

# CS5412: TIER 2 OVERLAYS

Lecture VI

Ken Birman

# Recap

2

- A week ago we discussed RON and Chord: typical examples of P2P network tools popular in the cloud
- Then we shifted attention and peeked into the data center itself. It has tiers (tier 1, 2, backend) and a wide range of technologies
- Many of those use a DHT “concept” and would be build on a DHT. But we can’t use Chord here!

# Today's focus

3

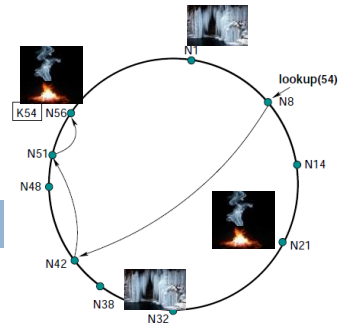
- How can we create distributed hash tables optimized for use in cloud computing settings?
- If you look deeply into systems like the ones we discussed last time, you'll find DHT technology at the base. So with a DHT you can layer fancier things on top... but the DHT determines the speed!

# First problem with Chord: Cost

4

- Internal to a cloud data center a DHT needs to be blindingly fast
  - ▣ Put operation should have cost no higher than 1 RPC directly to the nodes where the data will live
  - ▣ Get operation could have a cost of 1 RPC
- In Chord with as few as 1 000 participants, costs can include 9 routing hops. So this is unacceptable

# Another problem : Hot spots



5

- As conditions in a network change
  - ▣ Some items may become far more popular than others and be referenced often; others rarely: hot/cold spots
  - ▣ Members may join that are close to the place a finger pointer should point... but not exactly at the right spot
  - ▣ Churn could cause many of the pointers to point to nodes that are no longer in the network, or behind firewalls where they can't be reached
- This has stimulated work on “adaptive” overlays

# Today look at three examples

6

- Beehive: A way of extending Chord so that average delay for finding an item drops to a constant:  $O(1)$
- Pastry: A different way of designing the overlay so that nodes have a choice of where a finger pointer should point, enabling big speedups
- Kelips: A simple way of creating an  $O(1)$  overlay that trades extra memory for faster performance

# File systems on overlays

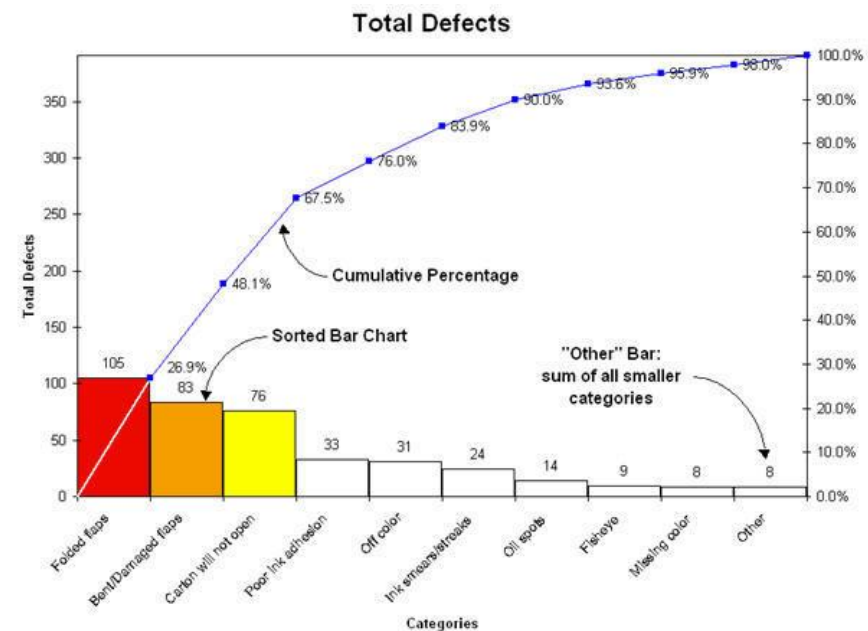
7

- If time permits, we'll also look at ways that overlays can “host” true file systems
- CFS and PAST: Two projects that used Chord and Pastry, respectively, to store blocks
- OceanStore: An archival storage system for libraries and other long-term storage needs

# Insight into adaptation

8

- Many “things” in computer networks exhibit Pareto popularity distributions
- This one graphs frequency by category for problems with cardboard shipping cartons
- Notice that a small subset of issues account for most problems





