CS514: Intermediate Course in Operating Systems

Professor Ken Birman
Vivek Vishnumurthy: TA

Reminder: Distributed Hash Table (DHT)

- A service, distributed over multiple machines, with hash table semantics
  - *Insert*(key, value), Value(s) = *Lookup*(key)
- Designed to work in a peer-to-peer (P2P) environment
  - No central control
  - Nodes under different administrative control
- But of course can operate in an "infrastructure" sense

P2P "environment"

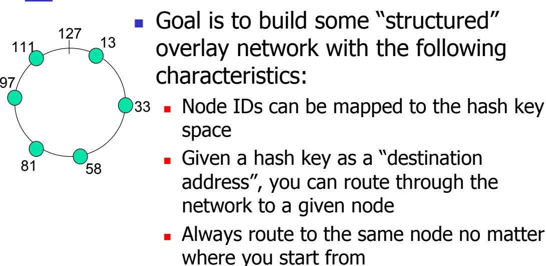
- Nodes come and go at will (possibly quite frequently---a few minutes)
- Nodes have heterogeneous capacities
  - Bandwidth, processing, and storage
- Nodes may behave badly
  - Promise to do something (store a file) and not do it (free-loaders)
  - Attack the system

Several flavors, each with variants

- Tapestry (Berkeley)
  - Based on Plaxton trees---similar to hypercube routing
  - The first* DHT
  - Complex and hard to maintain (hard to understand too!)
- CAN (ACIRI), Chord (MIT), and Pastry (Rice/MSR Cambridge)
  - Second wave of DHTs (contemporary with and independent of each other)

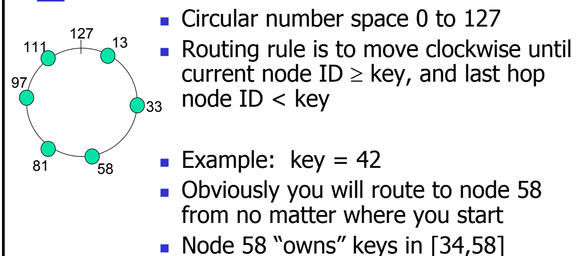
* Landmark Routing, 1988, used a form of DHT called Assured Destination Binding (ADB)

Basics of all DHTs



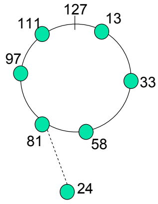
- Goal is to build some "structured" overlay network with the following characteristics:
  - Node IDs can be mapped to the hash key space
  - Given a hash key as a "destination address", you can route through the network to a given node
  - Always route to the same node no matter where you start from

Simple example (doesn't scale)



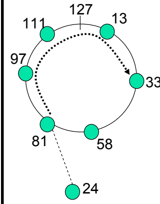
- Circular number space 0 to 127
- Routing rule is to move clockwise until current node ID ≥ key, and last hop node ID < key
- Example: key = 42
- Obviously you will route to node 58 from no matter where you start
- Node 58 "owns" keys in [34,58]

Building any DHT



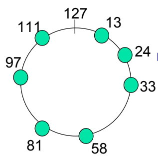
- Newcomer always starts with at least one known member

Building any DHT



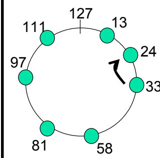
- Newcomer always starts with at least one known member
- Newcomer searches for "self" in the network
  - hash key = newcomer's node ID
  - Search results in a node in the vicinity where newcomer needs to be

Building any DHT



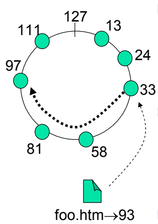
- Newcomer always starts with at least one known member
- Newcomer searches for "self" in the network
  - hash key = newcomer's node ID
  - Search results in a node in the vicinity where newcomer needs to be
- Links are added/removed to satisfy properties of network

Building any DHT



- Newcomer always starts with at least one known member
- Newcomer searches for "self" in the network
  - hash key = newcomer's node ID
- Search results in a node in the vicinity where newcomer needs to be
- Links are added/removed to satisfy properties of network
- Objects that now hash to new node are transferred to new node

Insertion/lookup for any DHT



- Hash name of object to produce key
  - Well-known way to do this
- Use key as destination address to route through network
  - Routes to the target node
- Insert object, or retrieve object, at the target node

Properties of all DHTs

- Memory requirements grow (something like) logarithmically with N (exception: Kelips)
- Routing path length grows (something like) logarithmically with N (several exceptions)
- Cost of adding or removing a node grows (something like) logarithmically with N
- Has caching, replication, etc…

