri noi intră în sistem
- peer-urile părăsesc sistemul
- sistemele P2P au o rată de **churn** ridicată
  - 25% pe oră în Overnet
  - 100% pe oră în Gnutella
  - mai mică în managed clusters
  - trăsătură comună în orice sistem distribuit, clusters, clouds (e.g. AWS)
- întotdeauna este nevoie de update-ul succesorilor și intrărilor în finger tables, precum și copierea cheilor

Un nou peer face join



- tracker-ul redirectează noul nod N40 către N45 (și N32)
- N32 își updatează succesorul la N40
- N40 inițializează succesorul la N45, și copiază finger table (ceea ce este parțial incorect)
- N40 dialoghează periodic cu vecinii pentru a-și updata finger table
- **stabilization protocol** rulează în fundal, periodic în fiecare nod: interoghează vecinii de succesori (și predecesori)

Un nou peer face join



- N40 va avea nevoie să copieze anumite fișiere de pe N45 (file ID între 32 și 40)
- apare un transfer de fișiere între succesor și predecesor

Un nou peer face join

- un nou peer afectează în medie O(log(N)) intrări în finger table-urile din sistem
- numărul de mesaje pentru un nou peer care intră este în $O(log^2(N))$
- situație similară la leave-ul unui peer
- pentru situația de failure, este nevoie de failure detectors

Stabilization protocol

- join-urile și leave-urile concurente, precum și failures, pot introduce bucle în pointeri, precum și failed lookups
- peer-urile din Chord rulează periodic un protocol de stabilizare ce verifică și updatează pointer-ii și cheile
- asigură că nu există bucle în finger tables, succesul final al lookup-urilor și lookup-uri în O(log(N)) cu probabilitate ridicată
- fiecare rundă de stabilizare a unui peer necesită un număr constant de mesaje
- strong stability necesită un număr de runde în $O(N^2)$
- rundele de stabilizare nu sunt sincrone!
- `https://en.wikipedia.org/wiki/Chord_(peer-to-peer)`

Churn

- nodurile intră (join), ies (leave), sau cad (fail)
- rate orare de churn între 25-100% din totalul numărului de noduri din sistem
- duce la copieri excesive, ne-necesare, ale cheilor (cheile sunt replicate)
- algoritmul de stabilizare consumă lățime de bandă în acest scop
- fișierele sunt replicate, dar ar fi suficient ca doar meta-informația despre fișiere să fie replicată
- alternative:
  - introducerea unui nivel de indirectare (orice sistem P2P): fișierul rămâne în peer-ul care l-a uploadat, se reduce bandwidth-ul
  - replicarea meta-datelor cu un grad mai ridicat $\rightarrow$ Kelips

Noduri virtuale

- datorită unui număr mic de noduri, hashing-ul poate deveni neuniform, iar load balancing-ul are de suferit
- soluția este ca fiecare nod să fie considerat ca o colecție de noduri virtuale ce se comportă independent
- fiecare nod virtual intră (join) în sistem
- se reduce dezechilibrul încărcării
- se creează (virtual) o populație mare de noduri

Chord - sumar

- **virtual ring** și **consistent hashing** sunt folosite în key-value stores precum Cassandra, Riak, Voldemort, DynamoDB (Amazon)
- sisteme de fișiere precum CollectionFS, Ivy (MIT)
- DNS lookup service folosește Chord

Pastry

- Anthony Rowston (MS Research), Peter Druschel (Rice University)
- asignează ID-uri nodurilor, precum Chord, folosind o funcție de hashing, creând un inel virtual
- un nod are noțiunea de **leaf set** - fiecare nod își cunoaște succesorii și predecesorii

Vecini în Pastry

- fiecare nod păstrează tabele de routing bazate pe **prefix matching** - similar routing-ului într-un hipercub
- routing-ul este bazat pe prefix matching, în O(log(N))
- se aleg vecinii astfel încât în rețeaua de bază, numărul de hop-uri dintre ei este mic

