Distributed Hash Tables

Ken Birman

Cornell University. CS5410 Fall 2008.

What is a Distributed Hash Table (DHT)?

- Exactly that ☺
- A service, distributed over multiple machines, with hash table semantics
  - *Put*(key, value), Value = *Get*(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

More specifically

- Hash table semantics:
  - *Put*(key, value),
  - Value = *Get*(key)
  - Key is a single flat string
  - Limited semantics compared to keyword search
- *Put*() causes value to be stored at one (or more) peer(s)
- *Get*() retrieves value from a peer
- *Put*() and *Get*() accomplished with unicast routed messages
  - In other words, it scales
- Other API calls to support application, like notification when neighbors come and go

P2P “environment”

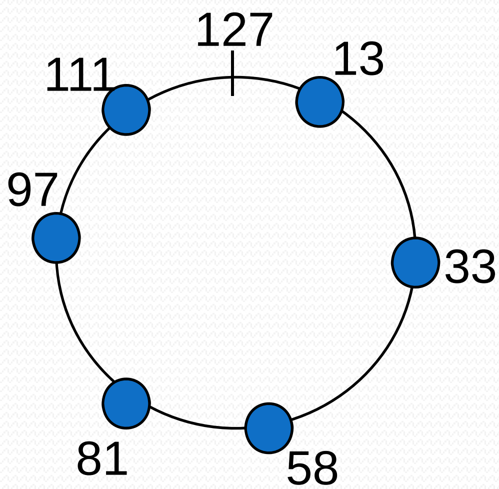
- Nodes come and go at will (possibly quite frequently---a few minutes)
- Nodes have heterogeneous capacities
  - Bandwidth, processing, and storage
- Nodes may behave badly
  - Promise to do something (store a file) and not do it (free-loaders)
  - Attack the system

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)

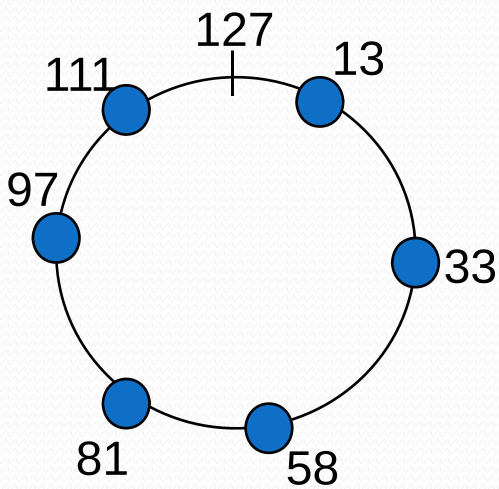
* Landmark Routing, 1988, used a form of DHT called Assured Destination Binding (ADB)

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

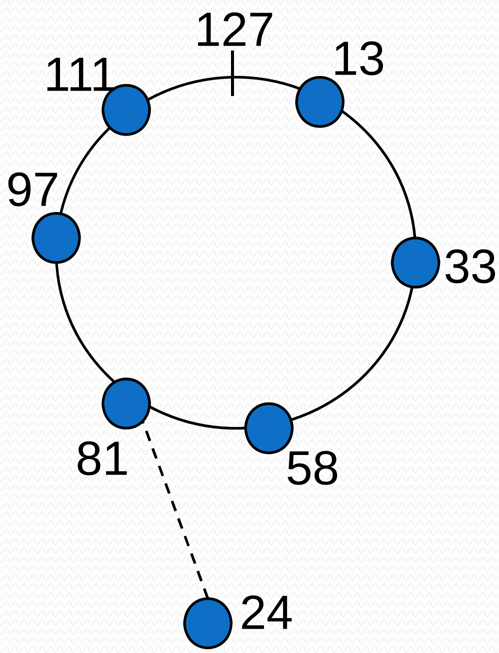
Simple example (doesn't scale)



- Circular number space 0 to 127
- Routing rule is to move counter-clockwise until current node ID ≥ key, and last hop node ID < key

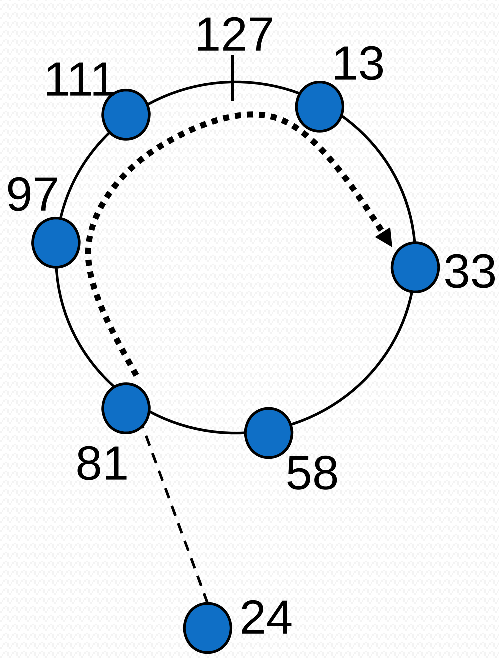
- Example: key = 42
- Obviously you will route to node 58 from no matter where you start

Building any DHT



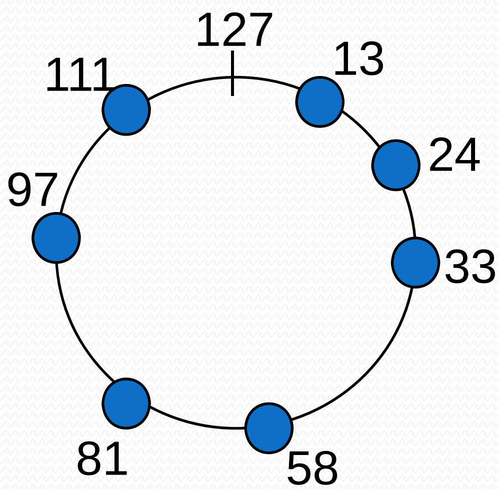
- Newcomer always starts with at least one known member

Building any DHT



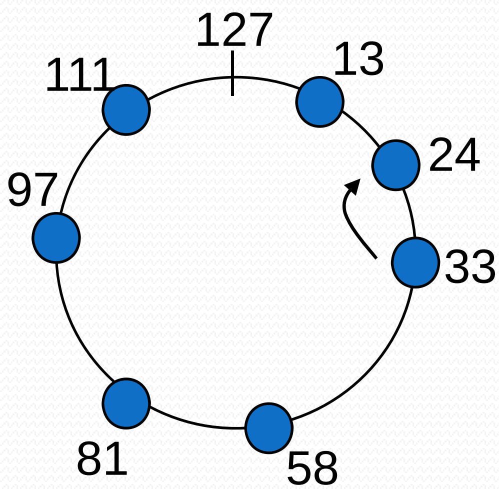
- Newcomer always starts with at least one known member
- Newcomer searches for “self” in the network
  - hash key = newcomer’s node ID
  - Search results in a node in the vicinity where newcomer needs to be

Building any DHT



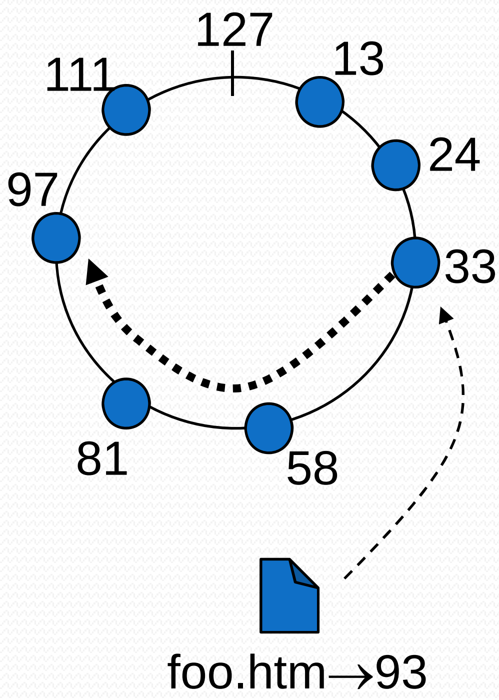
- Newcomer always starts with at least one known member
- Newcomer searches for “self” in the network
  - hash key = newcomer’s node ID
  - Search results in a node in the vicinity where newcomer needs to be
- Links are added/removed to satisfy properties of network

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 most DHTs

- Memory requirements grow (something like) logarithmically with N
- Routing path length grows (something like) logarithmically with N
- 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

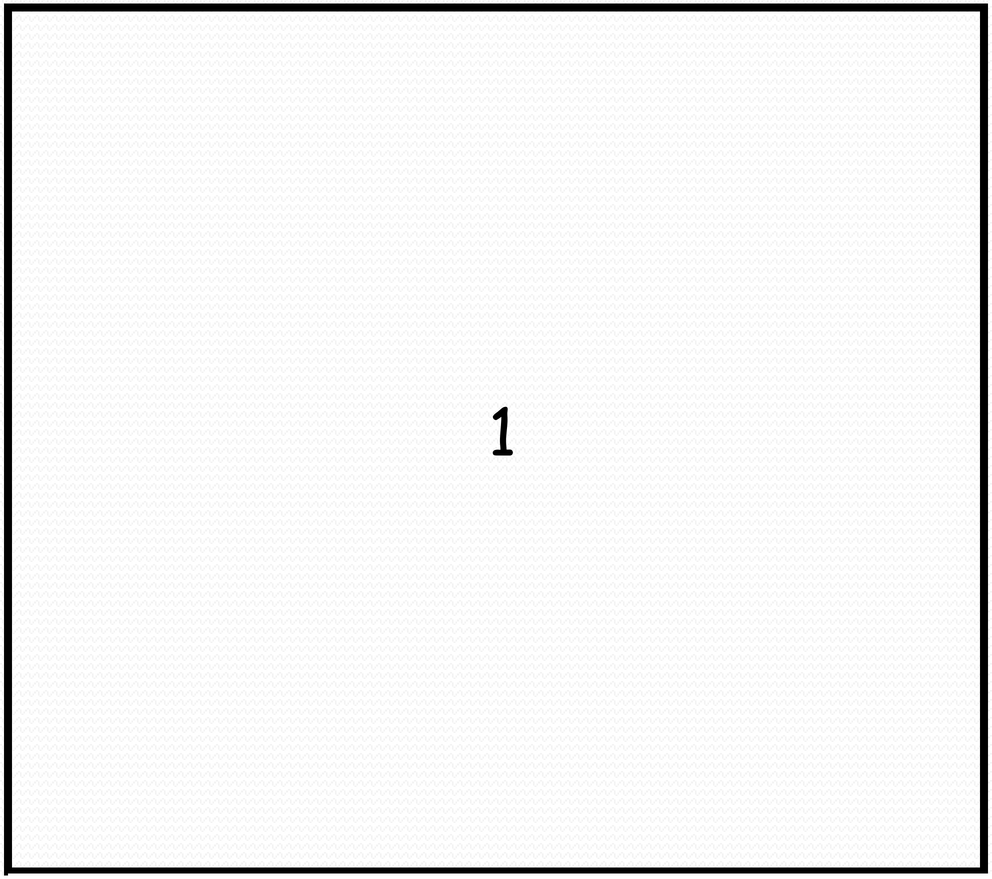
We’re going to look at four DHTs

- At varying levels of detail…
  - CAN (Content Addressable Network)
    - ACIRI (now ICIR)
  - Chord
    - MIT
  - Kelips
    - Cornell
  - Pastry
    - Rice/Microsoft Cambridge

Things we're going to look at

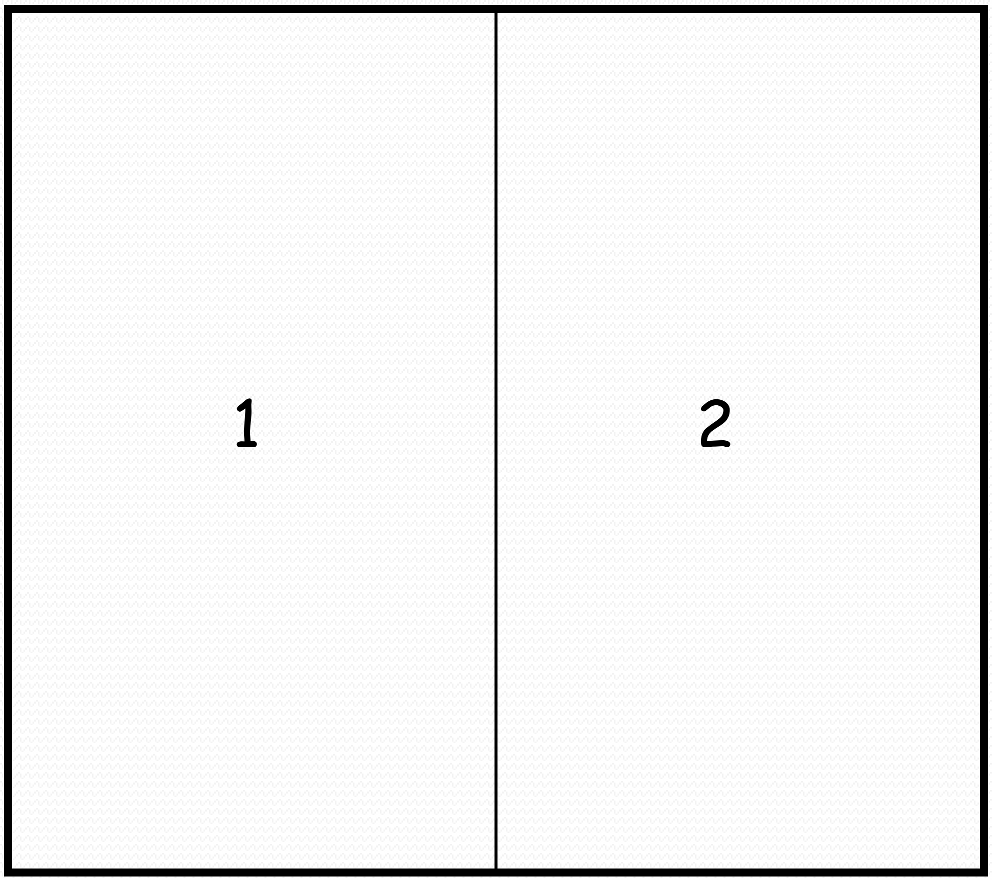
- What is the structure?
- How does routing work in the structure?
- How does it deal with node departures?
- How does it scale?
- How does it deal with locality?
- What are the security issues?