DHT Issues

- Resilience to failures
- Load Balance
  - Heterogeneity
  - Number of objects at each node
  - Routing hot spots
  - Lookup hot spots
- Locality (performance issue)
- Churn (performance and correctness issue)
- Security

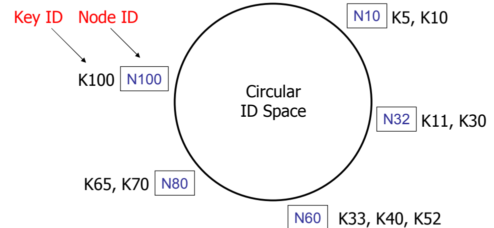
We're going to look at four DHTs

- At varying levels of detail…
  - Chord
    - MIT (Stoica *et al*)
  - Kelips
    - Cornell (Gupta, Linga, Birman)
  - Pastry
    - Rice/Microsoft Cambridge (Druschel, Rowstron)
  - Behive
    - Cornell (Ramasubramanian, Sirer)

Things we're going to look at

- What is the structure?
- How does routing work in the structure?
- How does it deal with node departures?
- How does it scale?
- How does it deal with locality?
- What are the security issues?

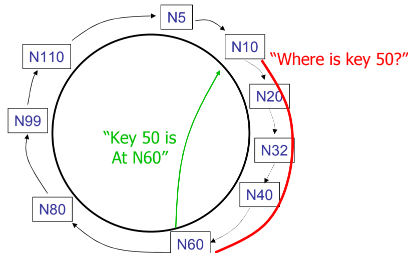
Chord uses a circular ID space



- Successor: node with next highest ID

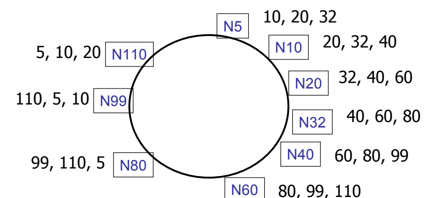
Chord slides care of Robert Morris, MIT

Basic Lookup



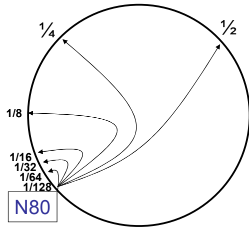
- Lookups find the ID's successor
- Correct if predecessor is correct

Successor Lists Ensure Robust Lookup



- Each node remembers r successors
- Lookup can skip over dead nodes to find blocks
- Periodic check of successor and predecessor links

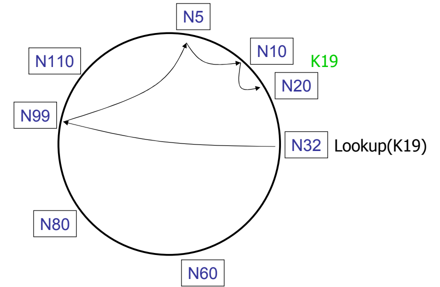
Chord "Finger Table" Accelerates Lookups



To build finger tables, new node searches for the key values for each finger

To do it efficiently, new nodes obtain successor's finger table, and use as a hint to optimize the search

Chord lookups take O(log N) hops



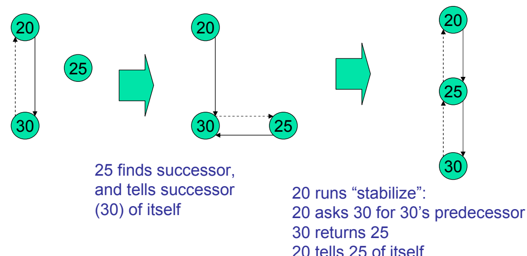
Drill down on Chord reliability

- Interested in maintaining a correct routing table (successors, predecessors, and fingers)
- Primary invariant: correctness of successor pointers
  - Fingers, while important for performance, do not have to be exactly correct for routing to work
  - Algorithm is to "get closer" to the target
  - Successor nodes always do this

Maintaining successor pointers

- Periodically run "stabilize" algorithm
  - Finds successor's predecessor
  - Repair if this isn't self
- This algorithm is also run at join
- Eventually routing will repair itself
- Fix_finger also periodically run
  - For randomly selected finger

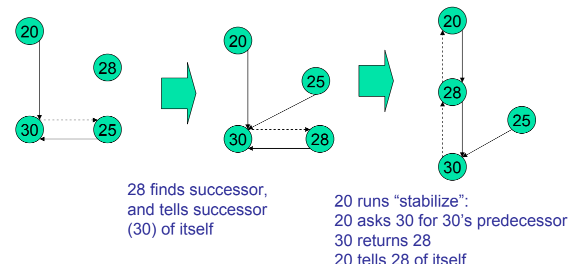
Initial: 25 wants to join correct ring (between 20 and 30)