# Beehive insight

9

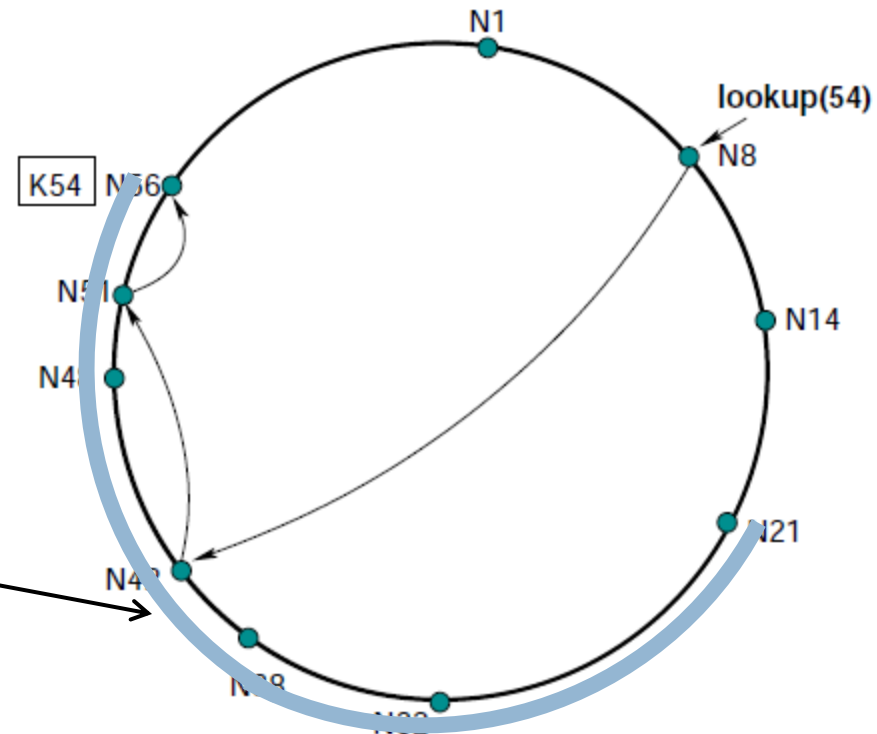
- Small subset of keys will get the majority of Put and Get operations
  - ▣ Intuition is simply that *everything* is Pareto!
- By replicating data, we can make the search path shorter for a Chord operation
- ... so by replicating in a way proportional to the popularity of an item, we can speed access to popular items!

# Beehive: Item replicated on $N/2$ nodes

10

- If an item isn't on "my side" of the Chord ring it must be on the "other side"

*In this example, by replicating a (key,value) tuple over half the ring, Beehive is able to guarantee that it will always be found in at most 1 hop. The system generalizes this idea, matching the level of replication to the popularity of the item.*



# Beehive strategy

11

- Replicate an item on  $N$  nodes to ensure  $O(0)$  lookup
- Replicate on  $N/2$  nodes to ensure  $O(1)$  lookup
- ...
- Replicate on just a single node (the “home” node) and worst case lookup will be the original  $O(\log n)$
- So use popularity of the item to select replication level

# Tracking popularity

12

- Each key has a home node (the one Chord would pick)
- Put (key,value) to the home node
- Get by finding any copy. Increment *access counter*
  - ▣ Periodically, aggregate the counters for a key at the home node, thus learning the access rate over time
  - ▣ A leader aggregates all access counters over all keys, then broadcasts the total access rate
    - ... enabling Beehive home nodes to learn relative rankings of items they host
    - ... and to compute the optimal replication factor for any target  $O(c)$  cost!

# Notice interplay of ideas here

13

- Beehive wouldn't work if every item was equally popular: we would need to replicate everything very aggressively. Pareto assumption addresses this
- Tradeoffs between parallel aspects (counting, creating replicas) and leader-driven aspects (aggregating counts, computing replication factors)
- We'll see ideas like these in many systems throughout CS541 2