CAN structure is a cartesian coordinate space in a D dimensional torus

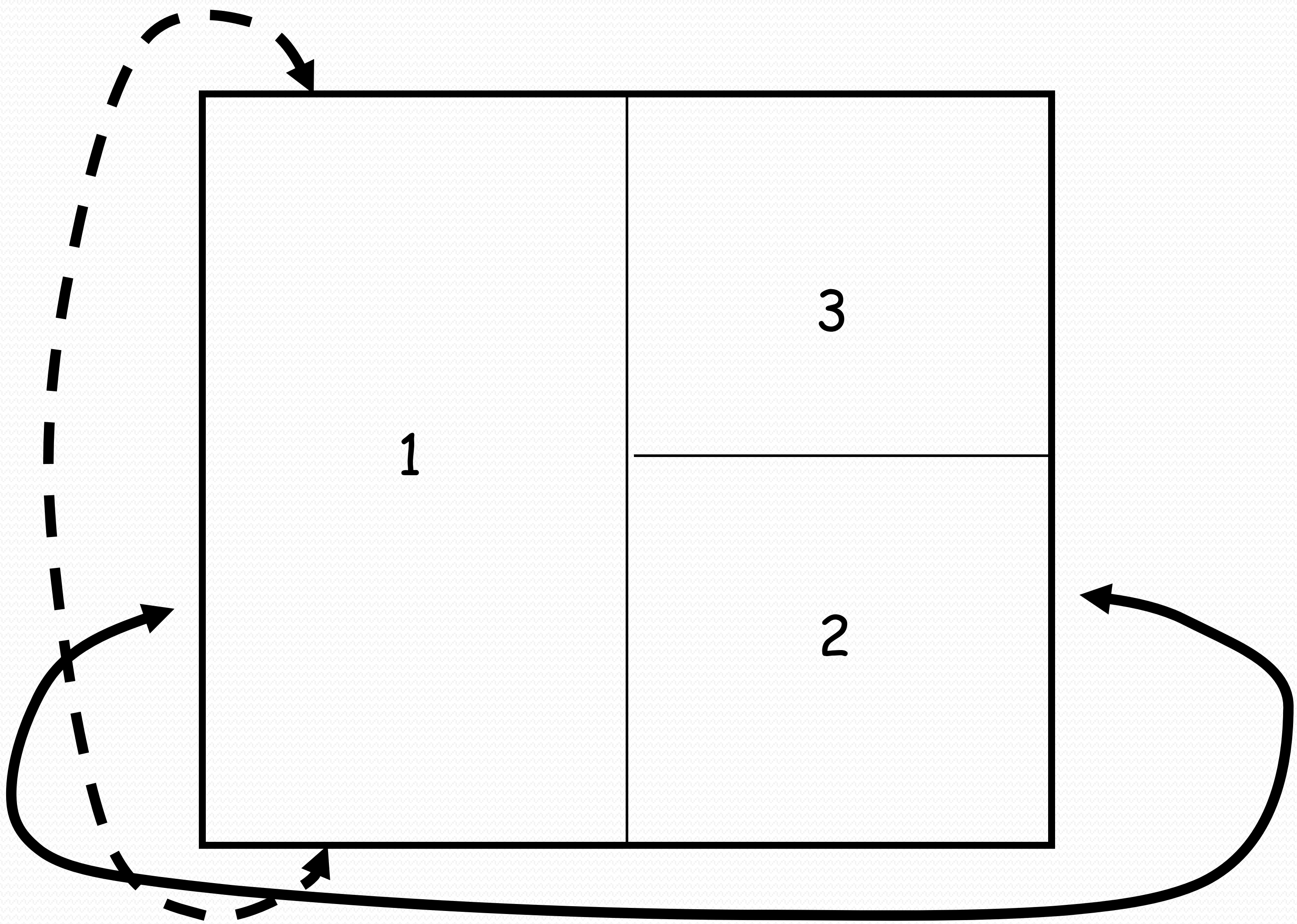


CAN graphics care of Santashil PalChaudhuri, Rice Univ

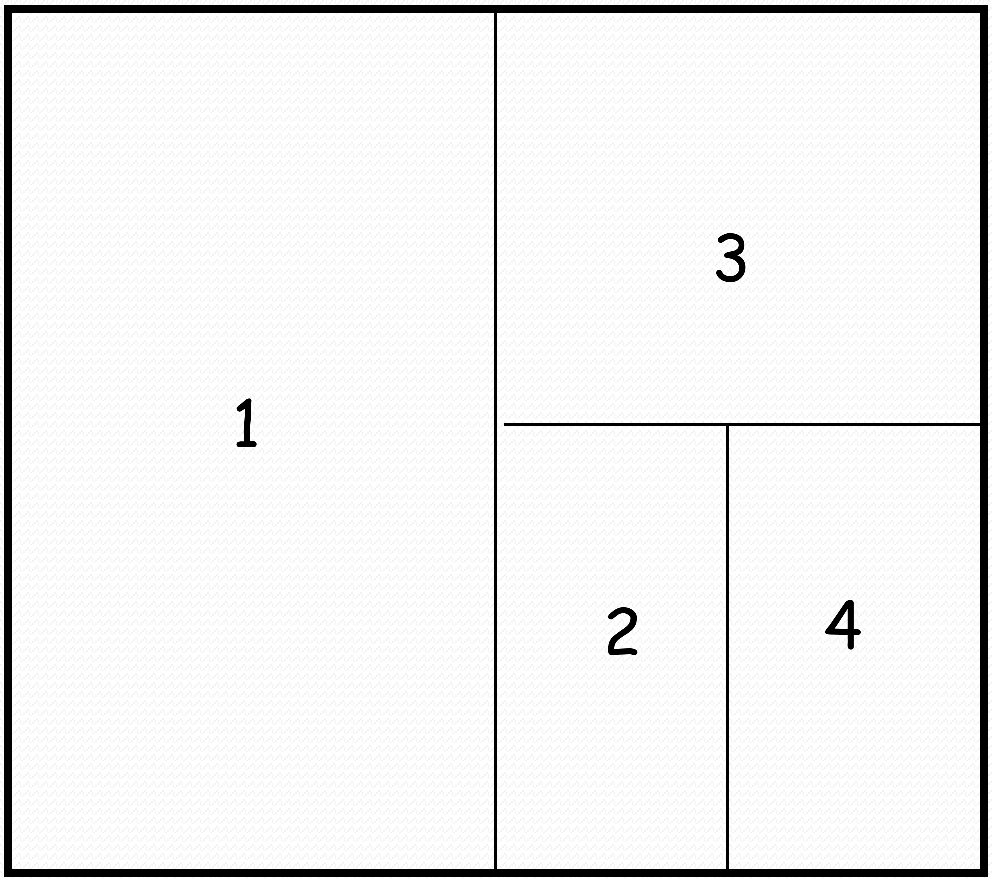
Simple example in two dimensions



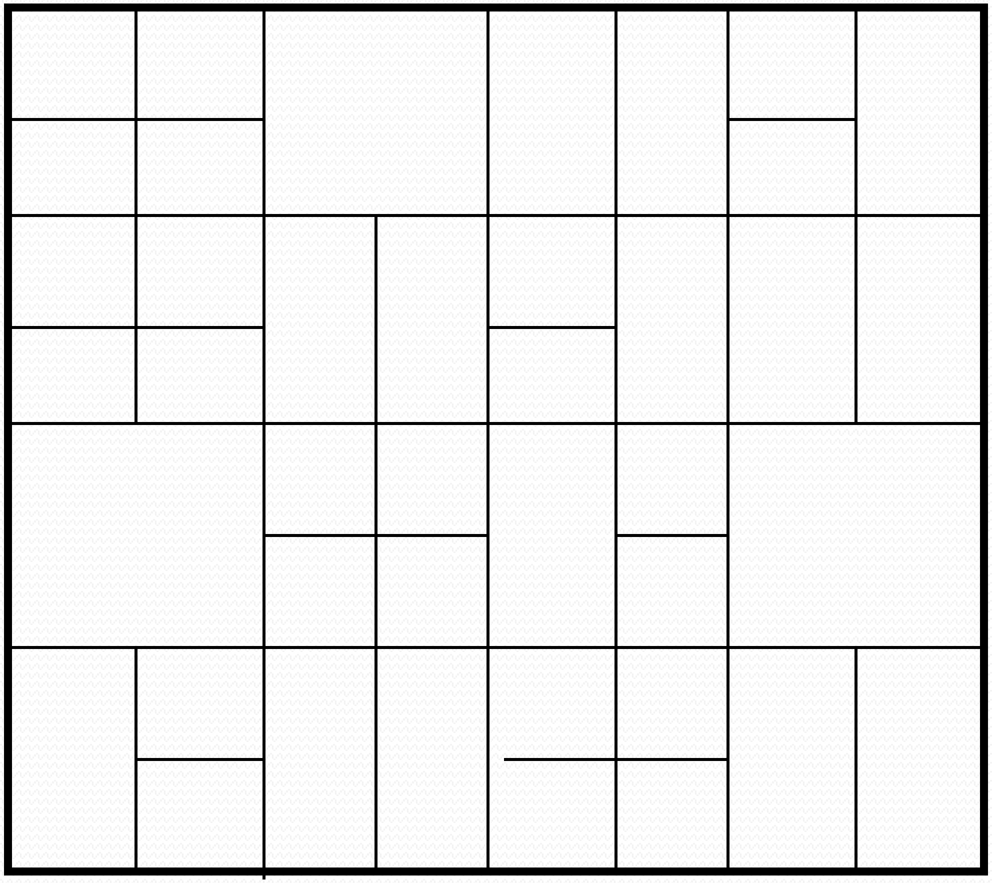
Note: torus wraps on "top" and "sides"