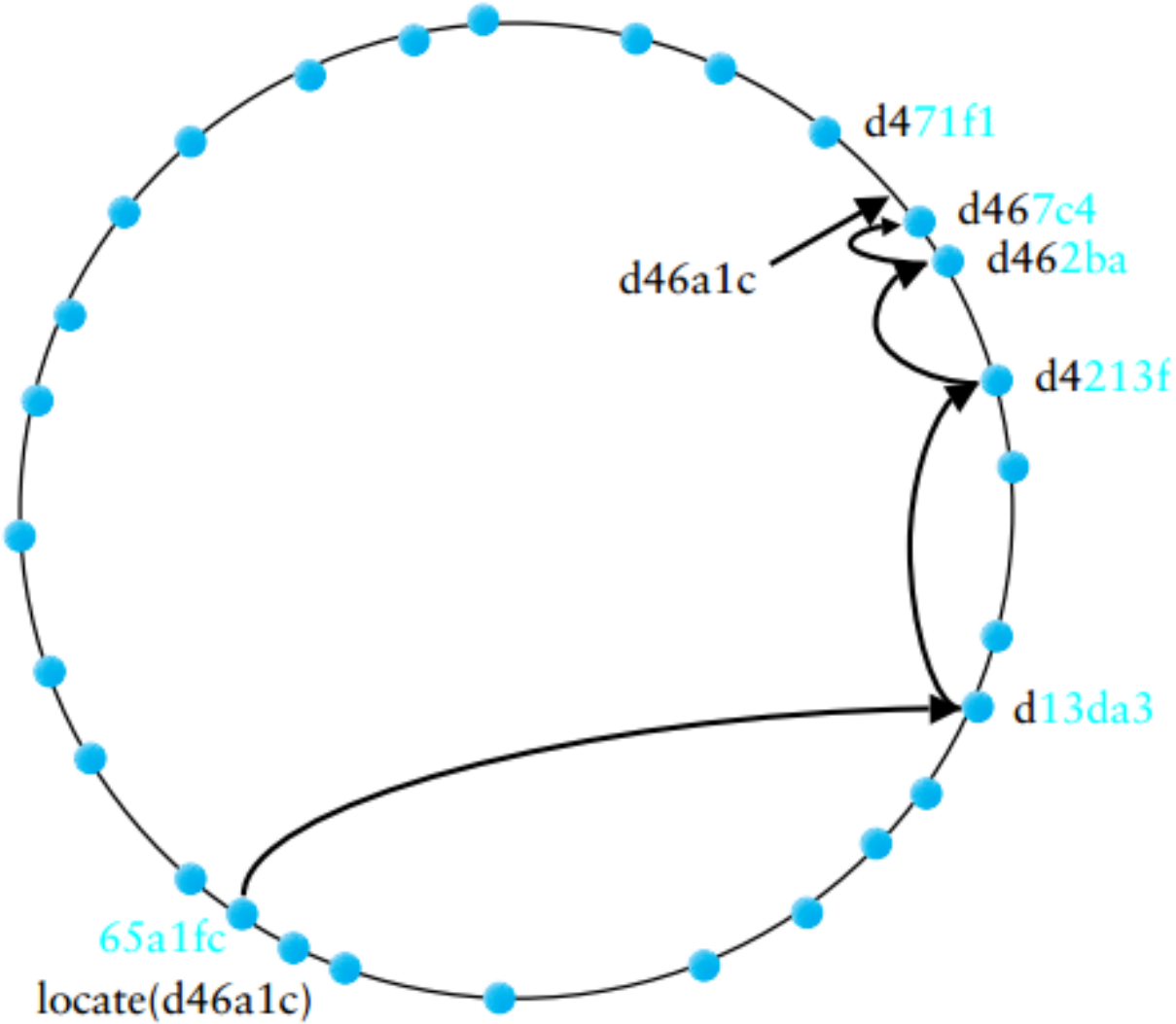
Routing în Pastry

- de ex. un peer are ID-ul 0111.0100.101
- peer-ul are drept vecini peer-uri cu ID-uri ce încep cu următoarele prefixe (* reprezintă un bit diferit de cel al ID-ului, pe acea poziție):
  - *
  - 0*
  - 01*
  - 011*
  - ...
  - 0111.0100.10*
- un peer cu N biți va avea N vecini
- când se face routing-ul către un peer, de exemplu 0111.01**11.001**, se trimite către un vecin al cărui ID începe cu cel mai lung prefix din ID-ul destinație, de exemplu 0111.01*