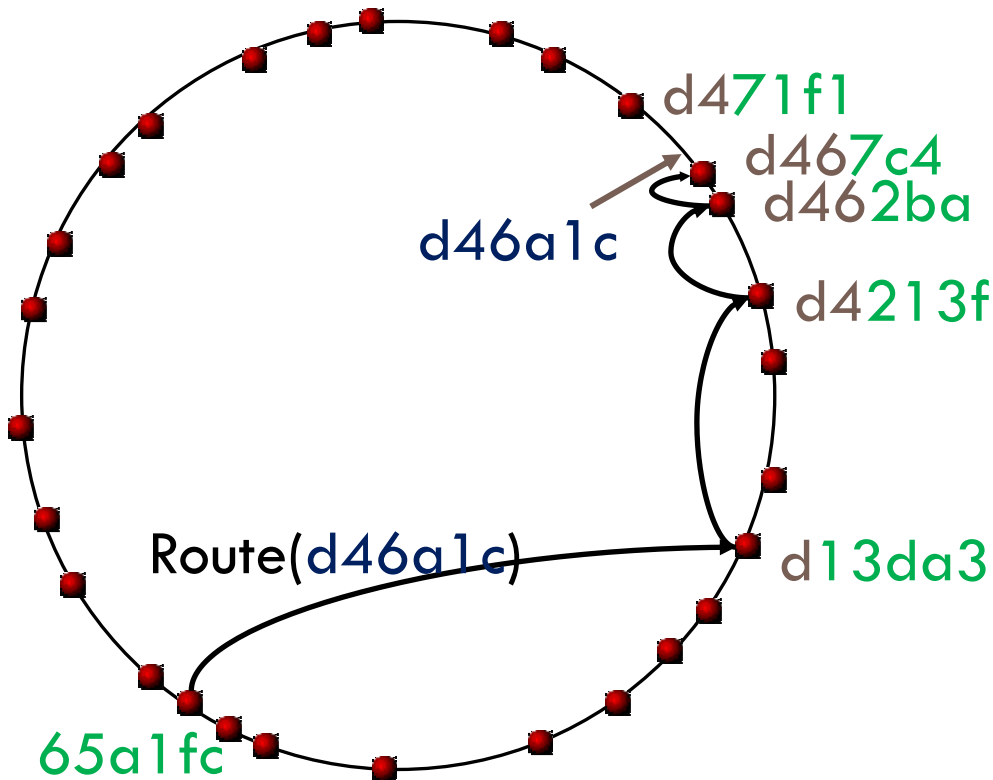
# Pastry

14

- A DHT much like Chord or Beehive
- But the goal here is to have more flexibility in picking finger links
  - ▣ In Chord, the node with hashed key  $H$  must look for the nodes with keys  $H/2$ ,  $H/4$ , etc....
  - ▣ In Pastry, there are a *set* of possible target nodes and this allows Pastry flexibility to pick one with good network connectivity, RTT (latency), load, etc

# Pastry also uses a circular number space

15



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

# Pastry routing table (for node 65a1fc)

16

<i>0</i>	<i>1</i>	<i>2</i>	<i>3</i>	<i>4</i>	<i>5</i>		<i>7</i>	<i>8</i>	<i>9</i>	<i>a</i>	<i>b</i>	<i>c</i>	<i>d</i>	<i>e</i>	<i>f</i>
<i>x</i>	<i>x</i>	<i>x</i>	<i>x</i>	<i>x</i>	<i>x</i>		<i>x</i>	<i>x</i>	<i>x</i>	<i>x</i>	<i>x</i>	<i>x</i>	<i>x</i>	<i>x</i>	<i>x</i>
<i>6</i>	<i>6</i>	<i>6</i>	<i>6</i>	<i>6</i>		<i>6</i>	<i>6</i>	<i>6</i>	<i>6</i>	<i>6</i>	<i>6</i>	<i>6</i>	<i>6</i>	<i>6</i>	<i>6</i>
<i>0</i>	<i>1</i>	<i>2</i>	<i>3</i>	<i>4</i>		<i>6</i>	<i>7</i>	<i>8</i>	<i>9</i>	<i>a</i>	<i>b</i>	<i>c</i>	<i>d</i>	<i>e</i>	<i>f</i>
<i>x</i>	<i>x</i>	<i>x</i>	<i>x</i>	<i>x</i>		<i>x</i>	<i>x</i>	<i>x</i>	<i>x</i>	<i>x</i>	<i>x</i>	<i>x</i>	<i>x</i>	<i>x</i>	<i>x</i>
<i>6</i>	<i>6</i>	<i>6</i>	<i>6</i>	<i>6</i>	<i>6</i>	<i>6</i>	<i>6</i>	<i>6</i>	<i>6</i>		<i>6</i>	<i>6</i>	<i>6</i>	<i>6</i>	<i>6</i>
<i>5</i>	<i>5</i>	<i>5</i>	<i>5</i>	<i>5</i>	<i>5</i>	<i>5</i>	<i>5</i>	<i>5</i>	<i>5</i>		<i>5</i>	<i>5</i>	<i>5</i>	<i>5</i>	<i>5</i>
<i>0</i>	<i>1</i>	<i>2</i>	<i>3</i>	<i>4</i>	<i>5</i>	<i>6</i>	<i>7</i>	<i>8</i>	<i>9</i>		<i>b</i>	<i>c</i>	<i>d</i>	<i>e</i>	<i>f</i>
<i>x</i>	<i>x</i>	<i>x</i>	<i>x</i>	<i>x</i>	<i>x</i>	<i>x</i>	<i>x</i>	<i>x</i>	<i>x</i>		<i>x</i>	<i>x</i>	<i>x</i>	<i>x</i>	<i>x</i>
<i>6</i>		<i>6</i>	<i>6</i>	<i>6</i>	<i>6</i>	<i>6</i>	<i>6</i>	<i>6</i>	<i>6</i>	<i>6</i>	<i>6</i>	<i>6</i>	<i>6</i>	<i>6</i>	<i>6</i>
<i>5</i>		<i>5</i>	<i>5</i>	<i>5</i>	<i>5</i>	<i>5</i>	<i>5</i>	<i>5</i>	<i>5</i>	<i>5</i>	<i>5</i>	<i>5</i>	<i>5</i>	<i>5</i>	<i>5</i>
<i>a</i>		<i>a</i>	<i>a</i>	<i>a</i>	<i>a</i>	<i>a</i>	<i>a</i>	<i>a</i>	<i>a</i>	<i>a</i>	<i>a</i>	<i>a</i>	<i>a</i>	<i>a</i>	<i>a</i>
<i>0</i>		<i>2</i>	<i>3</i>	<i>4</i>	<i>5</i>	<i>6</i>	<i>7</i>	<i>8</i>	<i>9</i>	<i>a</i>	<i>b</i>	<i>c</i>	<i>d</i>	<i>e</i>	<i>f</i>
<i>x</i>		<i>x</i>	<i>x</i>	<i>x</i>	<i>x</i>	<i>x</i>	<i>x</i>	<i>x</i>	<i>x</i>	<i>x</i>	<i>x</i>	<i>x</i>	<i>x</i>	<i>x</i>	<i>x</i>