Each node in CAN network occupies a “square” in the space

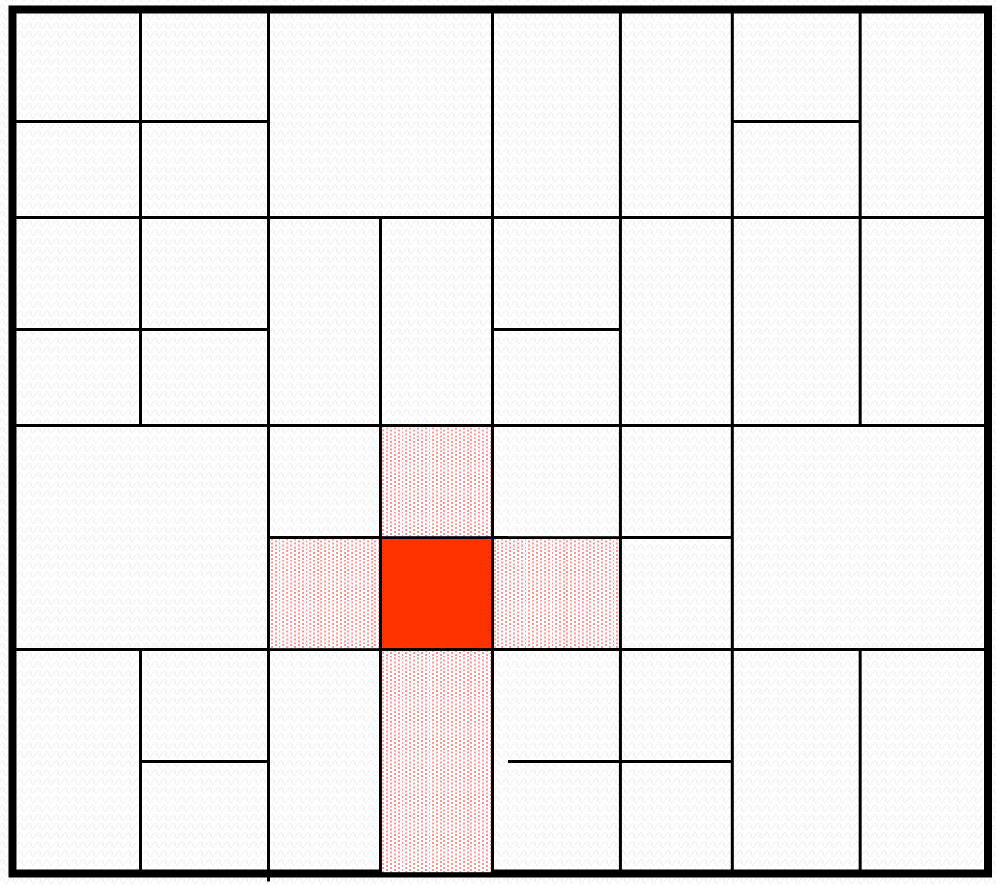


With relatively uniform square sizes



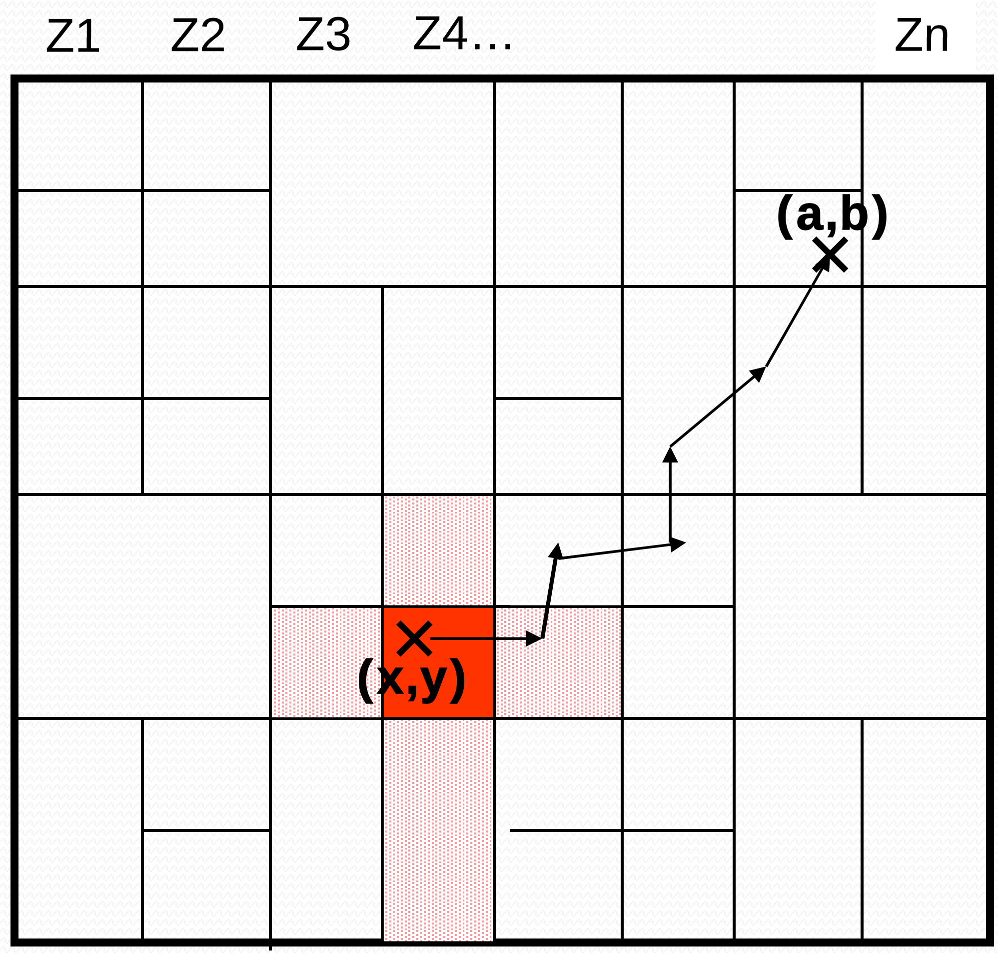
Neighbors in CAN network

- Neighbor is a node that:
- Overlaps d-1 dimensions
- Abuts along one dimension

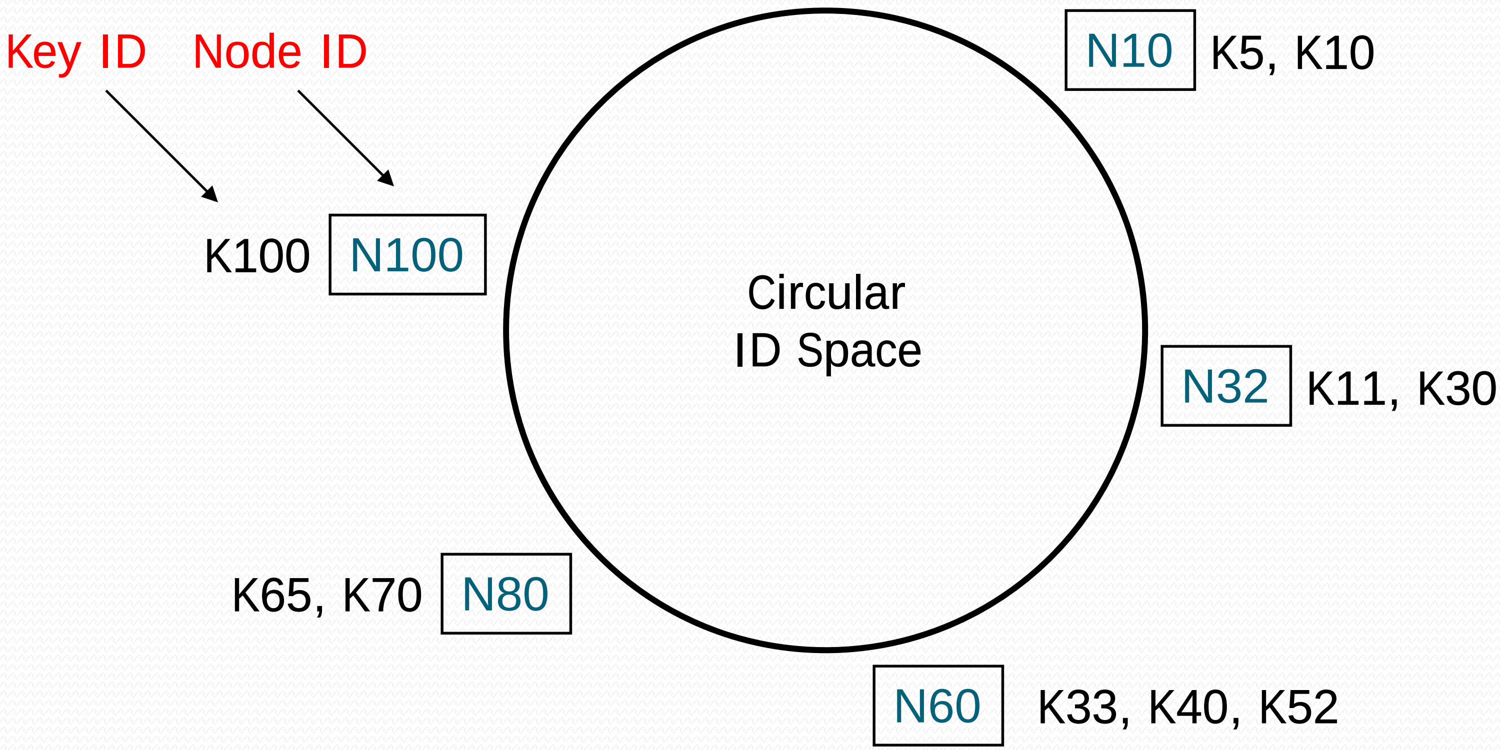


Route to neighbors closer to target

- d-dimensional space
- n zones
  - Zone is space occupied by a "square" in one dimension
- Avg. route path length
  - $(d/4)(n^{1/d})$
- Number neighbors = O(d)
- Tunable (vary d or n)
- Can factor proximity into route decision