Locality în Pastry

- pentru fiecare prefix, dintre toți vecinii potențiali, este selectat cel cu cel mai mic round-trip-time din rețeaua de bază
- deoarece prefixele scurte generează mai mulți candidați (împrăștiați în Internet), vecinii cu prefixe scurte sunt probabil să fie mai apropiați decât vecinii cu prefixe lungi
- în routing-ul de tip prefix, primele hop-uri sunt scurte, respectiv ultimele vor ajunge să fie lungi
- calea totală, comparată cu routing-ul Internet, este mai scurtă

Locality în Pastry

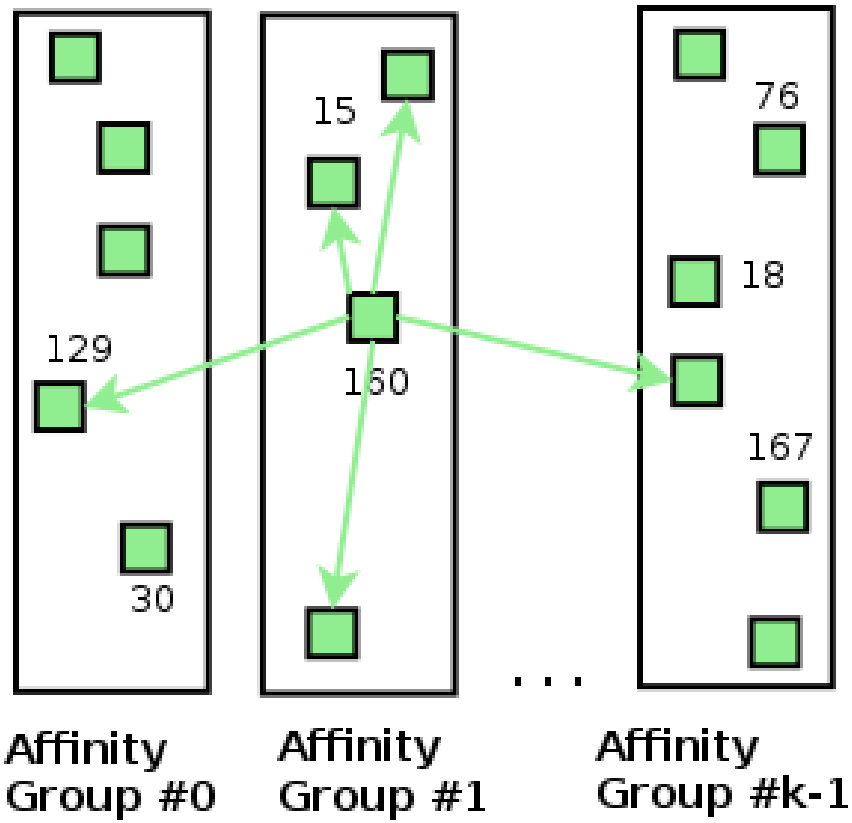


- se fac query-uri pentru nodurile din setul de prefixe
- dintre acesta se alege nodul cu ID-ul cel mai apropiat de target
- `http://www.scs.stanford.edu/nyu/04sp/notes/l24.pdf`