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

Similar to Chord’s list of successors



# Pastry join

17

- 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
  - ▣ 18<sup>th</sup> IFIP/ACM, Nov 2001

# Pastry leave

18

- 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

19

The screenshot shows the Adobe Acrobat interface with a document titled "[pastry-proximity.pdf]". The document content is a grid of characters and numbers. The grid is divided into several sections by horizontal lines. The top section has columns labeled 0-9, a, b, c, d, e, f. The second section has columns labeled 6, 0-5, 6-9, a-f. The third section has columns labeled 6, 5, 0-4, 5, 6-9, b-f. The fourth section has columns labeled 6, 5, a, 0-9, x. The fifth section has columns labeled 6, 5, a, 0-9, x. A red circle highlights the character '5' in the third row, fifth column of the grid.

0	1	2	3	4	5		7	8	9	a	b	c	d	e	f
x	x	x	x	x	x		x	x	x	x	x	x	x	x	x
6	6	6	6	6		6	6	6	6	6	6	6	6	6	6
0	1	2	3	4		6	7	8	9	a	b	c	d	e	f
x	x	x	x	x		x	x	x	x	x	x	x	x	x	x
6	6	6	6	6	6	6	6	6	6		6	6	6	6	6
5	5	5	5	5	5	5	5	5	5		5	5	5	5	5
0	1	2	3	4	5	6	7	8	9		b	c	d	e	f
x	x	x	x	x	x	x	x	x	x		x	x	x	x	x
6		6	6	6	6	6	6	6	6	6	6	6	6	6	6
5		5	5	5	5	5	5	5	5	5	5	5	5	5	5
a		a	a	a	a	a	a	a	a	a	a	a	a	a	a
0		2	3	4	5	6	7	8	9	a	b	c	d	e	f
x		x	x	x	x	x	x	x	x	x	x	x	x	x	x

CS5412 Spring 2014 (Cloud Computing: Birman)

# Ask other neighbors

20

The screenshot shows the Adobe Acrobat interface with a document titled "[pastry-proximity.pdf]". The document content is a grid of characters arranged in a 5x16 layout. The grid is divided into five horizontal sections by diagonal lines. The characters in the grid are as follows:

0	1	2	3	4	5		7	8	9	a	b	c	d	e	f
x	x	x	x	x	x		x	x	x	x	x	x	x	x	x
6	6	6	6	6		6	6	6	6	6	6	6	6	6	6
0	1	2	3	4		6	7	8	9	a	b	c	d	e	f
x	x	x	x	x		x	x	x	x	x	x	x	x	x	x
6	6	6	6	6	6	6	6	6	6		6	6	6	6	6
5	5	5	5	5	5	5	5	5	5		5	5	5	5	5
0	1	2	3	4	5	6	7	8	9		b	c	d	e	f
x	x	x	x	x	x	x	x	x	x		x	x	x	x	x
6		6	6	6	6	6	6	6	6	6	6	6	6	6	6
5		5	5	5	5	5	5	5	5	5	5	5	5	5	5
a		a	a	a	a	a	a	a	a	a	a	a	a	a	a
0		2	3	4	5	6	7	8	9	a	b	c	d	e	f
x		x	x	x	x	x	x	x	x	x	x	x	x	x	x

Annotations in the image include:

- A red oval around the character '5' in the third row, fifth column.
- A blue oval around the character '6' in the third row, thirteenth column.
- A blue oval around the character '6' in the fourth row, seventh column.