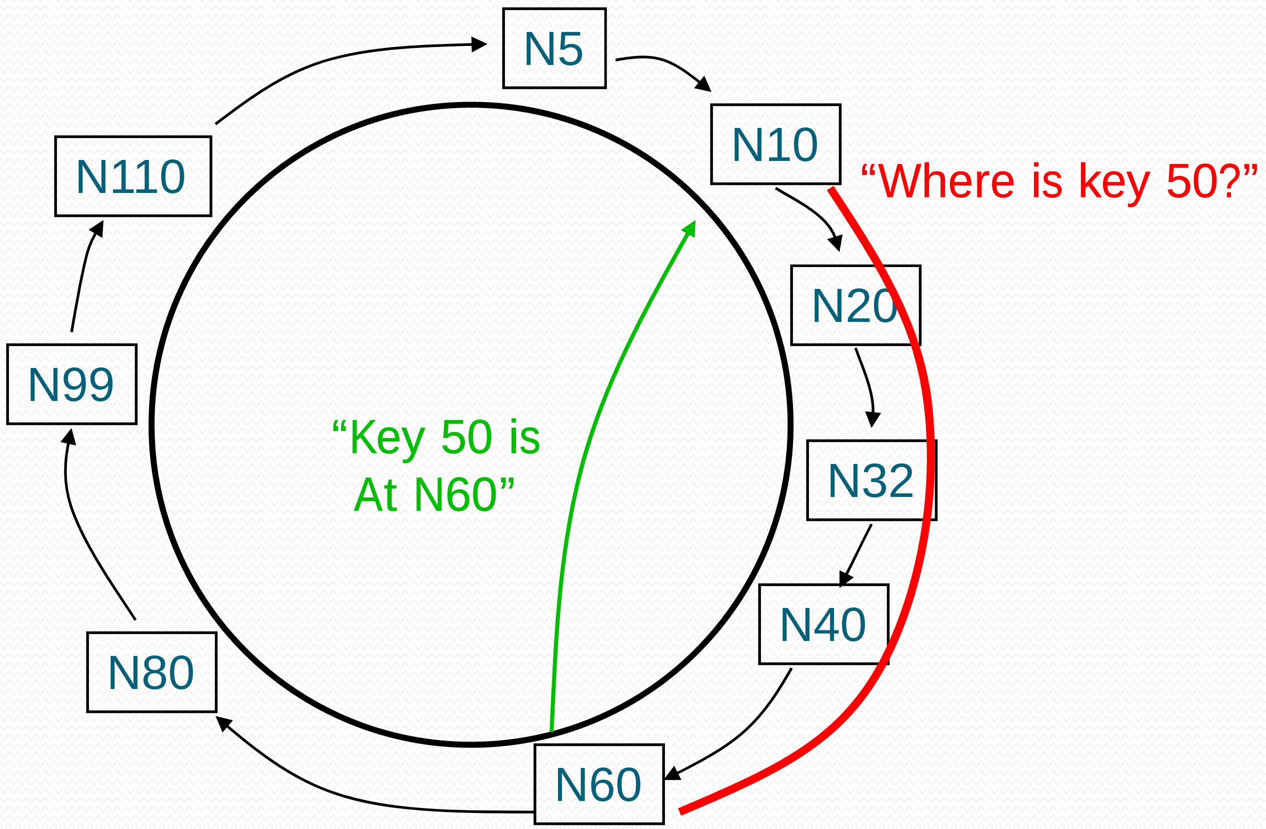
Chord uses a circular ID space



- Successor: node with next highest ID

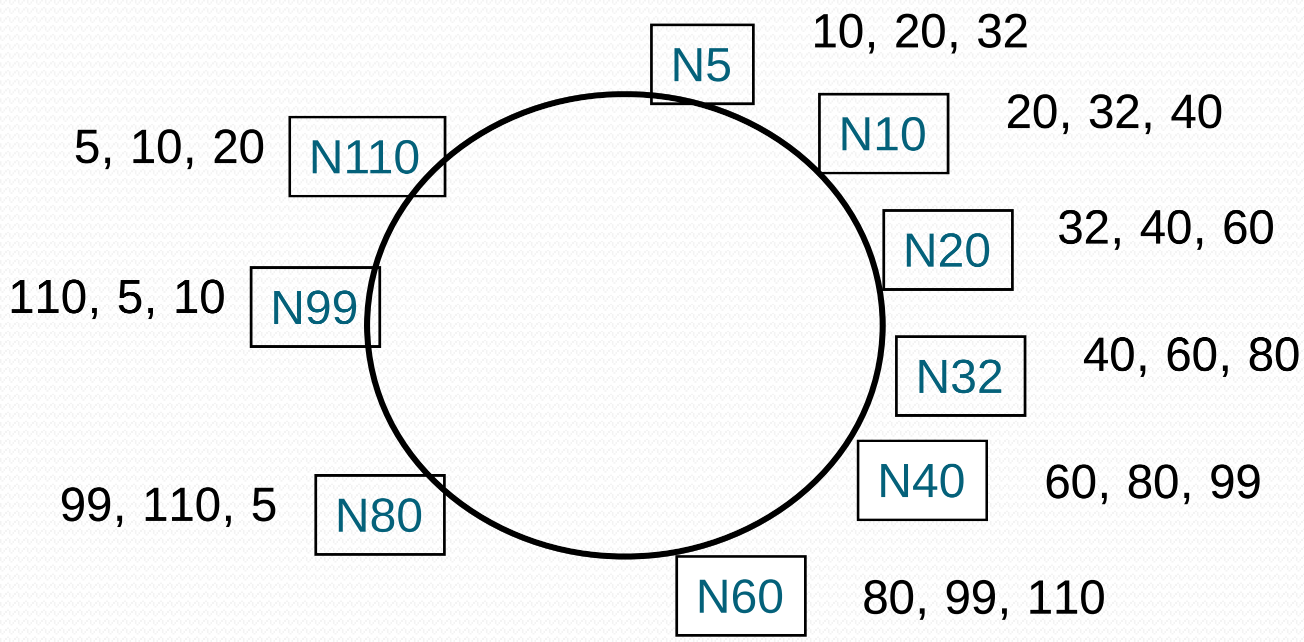
Chord slides care of Robert Morris, MIT

Basic Lookup



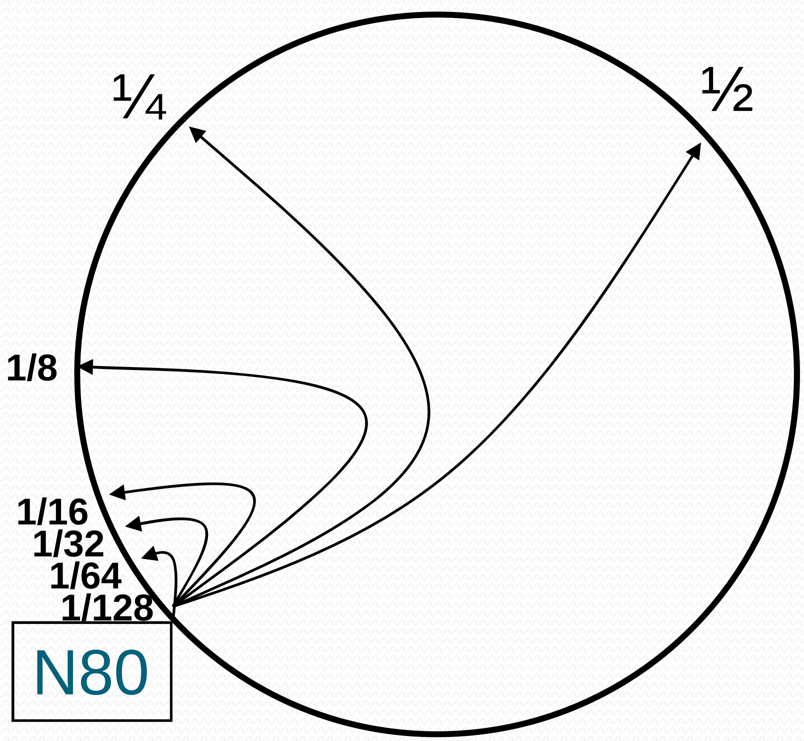
- Lookups find the ID's predecessor
- Correct if successors are correct

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

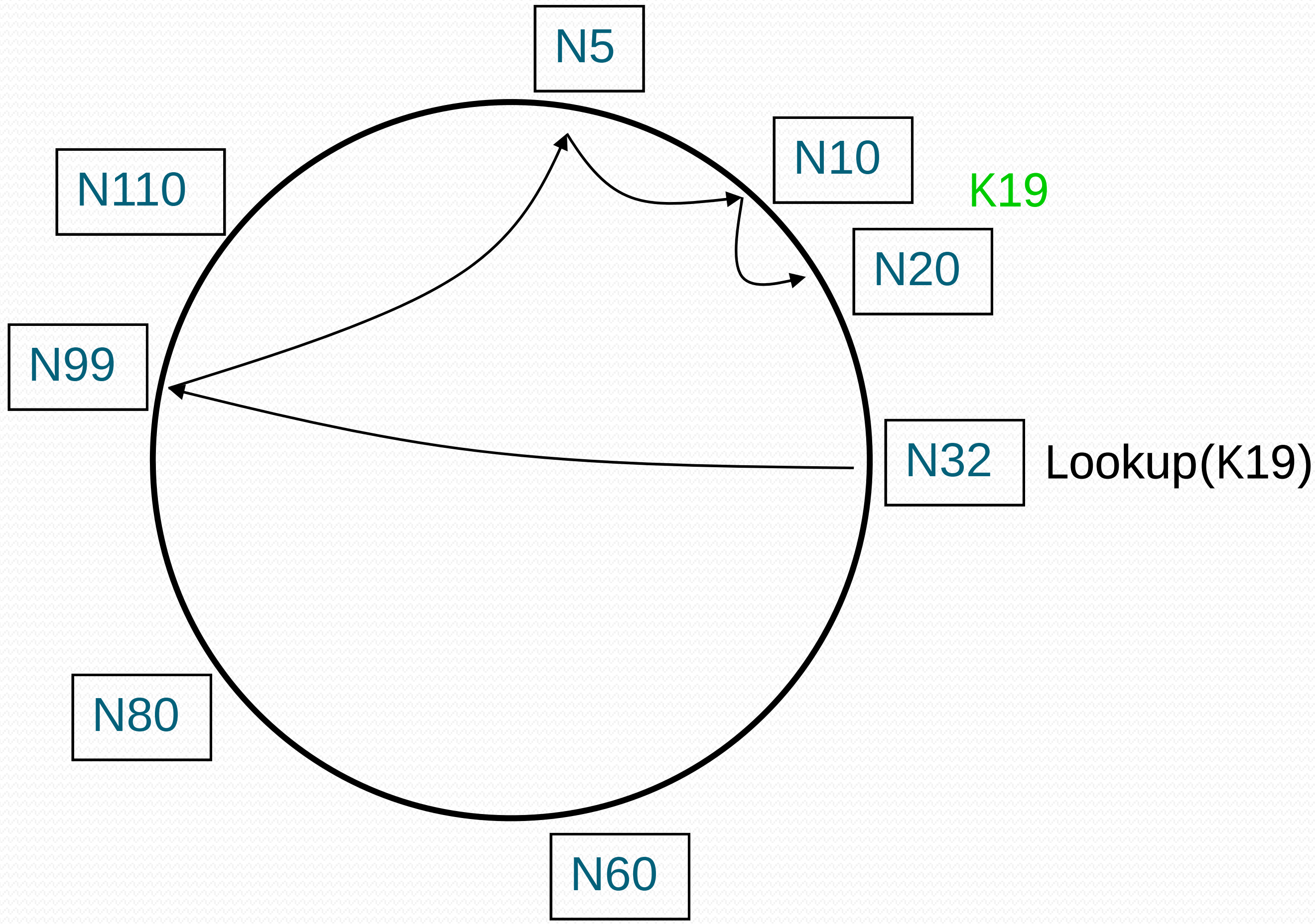
Chord "Finger Table" Accelerates Lookups



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



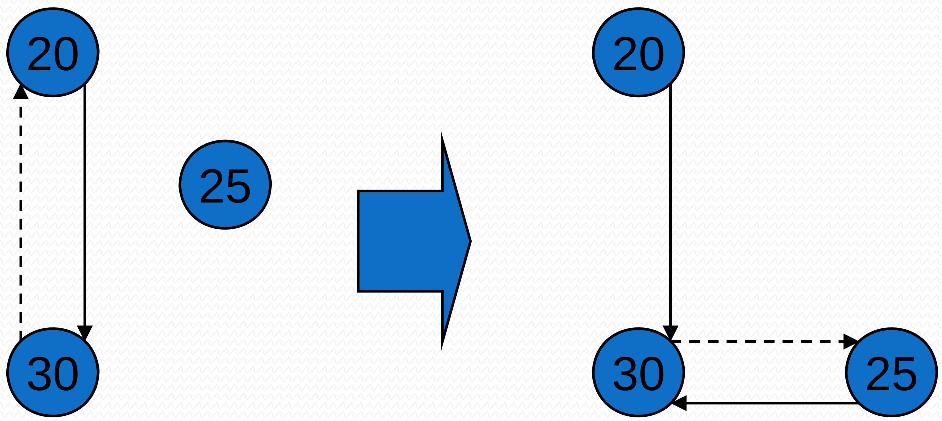
Drill down on Chord reliability

- 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

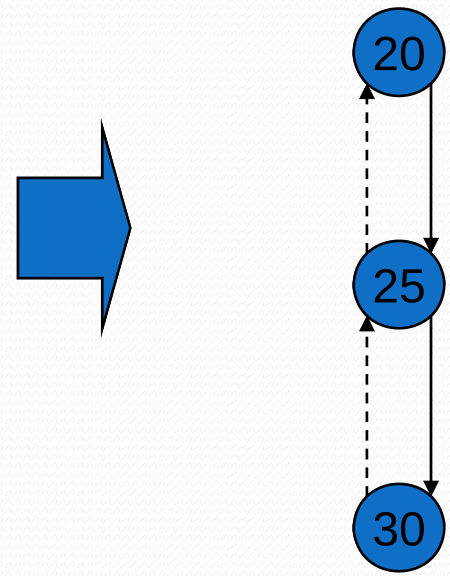
Maintaining successor pointers

- 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)