Chord și Pastry

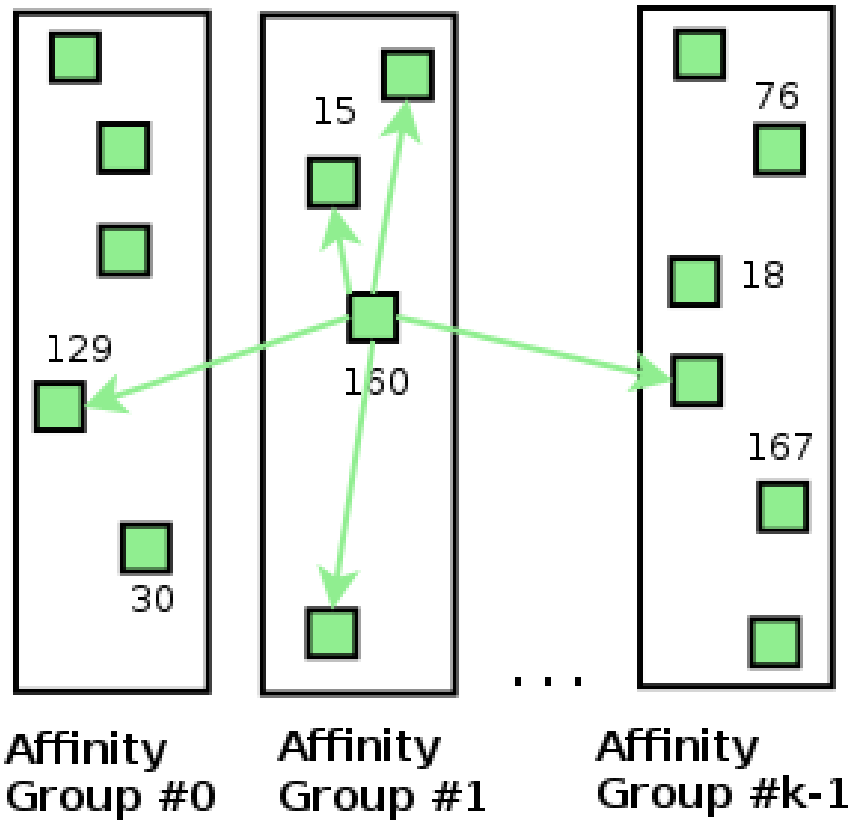
- mai structurate decât Gnutella
- sunt DHT
- algoritmii de lookup sunt de tip routing
- churn-ul ridicat determină o abordare complexă
- memoria este în O(log(N))
- cost-ul de lookup O(log(N)), nu este constant
- există un sistem cu un cost de lookup în O(1)?

Kelips - DHT lookup constant



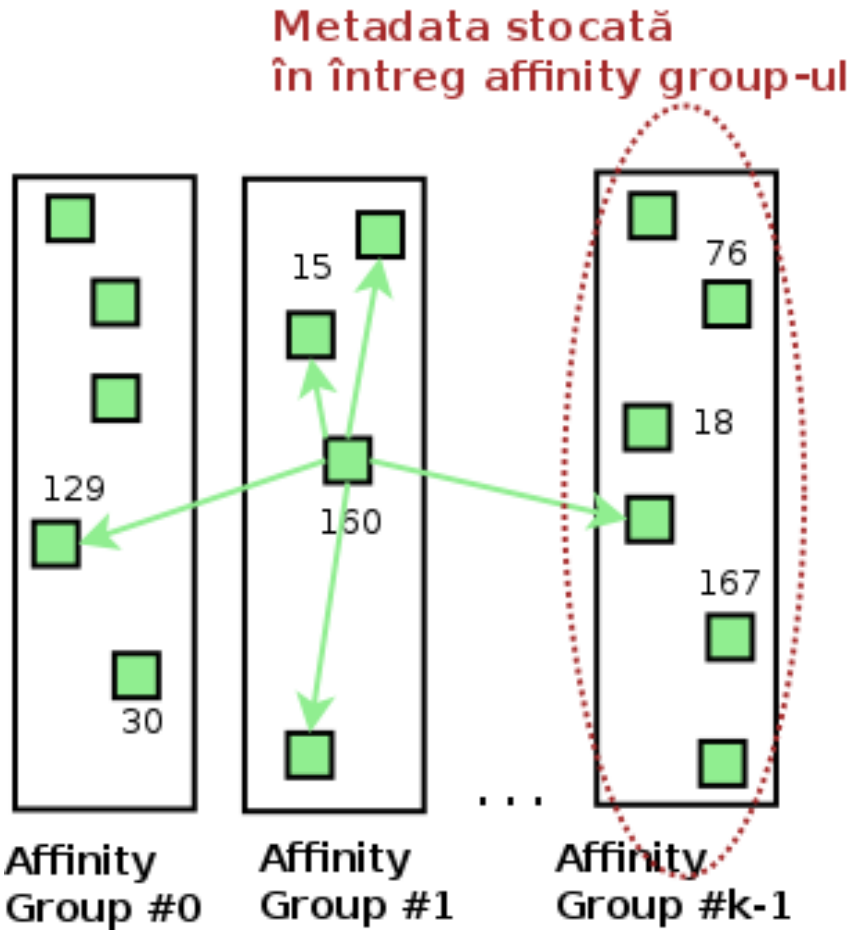
- sunt k affinity groups (AG)
- $k \approx \sqrt{N}$
- affinity group(node i) = SHA-1(i) modulo k
- vecinii unui nod:
  - (aproape) toate nodurile din AG-ul său
  - un singur nod de contact pentru celelalte AG-uri externe