The Acrobat interface includes a menu bar (File, Edit, Document, Tools, View, Window, Help), a sidebar with panels (Bookmarks, Thumbnails, Comments, Signatures), and a status bar at the bottom showing "3 of 15" and "8.5 x 11 in".

Try asking some neighbor in the same row for its 65x entry

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

# CAN, Chord, Pastry differences

21

- 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
    - How might an attacker try to manipulate system?

# Security issues

22

- 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

23

- 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

24

- 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

25

- 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

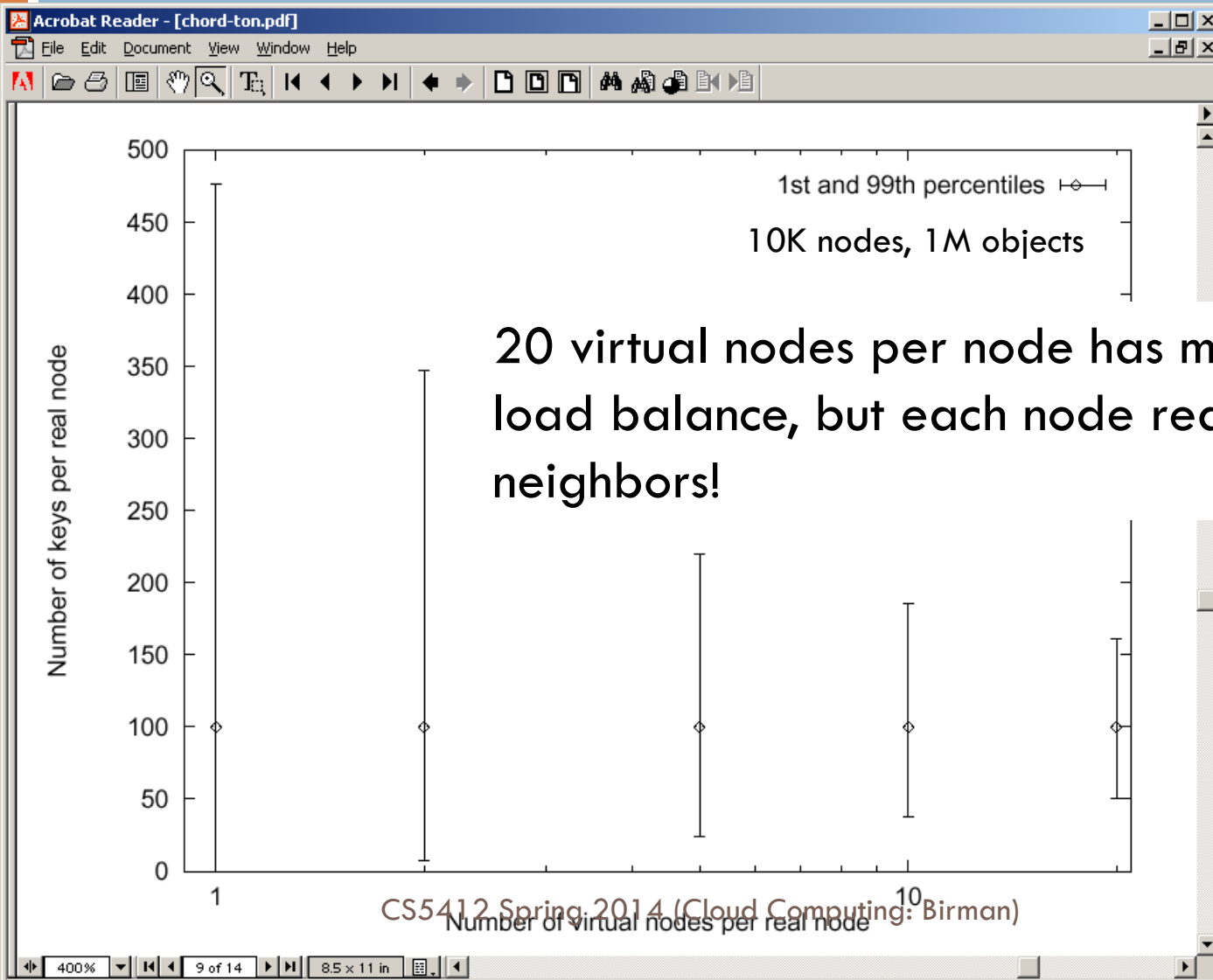
# Load balancing

26

- 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

27



# Fireflies

28

- Van Renesse uses this same trick (virtual nodes)
- In his version a form of attack-tolerant agreement is used so that the virtual nodes can repel many kinds of disruptive attacks
- We won't have time to look at the details today

# Another major concern: churn

29

- 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,  $\sim 6\%$  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

30

- 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

31

- Network partitioned into  $\sqrt{N}$  “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)!*

# Rationale?

32

- Kelips has a completely predictable behavior under worst-case conditions
  - ▣ It may do “better” but won’t do “worse”
  - ▣ Bounded message sizes and rates that never exceed what the administrator picks no matter how much churn occurs
  - ▣ Main impact of disruption: Kelips may need longer before Get is guaranteed to return value from prior Put with the same key



# Kelips

110 knows about other members – 230, 30...