25 finds successor, and tells successor (30) of itself

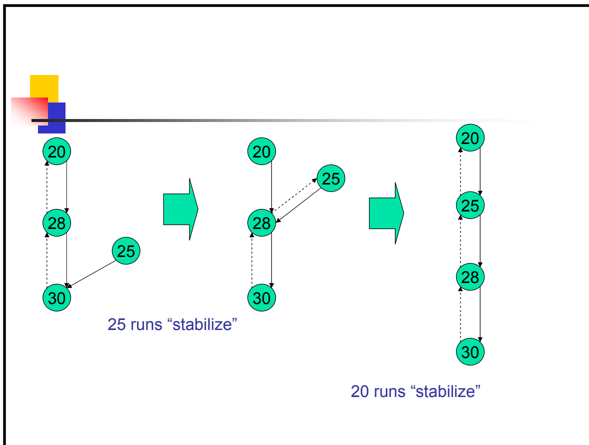
20 runs "stabilize":
20 asks 30 for 30's predecessor
30 returns 25
20 tells 25 of itself

This time, 28 joins before 20 runs "stabilize"



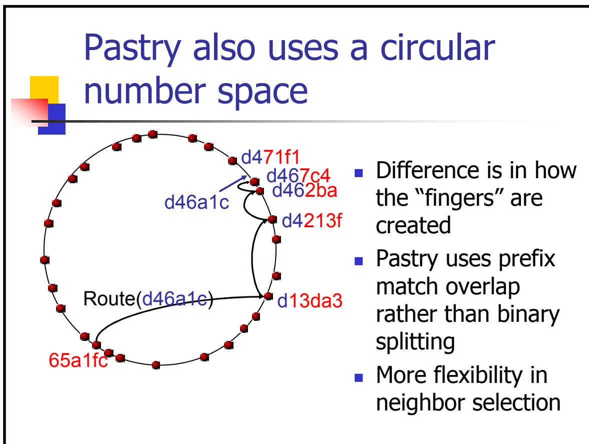
28 finds successor, and tells successor (30) of itself

20 runs "stabilize":
20 asks 30 for 30's predecessor
30 returns 28
20 tells 28 of itself



Chord problems?

- With intense “churn” ring may be very disrupted
  - Worse case: partition can provoke formation of two distinct rings (think “Cornell ring” and “MIT ring”)
  - And they could have finger pointers to each other, but not link pointers
  - This might never heal itself…
  - But scenario would be hard to induce.
- Unable to resist Sybil attacks



Pastry routing table (for node 65a1fc)

0x	1x	2x	3x	4x	5x		7x	8x	9x	ax	bx	cx	dx	ex	fx
60x	61x	62x	63x	64x		66x	67x	68x	69x	6ax	6bx	6cx	6dx	6ex	6fx
650x	651x	652x	653x	654x	655x	656x	657x	658x	659x		65bx	65cx	65dx	65ex	65fx
65a0x		65a2x	65a3x	65a4x	65a5x	65a6x	65a7x	65a8x	65a9x	65aax	65abx	65acx	65adx	65aex	65afx

Pastry nodes also have a “leaf set” of immediate neighbors up and down the ring

Similar to Chord’s list of successors

Pastry join

- X = new node, A = bootstrap, Z = nearest node
- A finds Z for X
- In process, A, Z, and all nodes in path send state tables to X
- X settles on own table
  - Possibly after contacting other nodes
- X tells everyone who needs to know about itself
- Pastry paper doesn’t give enough information to understand how concurrent joins work
  - 18th IFIP/ACM, Nov 2001

Pastry leave

- Noticed by leaf set neighbors when leaving node doesn’t respond
  - Neighbors ask highest and lowest nodes in leaf set for new leaf set
- Noticed by routing neighbors when message forward fails
  - Immediately can route to another neighbor
  - Fix entry by asking another neighbor in the same “row” for its neighbor
  - If this fails, ask somebody a level up

For instance, this neighbor fails

Ask other neighbors

Try asking some neighbor in the same row for its 655x entry

If it doesn't have one, try asking some neighbor in the row below, etc.