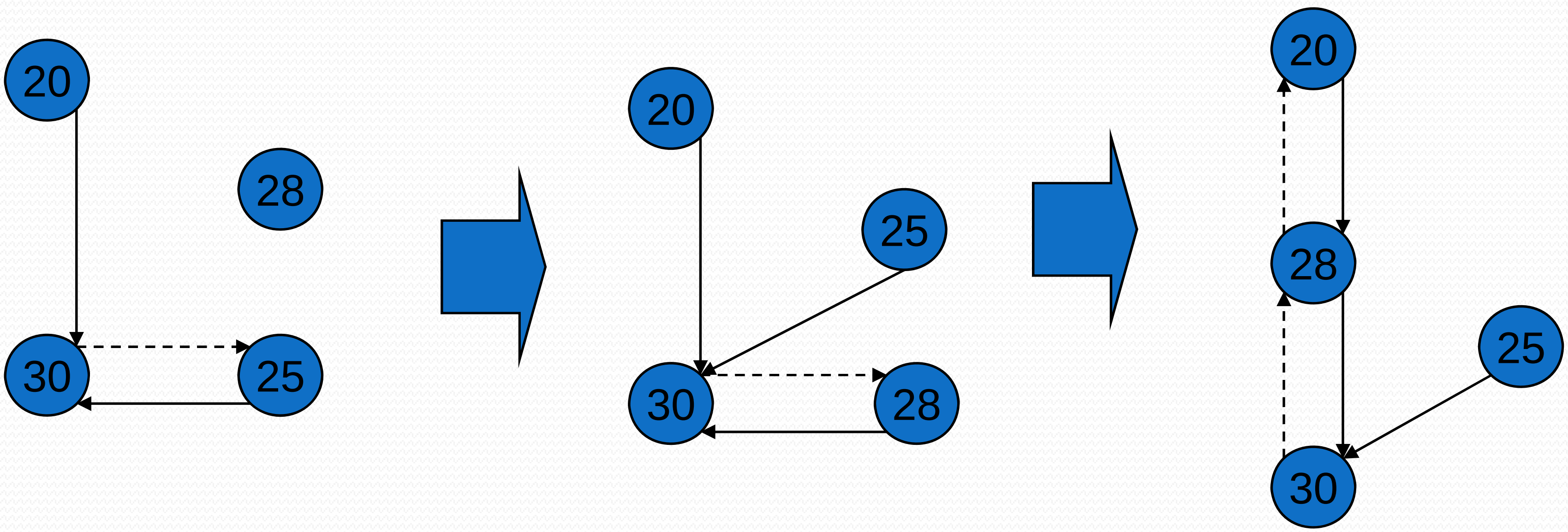


25 finds successor,
and tells successor
(30) of itself



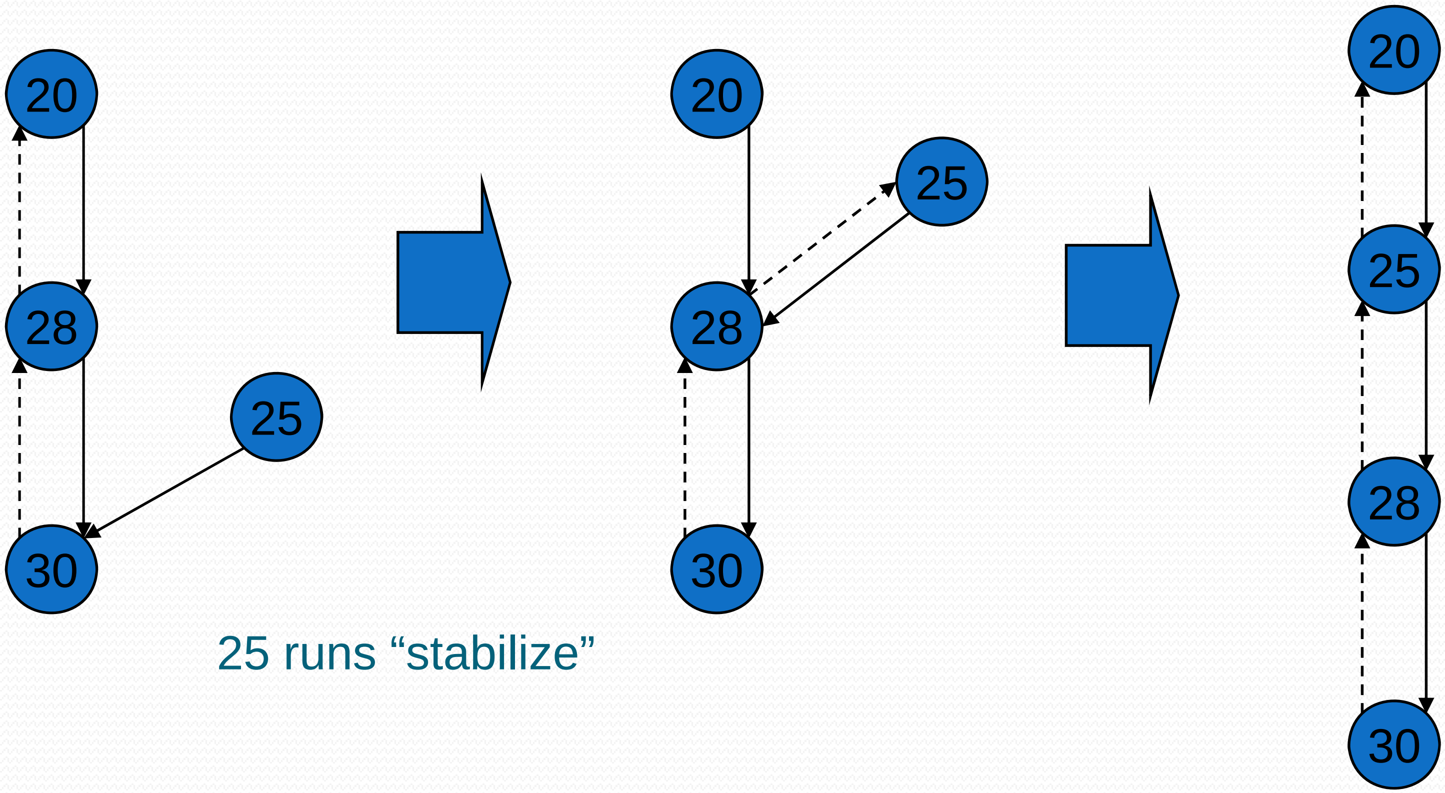
20 runs “stabilize”:
20 asks 30 for 30’s predecessor
30 returns 25
20 tells 25 of itself

This time, 28 joins before 20 runs “stabilize”



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

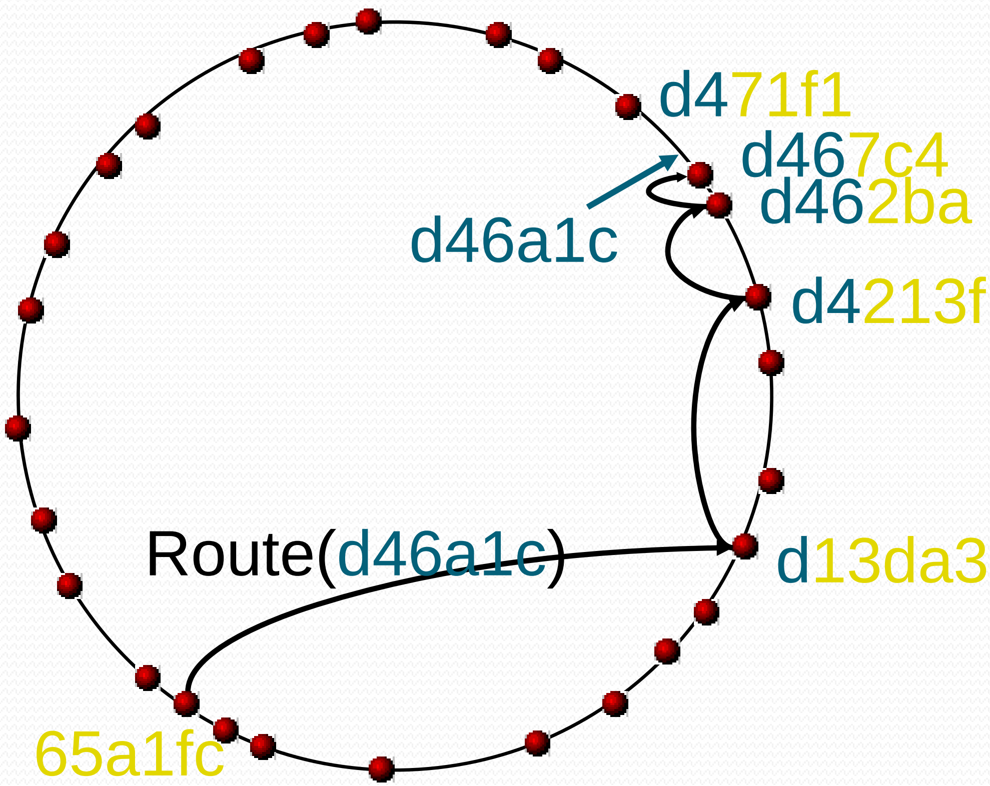
20 runs “stabilize”:
20 asks 30 for 30’s predecessor
30 returns 28
20 tells 28 of itself



25 runs “stabilize”

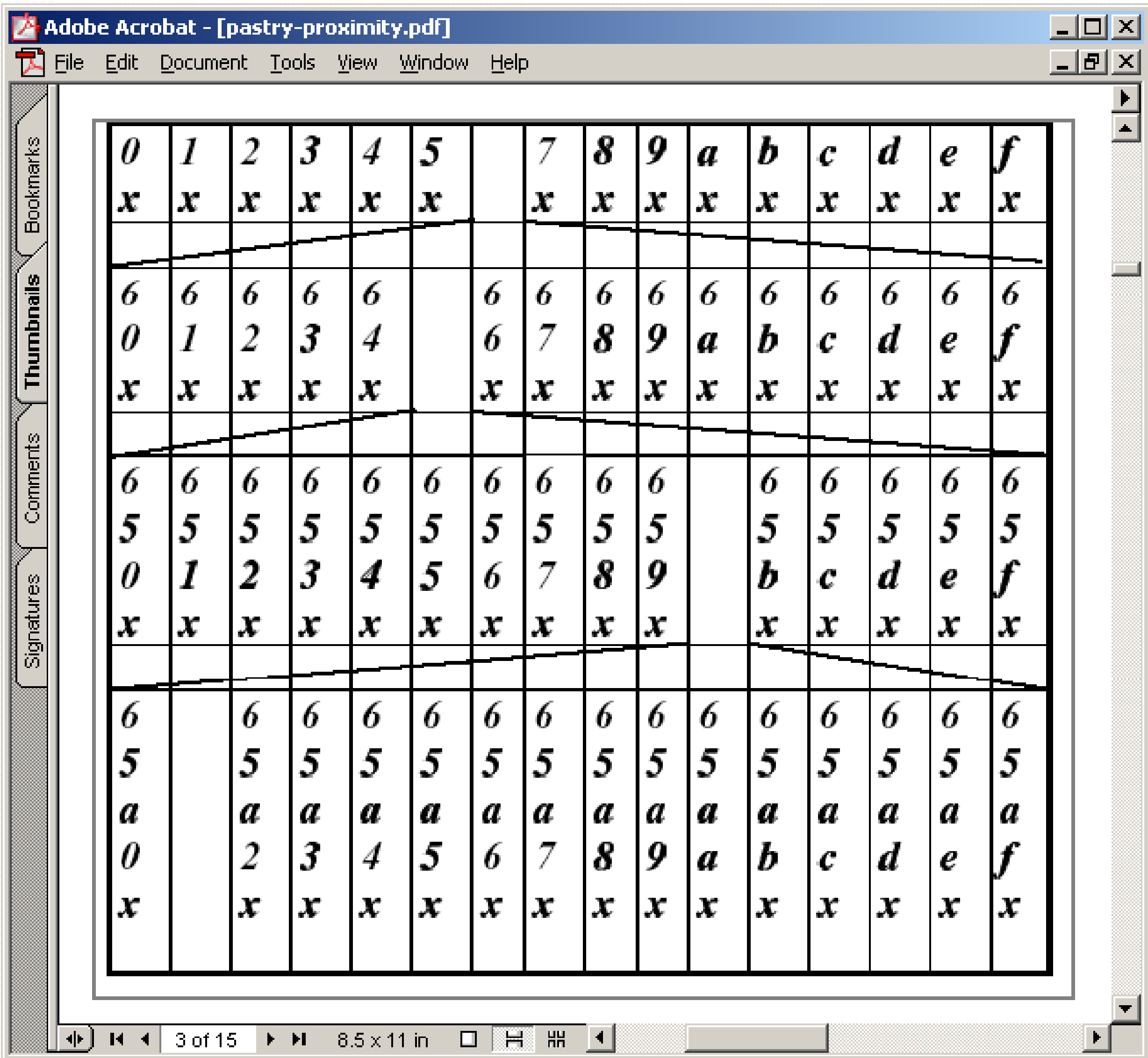
20 runs “stabilize”

Pastry also uses a circular number space



- Difference is in how the “fingers” are created
- Pastry uses prefix match overlap rather than binary splitting
- More flexibility in neighbor selection