Affinity group view

id	hbeat	rtt
30	234	90ms
230	322	30ms

Affinity group pointers

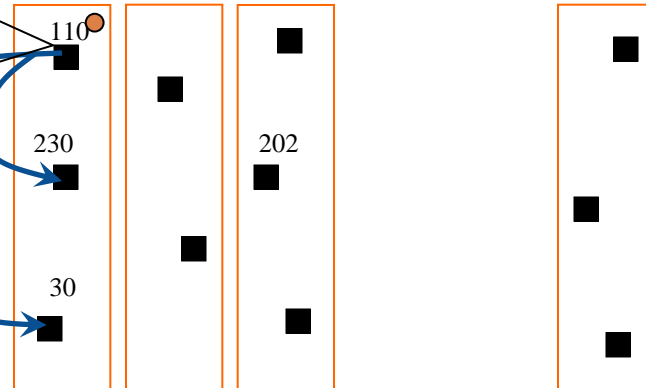
Affinity Group, peer membership thru consistent hash

0

1

2

$\sqrt{N}-1$



$\sqrt{N}$  members per affinity group

# Kelips

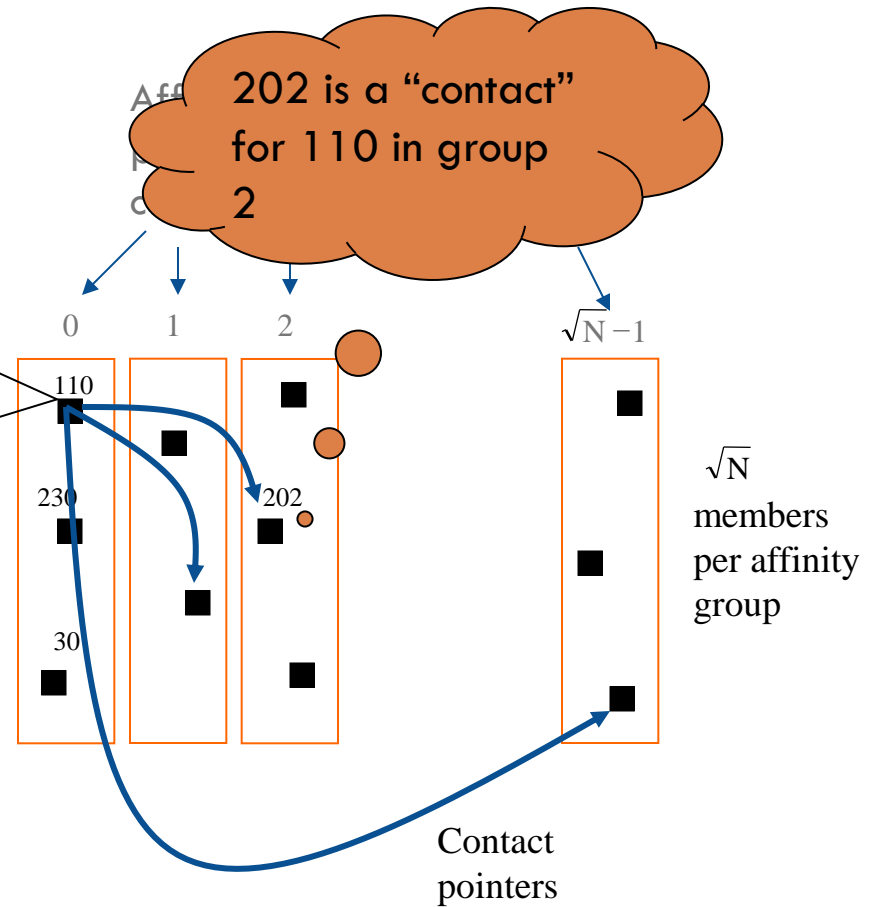
34

Affinity group view

id	hbeat	rtt
30	234	90ms
230	322	30ms

Contacts

group	contactNode
...	...
2	202



# Kelips

35

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

## Affinity group view

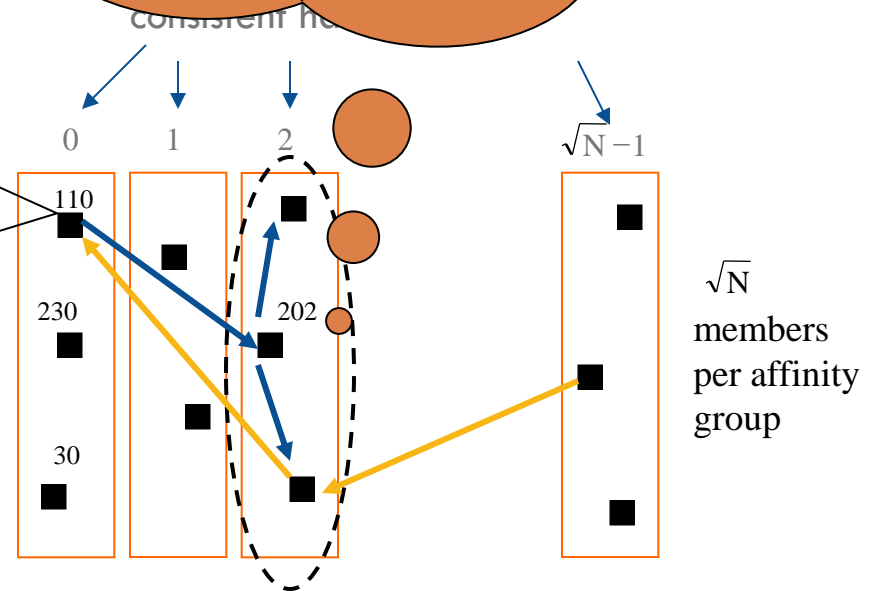
id	hbeat	rtt
30	234	90ms
230	322	30ms

## Contacts

group	contactNode
...	...
2	202

## Resource Tuples

resource	info
...	...