Kelips - DHT lookup constant



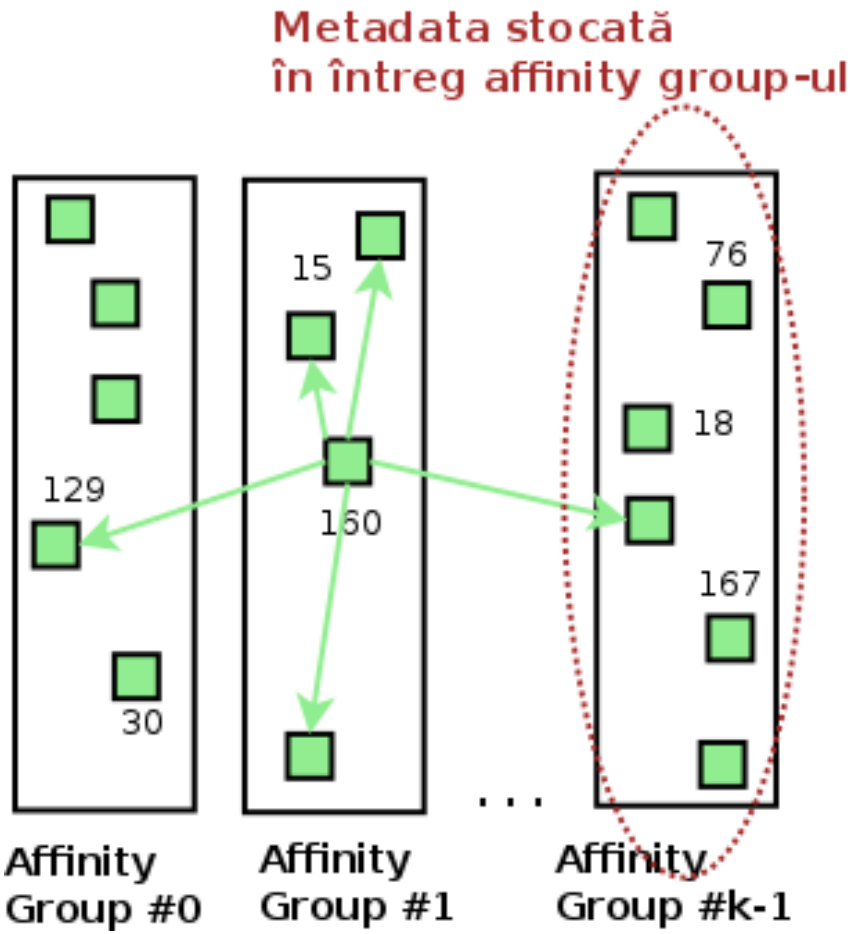
- $\sqrt{N}$ membri într-un AG
- $\sqrt{N}$ affinity groups
- un peer are $\sqrt{N}$ vecini în AG-ul său
- un peer are și $\sqrt{N} - 1$ contacte în alte AG-uri
- așadar, un peer va menține legături cu $2\sqrt{N}$ peers ($>$ log(N))