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

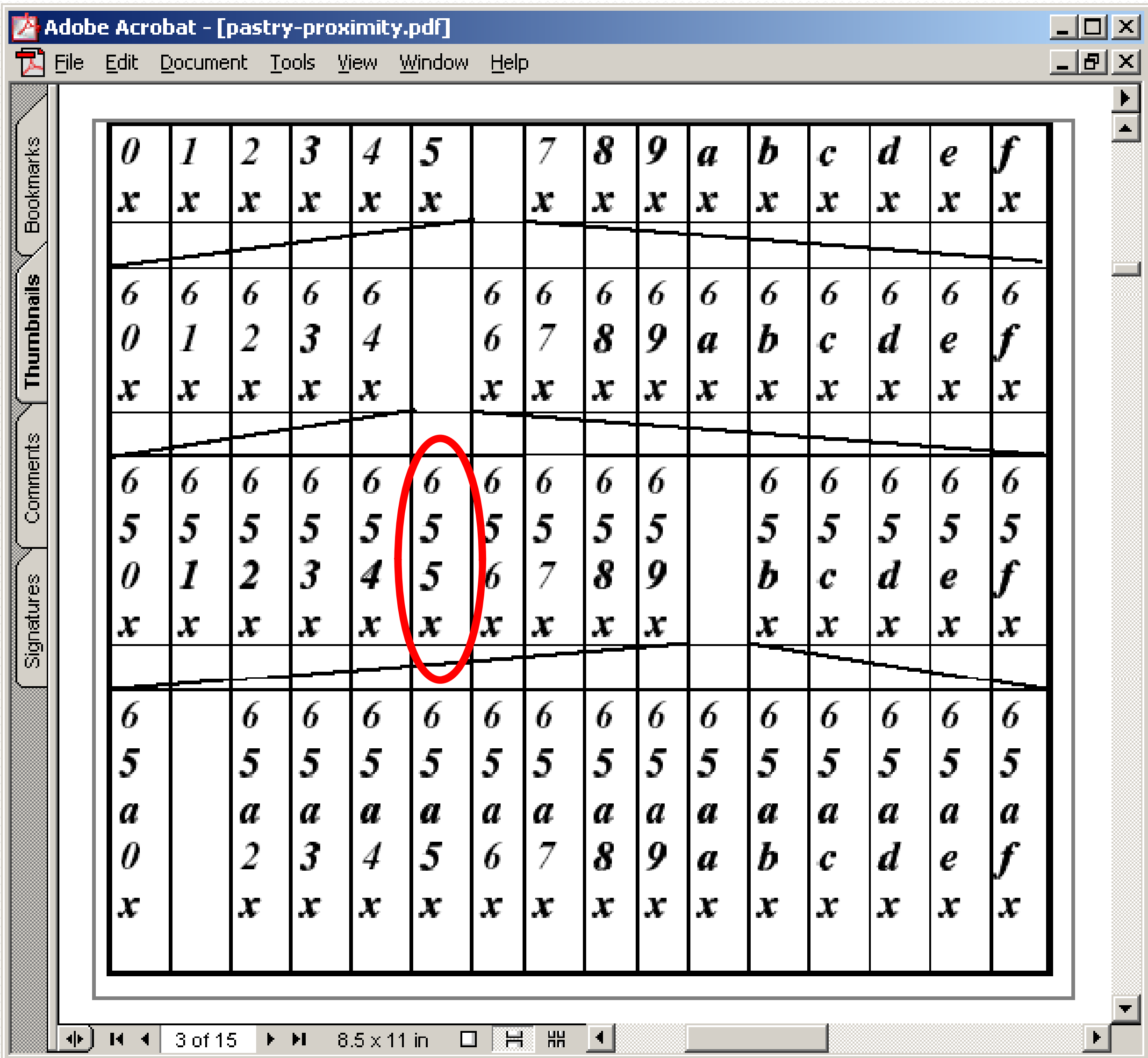
Pastry join

- 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

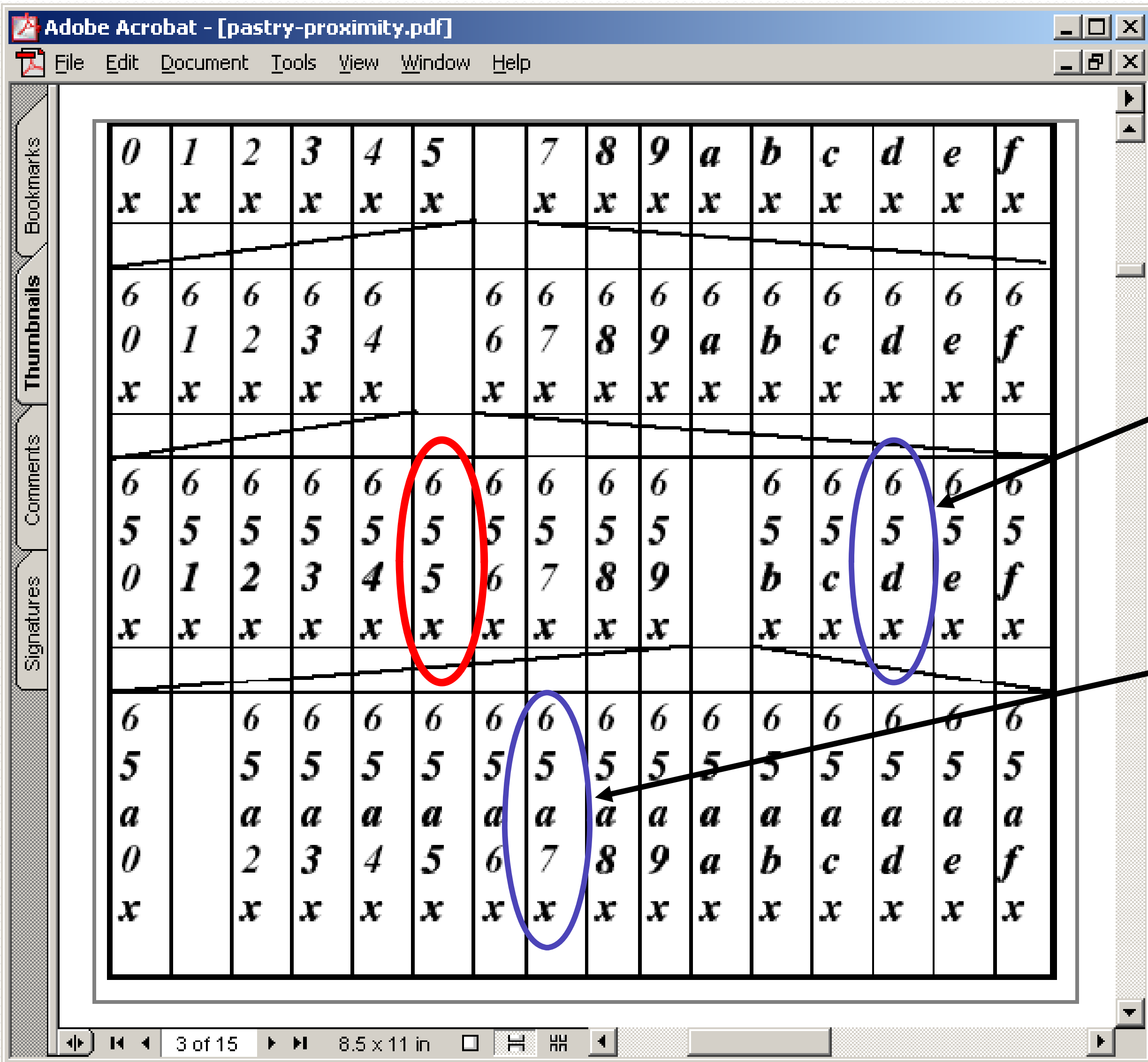
Pastry leave

- 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
  - 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
    - Harder for an attacker to manipulate system?

Security issues

- In many P2P systems, members may be malicious
- If peers untrusted, all content must be signed to detect forged content
  - Requires certificate authority
  - Like we discussed in secure web services talk
  - This is not hard, so can assume at least this level of security

Security issues: Sybil attack

- Attacker pretends to be multiple system
  - If surrounds a node on the circle, can potentially arrange to capture all traffic
  - Or if not this, at least cause a lot of trouble by being many nodes
- Chord requires node ID to be an SHA-1 hash of its IP address
  - But to deal with load balance issues, Chord variant allows nodes to replicate themselves
- *A central authority must hand out node IDs and certificates to go with them*
  - Not P2P in the Gnutella sense

General security rules

- Check things that can be checked
  - Invariants, such as successor list in Chord
- Minimize invariants, maximize randomness
  - Hard for an attacker to exploit randomness
- Avoid any single dependencies
  - Allow multiple paths through the network
  - Allow content to be placed at multiple nodes
- But all this is expensive…

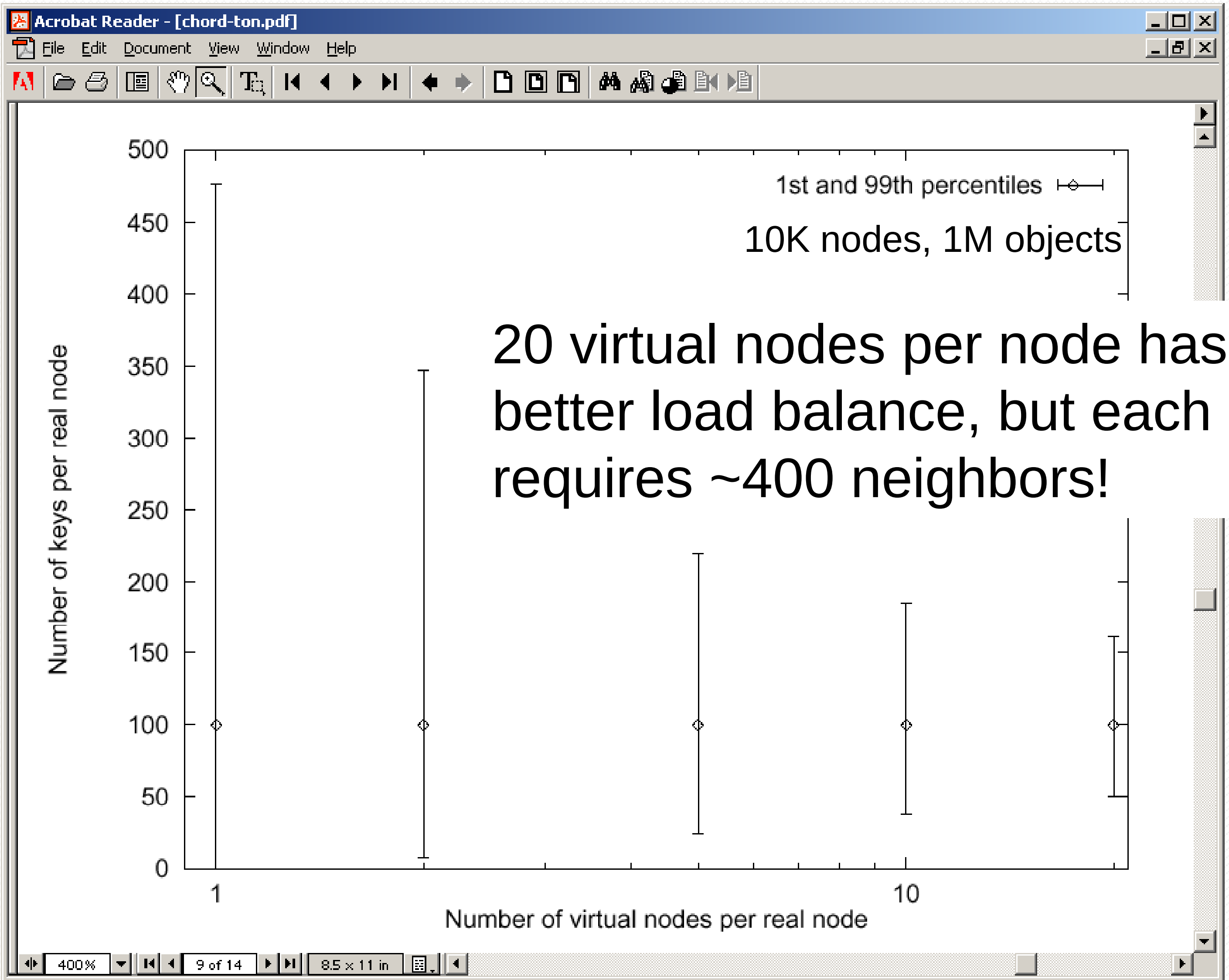
Load balancing

- Query hotspots: given object is popular
  - Cache at neighbors of hotspot, neighbors of neighbors, etc.
  - Classic caching issues
- Routing hotspot: node is on many paths
  - Of the three, Pastry seems most likely to have this problem, because neighbor selection more flexible (and based on proximity)
  - This doesn't seem adequately studied

Load balancing

- Heterogeneity (variance in bandwidth or node capacity
- Poor distribution in entries due to hash function inaccuracies
- One class of solution is to allow each node to be multiple virtual nodes
  - Higher capacity nodes virtualize more often
  - But security makes this harder to do

Chord node virtualization



20 virtual nodes per node has much better load balance, but each node requires ~400 neighbors!

Primary concern: churn

- Churn: nodes joining and leaving frequently
- Join or leave requires a change in some number of links
- Those changes depend on correct routing tables in other nodes
  - Cost of a change is higher if routing tables not correct
  - In chord, ~6% of lookups fail if three failures per stabilization
- But as more changes occur, probability of incorrect routing tables increases

Control traffic load generated by churn

- Chord and Pastry appear to deal with churn differently
- Chord join involves some immediate work, but repair is done periodically
  - Extra load only due to join messages
- Pastry join and leave involves immediate repair of all effected nodes' tables
  - Routing tables repaired more quickly, but cost of each join/leave goes up with frequency of joins/leaves
  - Scales quadratically with number of changes???
  - Can result in network meltdown???