CAN, Chord, Pastry differences

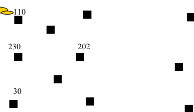
- CAN, Chord, and Pastry have deep similarities
- Some (important???) differences exist
  - We didn't look closely at it, but CAN nodes tend to know of multiple nodes that allow equal progress
    - Can therefore use additional criteria (RTT) to pick next hop
  - Pastry allows greater choice of neighbor
    - Can thus use additional criteria (RTT) to pick neighbor
  - In contrast, Chord has more determinism
    - Some recent work exploits this to make Chord tolerant of Byzantine attacks (but cost is quite high)

Kelips takes a different approach

- Network partitioned into √N "affinity groups"
- Hash of node ID determines which affinity group a node is in
- Each node knows:
  - One or more nodes in each group
  - All objects and nodes in own group
- *But this knowledge is soft-state, spread through peer-to-peer "gossip" (epidemic multicast)!*

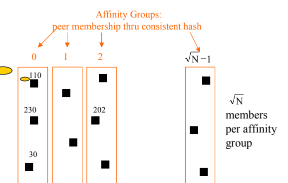
Kelips

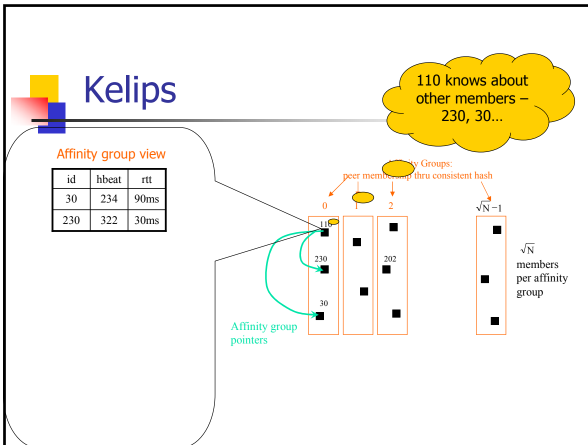
Take a a collection of "nodes"

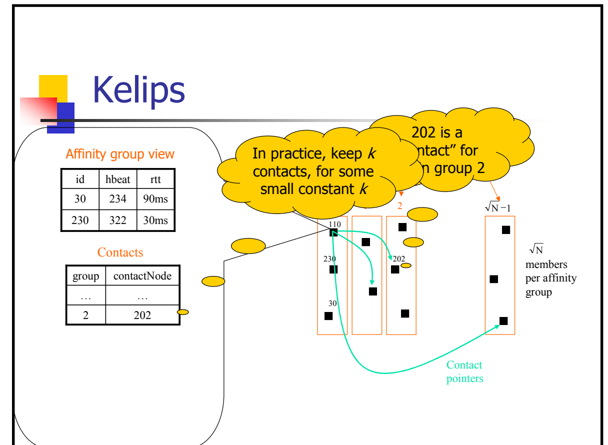


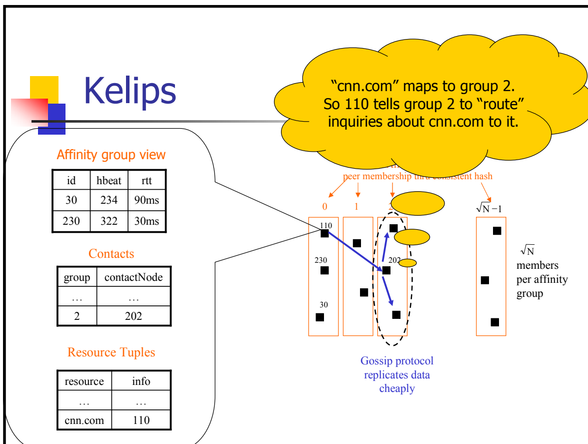
Kelips

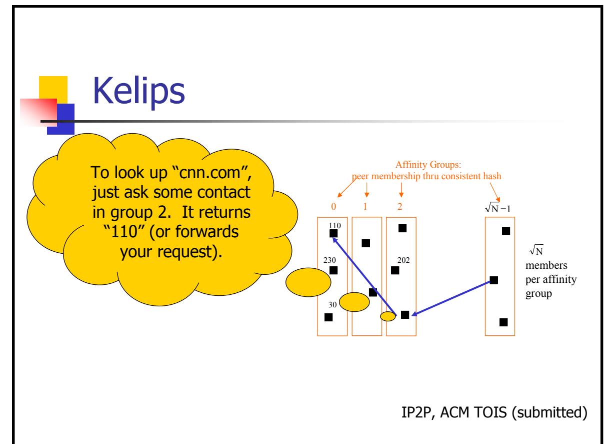
Map nodes to affinity groups











Kelips gossip

- Operates at constant “background” rate
  - Independent of frequency of changes in the system
  - Average overhead may be higher than other DHTs, but not bursty
- If churn too high, system performs poorly (failed lookups), *but does not collapse…*