cnn.com	110



Gossip protocol replicates data cheaply

$\sqrt{N}$  members per affinity group

# How it works

36

- 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

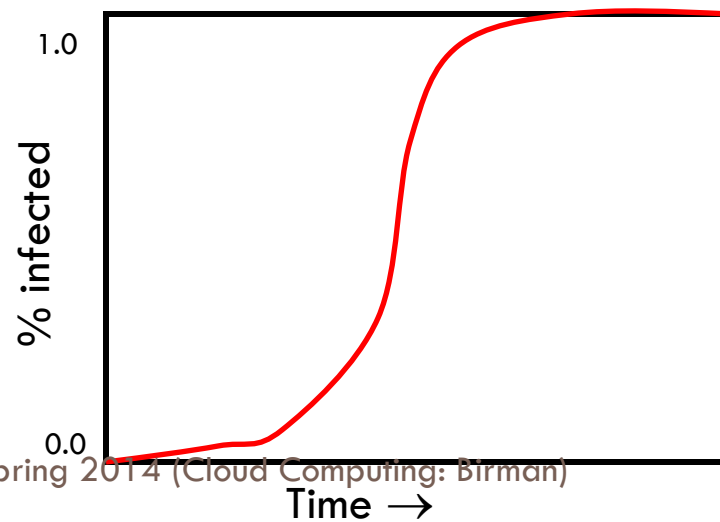
37

- 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

38

- Participants' loads independent of size
- Network load linear in system size
- Information spreads in  $\log(\text{system size})$  time



# Gossip in distributed systems

39

- 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

40

- 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

41

- Until messages get full, everyone will know 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