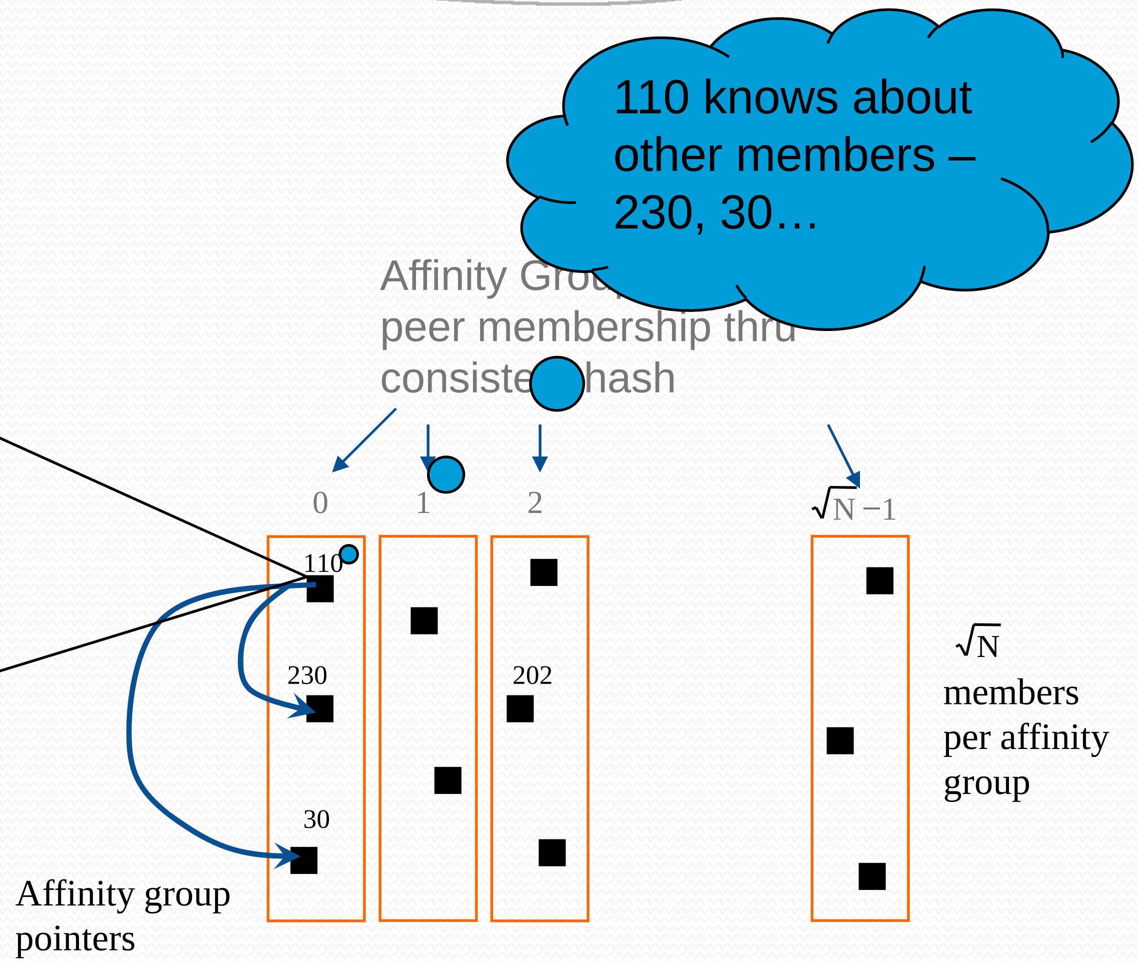
Kelips takes a different approach

- 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

Affinity group view

id	hbeat	rtt
30	234	90ms
230	322	30ms



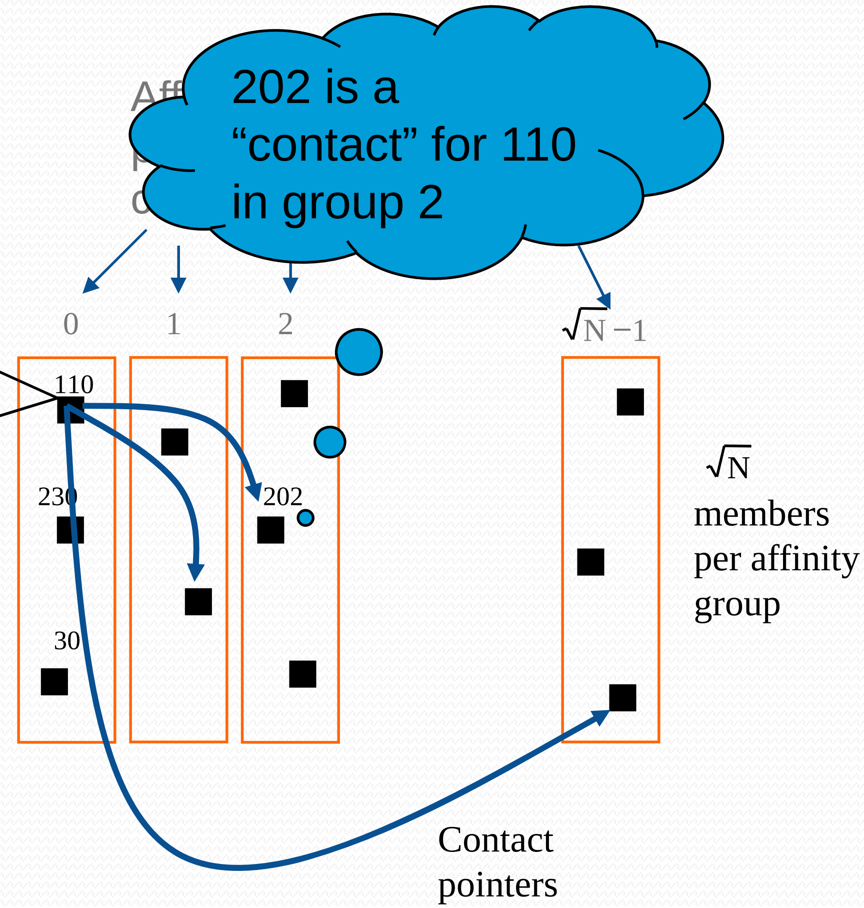
Kelips

Affinity group view

id	hbeat	rtt
30	234	90ms
230	322	30ms

Contacts

group	contactNode
…	…
2	202



Kelips

“cnn.com” maps to group 2. So 110 tells group 2 to “route” inquiries about cnn.com to it.

Affinity group view

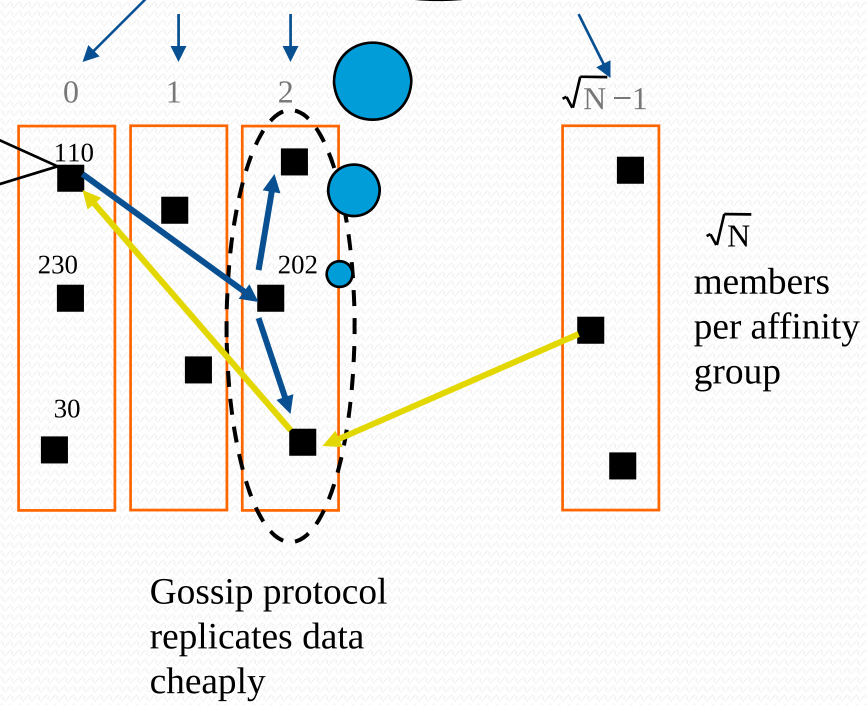
id	hbeat	rtt
30	234	90ms
230	322	30ms

Contacts

group	contactNode
…	…
2	202

Resource Tuples

resource	info
…	…
cnn.com	110



How it works

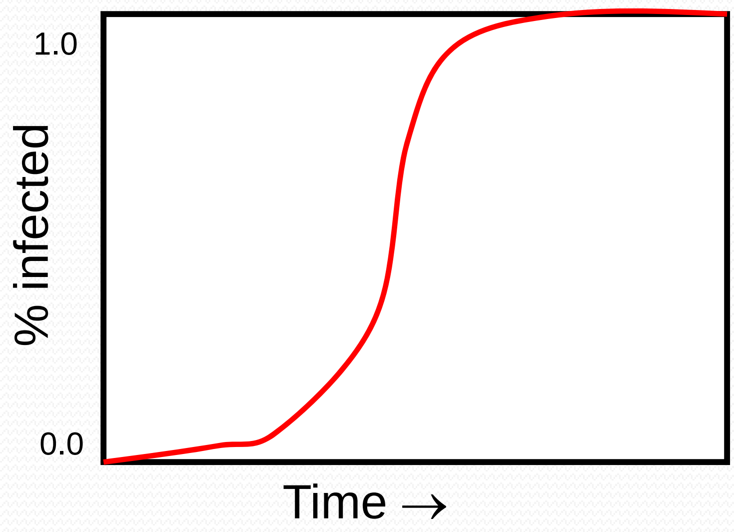
- Kelips is *entirely* gossip based!
  - Gossip about membership
  - Gossip to replicate and repair data
  - Gossip about "last heard from" time used to discard failed nodes
- Gossip "channel" uses fixed bandwidth
  - … fixed rate, packets of limited size

Gossip 101

- Suppose that I know something
- I'm sitting next to Fred, and I tell him
  - Now 2 of us "know"
- Later, he tells Mimi and I tell Anne
  - Now 4
- This is an example of a *push* epidemic
- *Push-pull* occurs if we exchange data

Gossip scales very nicely

- Participants' loads independent of size
- Network load linear in system size
- Information spreads in log(system size) time



Gossip in distributed systems

- We can gossip about membership
  - Need a bootstrap mechanism, but then discuss failures, new members
- Gossip to repair faults in replicated data
  - "I have 6 updates from Charlie"
- If we aren't in a hurry, gossip to replicate data too

Gossip about membership

- Start with a *bootstrap protocol*
  - For example, processes go to some web site and it lists a dozen nodes where the system has been stable for a long time
  - Pick one at random
- Then track "processes I've heard from recently" and "processes other people have heard from recently"
- Use push gossip to spread the word

Gossip about membership

- Until messages get full, everyone will known when everyone else last sent a message
  - With delay of log(N) gossip rounds…
- But messages will have bounded size
  - Perhaps 8K bytes
  - Then use some form of “prioritization” to decide what to omit – but *never send more, or larger messages*
  - Thus: load has a fixed, constant upper bound except on the network itself, which usually has infinite capacity