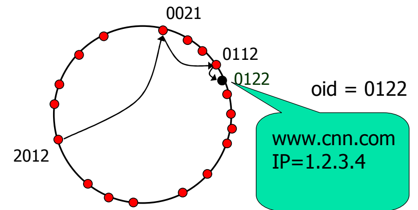
Beehive

- A DHT intended for supporting righ-performance infrastructure services with proactive caching
  - Focus of “real system” is on DNS but has other applications
  - Proactive caching: a form of replication. DNS already caches… Beehive pushes updates to improve hit rates

Domain Name Service

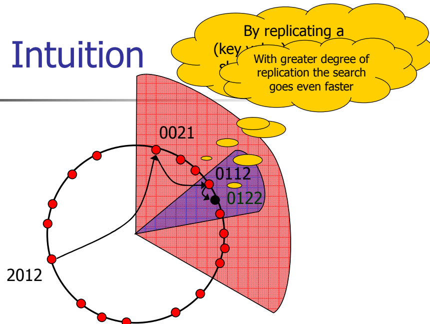
- Translates textual names to internet addresses
  - “www.cnn.com” -> 1.2.3.4
- Relies on a static hierarchy of servers
  - Prone to DoS attacks
  - Fragile, expensive to maintain
  - Slow and sluggish
- In 2002, a DDoS attack almost disabled the root name servers

A Self-Organizing Solution…



- Great idea, but O(log N) is too slow on the Internet

Beehive Intuition



Optimization problem: Minimize total number of replicas s.t., average lookup performance ≤ C

Beehive Summary

- general replication framework
- suitable for structured DHTs
  - decentralization, self-organization, resilience
- properties
  - high performance: O(1) average lookup time
  - scalable: minimize number of replicas and reduce storage, bandwidth, and network load
  - adaptive: promptly respond to changes in popularity – flash crowds

Analytical Problem

Minimize (storage/bandwidth)

$$x_0 + x_1/b + x_2/b^2 + \ldots + x_{K-1}/b^{K-1}$$

such that (average lookup time is C hops)

$$(x_0^{1-\alpha} + x_1^{1-\alpha} + x_2^{1-\alpha} + \ldots + x_{K-1}^{1-\alpha}) \geq K - C$$

and

$$x_0 \leq x_1 \leq x_2 \leq \ldots \leq x_{K-1} \leq 1$$

b: base K: $\log_b(N)$

x_j: fraction of objects replicated at level j or lower

Optimal Solution

$$x^*_j = \left[\frac{d^j (K' - C)}{1 + d + \ldots + d^{K'-1}} \right]^{\frac{1}{1-\alpha}} \qquad 0 \leq j \leq K' - 1$$

$$d = b^{(1-\alpha)/\alpha}$$

$$x^*_j = 1 \qquad K' \leq j \leq K$$

K’ is determined by setting (typically 2 or 3)

$$x^*_{K'-1} \leq 1 \quad \Rightarrow \quad d^{K'-1} (K' - C) / (1 + d + \ldots + d^{K'-1}) \leq 1$$

optimal per node storage

$$(1 - 1/b) / (1 + d + \ldots + d^{K'-1})^{\alpha/(1-\alpha)} + 1/b^{K'}$$

analytical model

- optimization problem

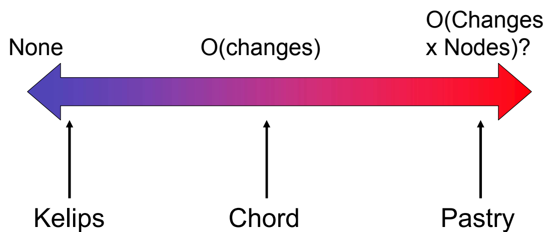
  minimize: total number of replicas, s.t.,

  average lookup performance ≤ C
- configurable target lookup performance
  - continuous range, sub one-hop
- minimizing number of replicas decreases storage and bandwidth overhead

CoDoNS

- Sirer's group built an alternative to DNS
  - Safety net and replacement for legacy DNS
  - Self-organizing, distributed
- Deployed across the globe
- Achieves *better average response time* than DNS
- See NSDI '04, SIGCOMM '04 for details

Control traffic load generated by churn



DHT applications

- Company could use DHT to create a big index of "our worldwide stuff"
- Beehive: Rebuilds DNS as a DHT application
  - No structure required in name!
  - Fewer administration errors
  - No DoS target
- But benefits aren't automatic. Contrast with DNS over Chord
  - Median lookup time increased from 43ms (DNS) to 350ms

To finish up

- Various applications have been designed over DHTs
  - File system, DNS-like service, pub/sub system
- DHTs are elegant and promising tools
- Concerns about churn and security