Fișiere în Kelips și metadata



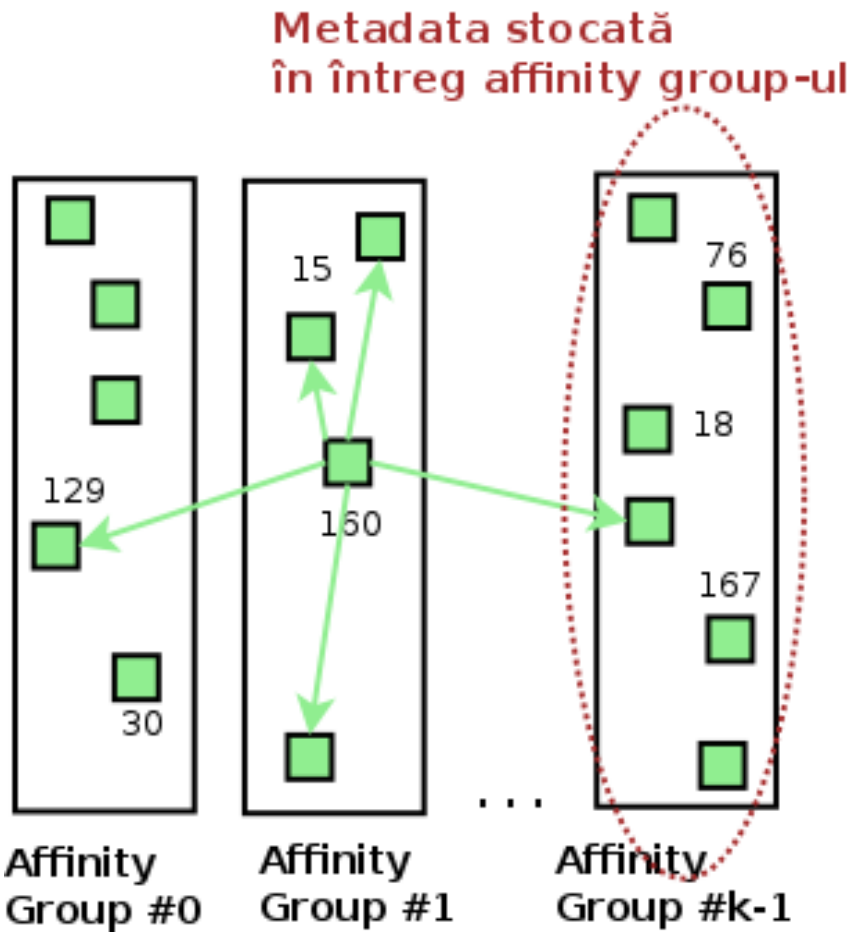
- fișierele pot fi stocate în oricare (câteva) noduri
- se decuplează replicarea și localizarea fișierului de query (nivel de indirectare)
- fiecare nume de fișier hashed pe un AG
- toate nodurile din AG replică metadatele (nume de fișier și locație)
- AG-ul NU stochează fișiere, doar metadatele (mici)