Back to Kelips: Quick reminder

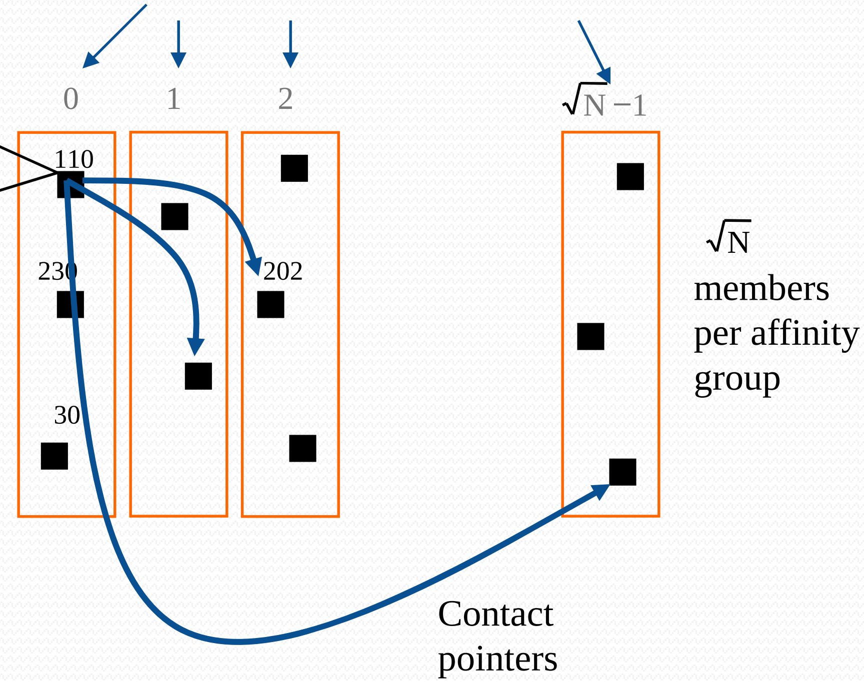
Affinity group view

id	hbeat	rtt
30	234	90ms
230	322	30ms

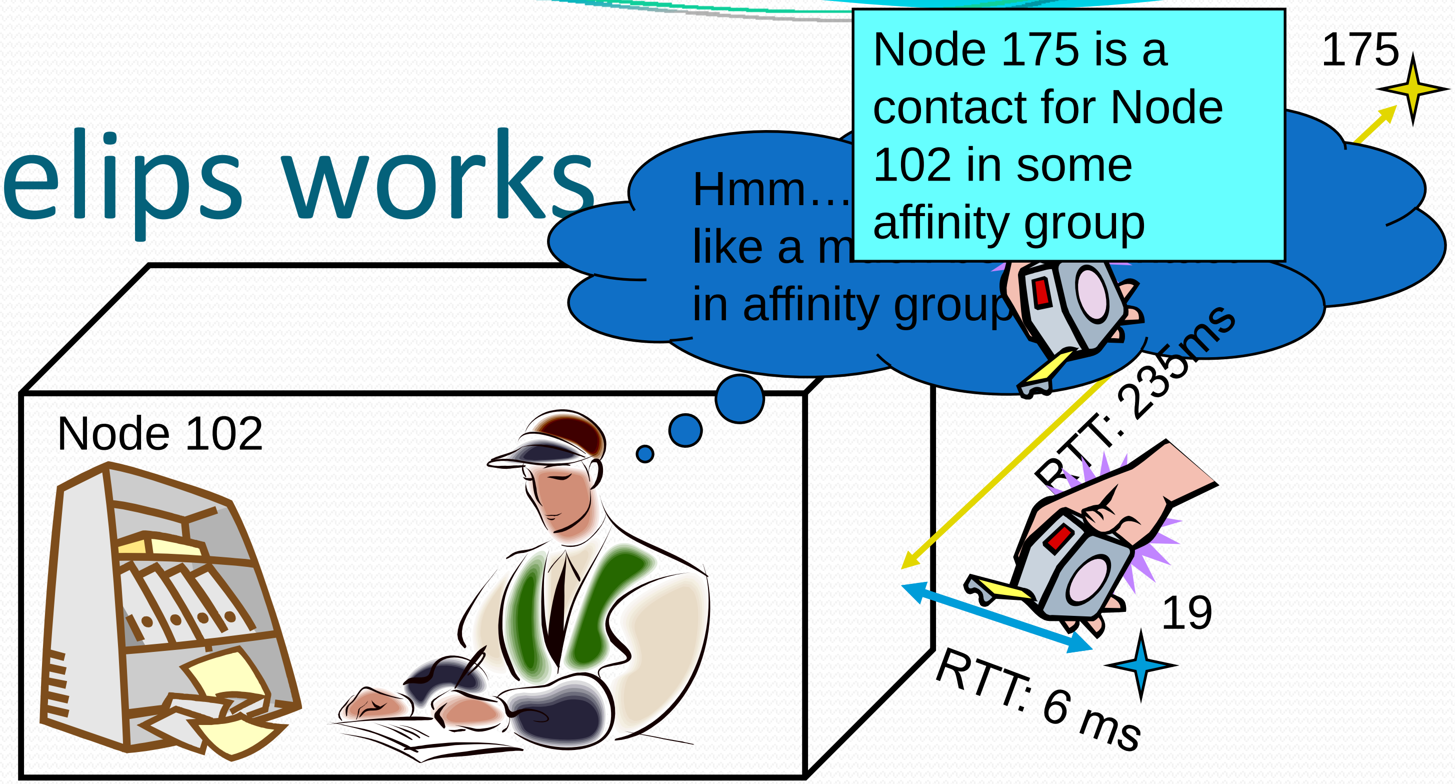
Contacts

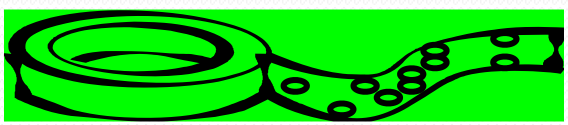
group	contactNode
…	…
2	202

Affinity Groups:
peer membership thru
consistent hash



How Kelips works





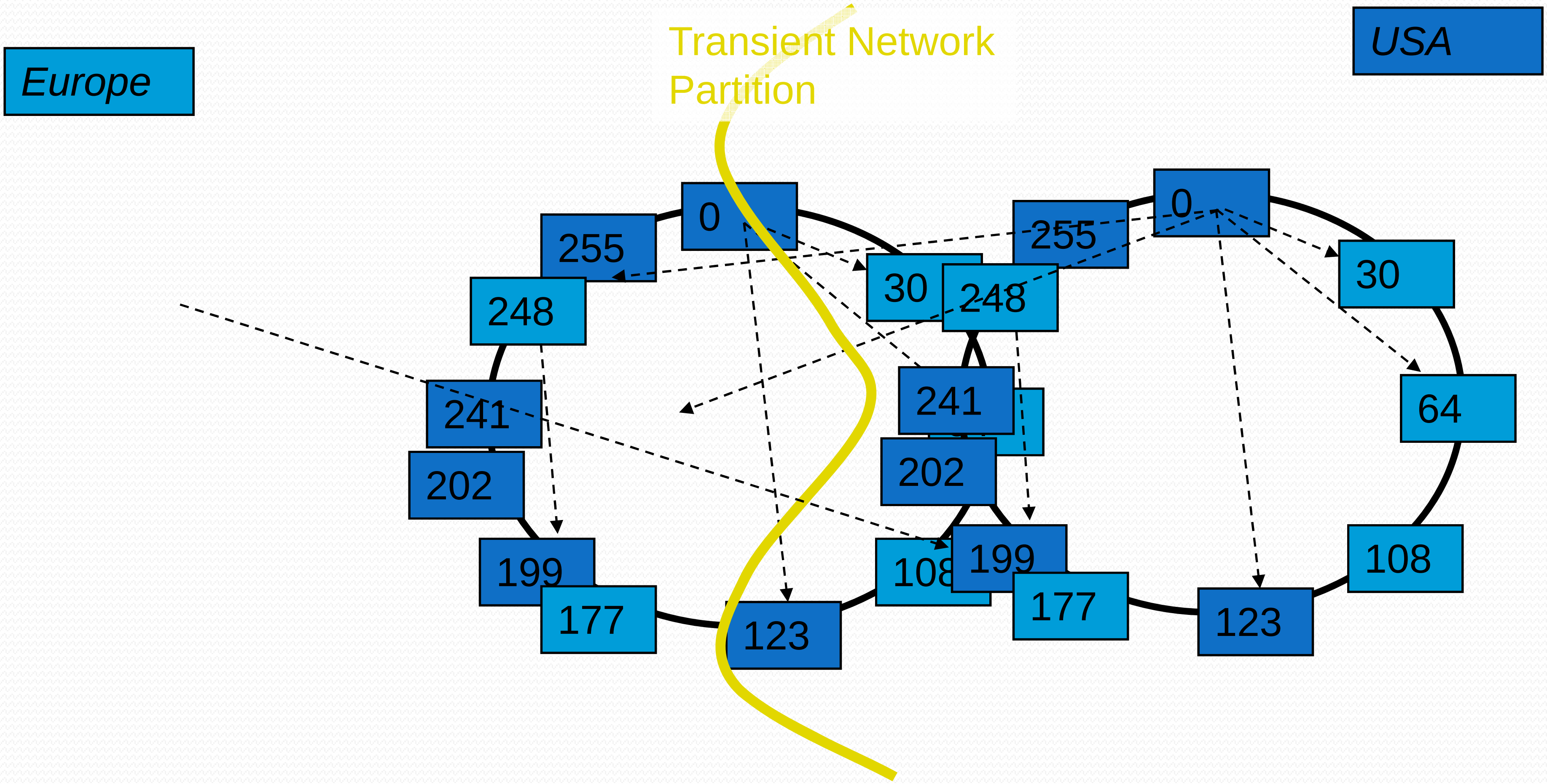
Gossip data stream

- Gossip about everything
- Heuristic to pick *contacts*: periodically ping contacts to check liveness, RTT… swap so-so ones for better ones.

Replication makes it robust

- Kelips should work even during disruptive episodes
  - After all, tuples are replicated to $\sqrt{N}$ nodes
  - Query k nodes concurrently to overcome isolated crashes, also reduces risk that very recent data could be missed
- … we often overlook importance of showing that systems work while recovering from a disruption

Chord can malfunction if the network partitions…

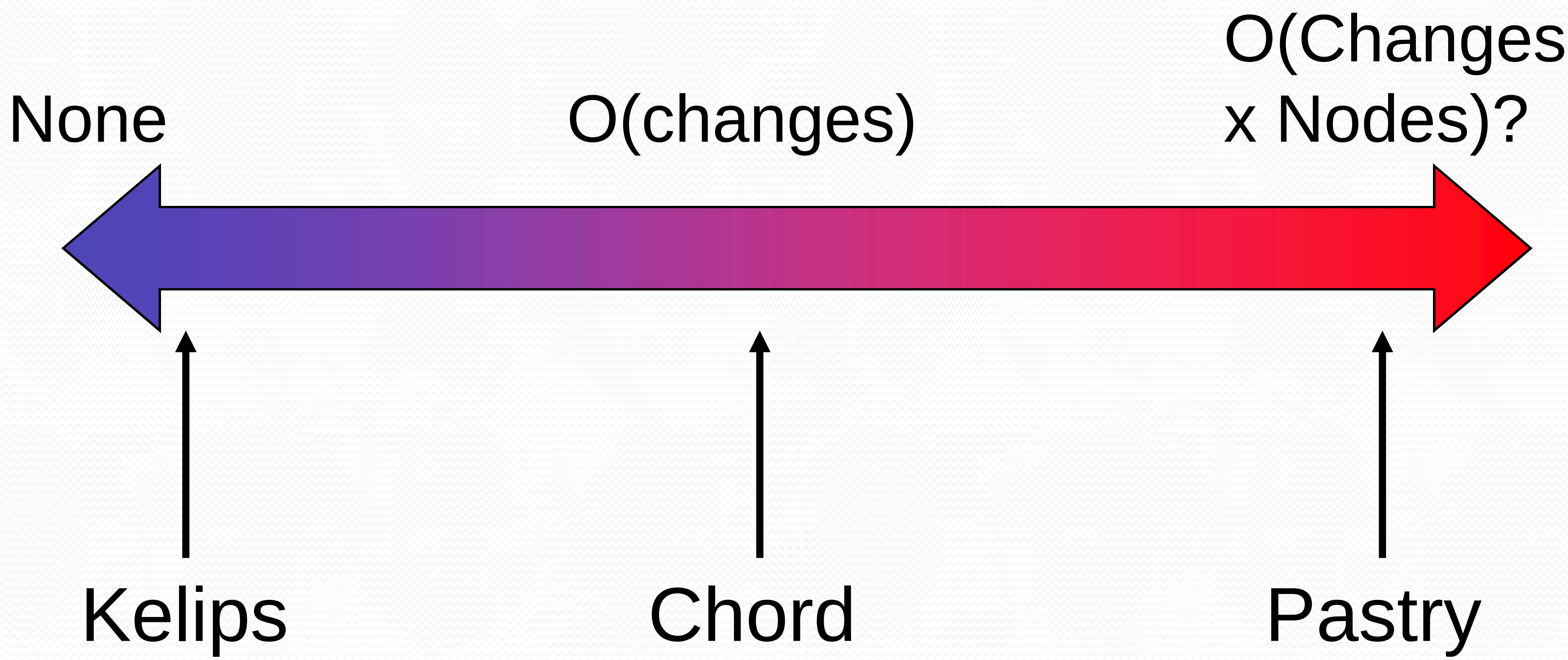


… so, who cares?

- Chord lookups can fail… and it suf high overheads when nodes churn
  - Loads surge just when thi eady disrupted… quite often, because of
  - And can't predict h g Chord might remain disrupted onc at way
- Worst case s *: Chord can become inconsistent and stay y*

> **The Fine Print**
>
> The scenario you have been shown is of low probability. In all likelihood, Chord would repair itself after any partitioning failure that might really arise. Caveat emptor and all that.

Control traffic load generated by churn



Take-Aways?

- Surprisingly easy to superimpose a hash-table lookup onto a potentially huge distributed system!
  - We've seen three O(log N) solutions and one O(1) solution (but Kelips needed more space)
- Sample applications?
  - Peer to peer file sharing
  - Amazon uses DHT for the shopping cart
  - CoDNS: A better version of DNS