Lookup în Kelips



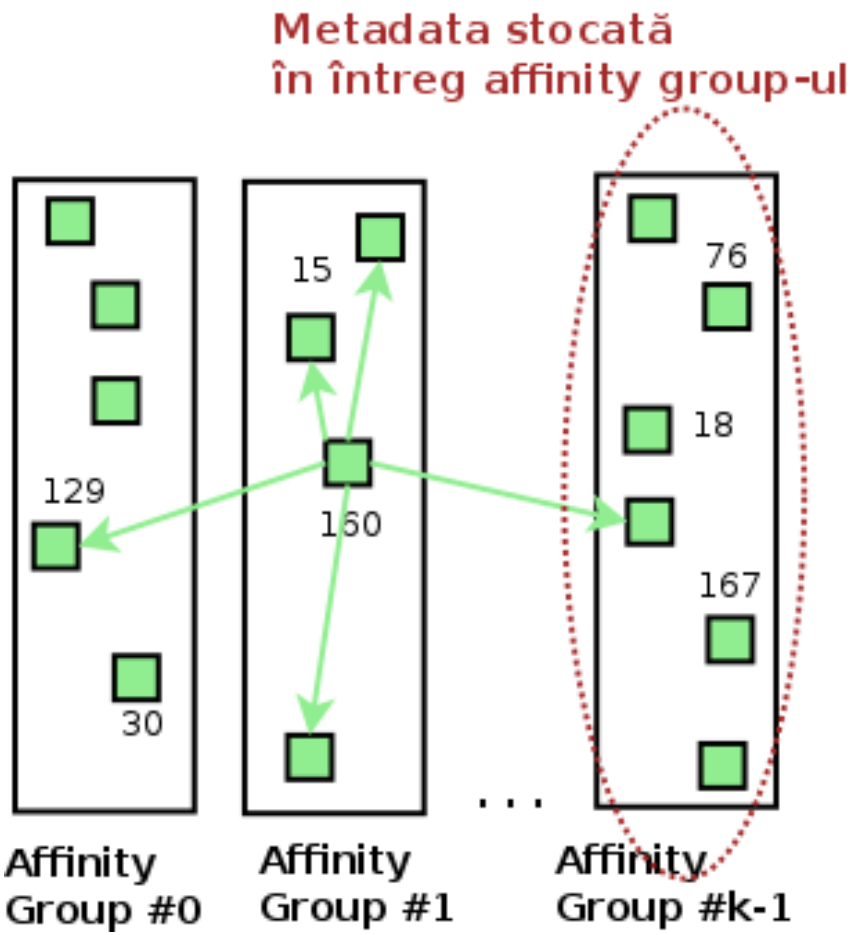
- lookup
  - caută AG-ul fișierului prin aplicarea hash(filename)
  - obține AG-ul fișierului de la contactul din grupul respectiv
  - în cazul în care contactul e căzut, va întreba unul din vecinii de AG cu privire la contactul aceluia în AG-ul căutat
  - lookup poate urca la 2
  - conectarea unui nod cu alt peer din alt AG se face pe hop-ul cel mai scurt (underlying net)

Lookup în Kelips



- lookup = 1 hop (sau câteva)
  - se obține informație rapid de la unul din vecini
  - per nod, costul de memorie este în $O(\sqrt{N})$
  - informația de membership este 1.93 MB pentru 100K noduri și 10M fișiere
  - mult mai mic decât disponibilitatea oricărei componente off-the-shelf

Kelips - membership și metadata



- liste de membership
  - diseminarea listelor de membership pt. AG-uri se realizează cu gossip
  - gossip în fiecare AG
  - gossip și între AGs
  - dissemination time O(log(N))
- metadata
  - trebuie actualizată periodic (gossip heartbeats de la sursă, pentru fiecare fișier)
  - în lipsa heartbeat-urilor, metadata (fișierul) se pierde după un timeout

Chord vs. Pastry vs. Kelips

- memory - cost de lookup - bandă cheltuită
- Kelips folosește ceva mai multă memorie decât Chord și Pastry și mai multă lățime de bandă, dar are lookup 1
- Chord și Pastry au O(log(N)) pentru memorie și lookup, Kelips are $O(\sqrt{N})$ pentru memorie, lookup în O(1)

Sumar

- sisteme P2P cu proprietăți demonstrate - problema DHT
  - Chord (MIT, Berkeley)
  - Pastry (MS Research, Rice University)
  - Kelips (Cornell University)