42

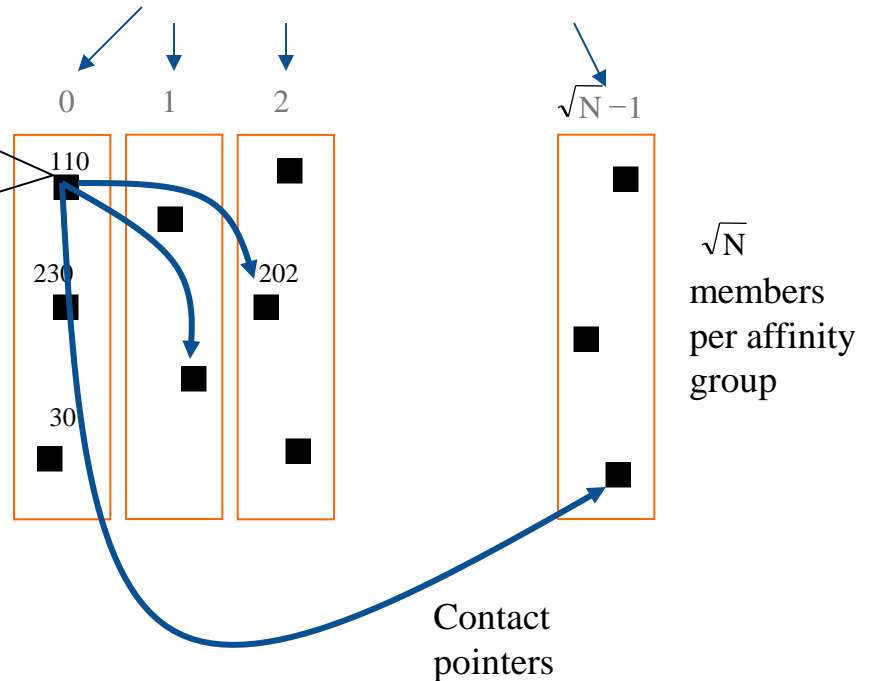
Affinity group view

id	hbeat	rtt
30	234	90ms
230	322	30ms

Contacts

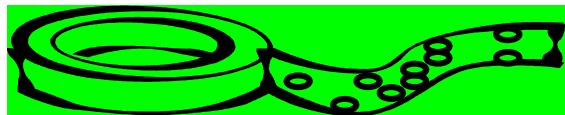
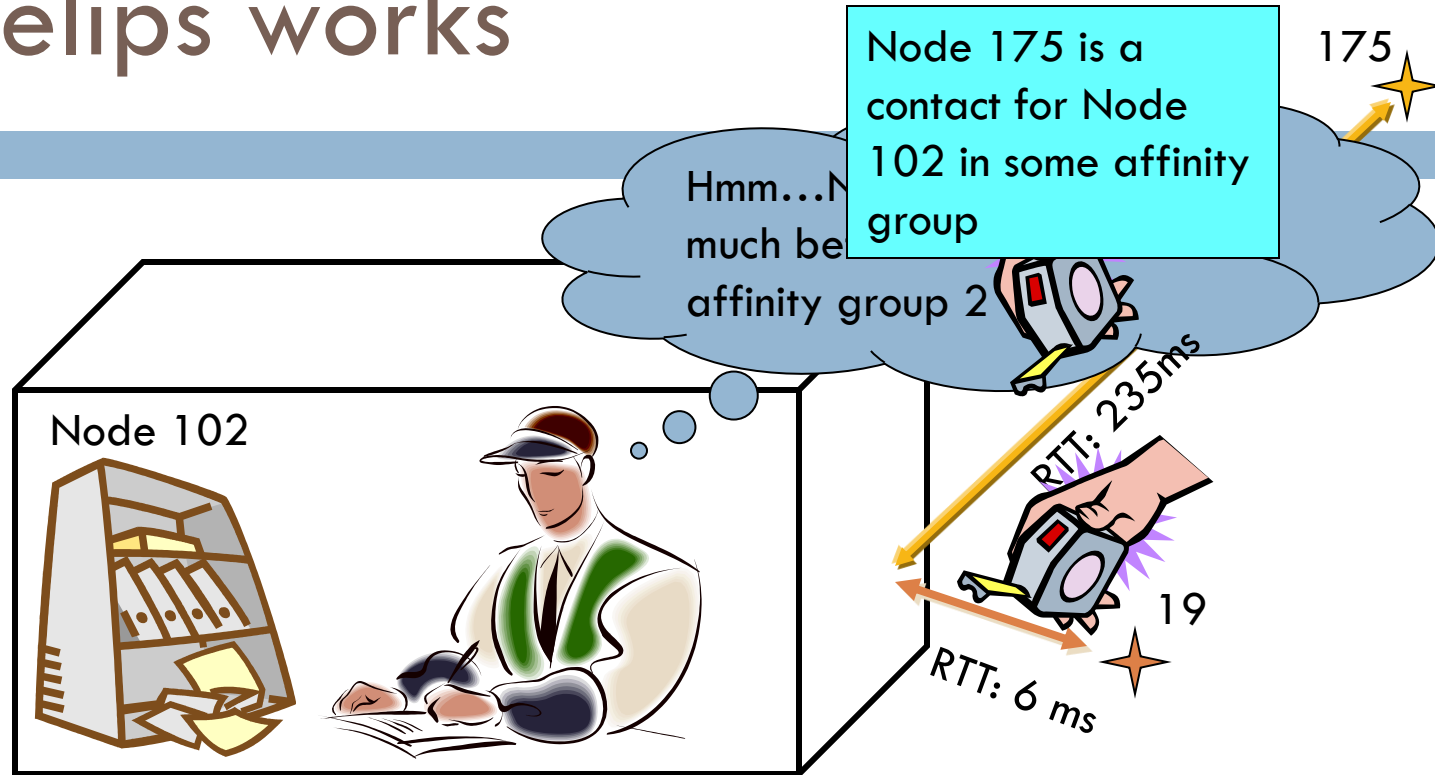
group	contactNode
...	...
2	202

Affinity Groups:  
peer membership thru  
consistent hash



# How Kelips works

43



Gossip data stream

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

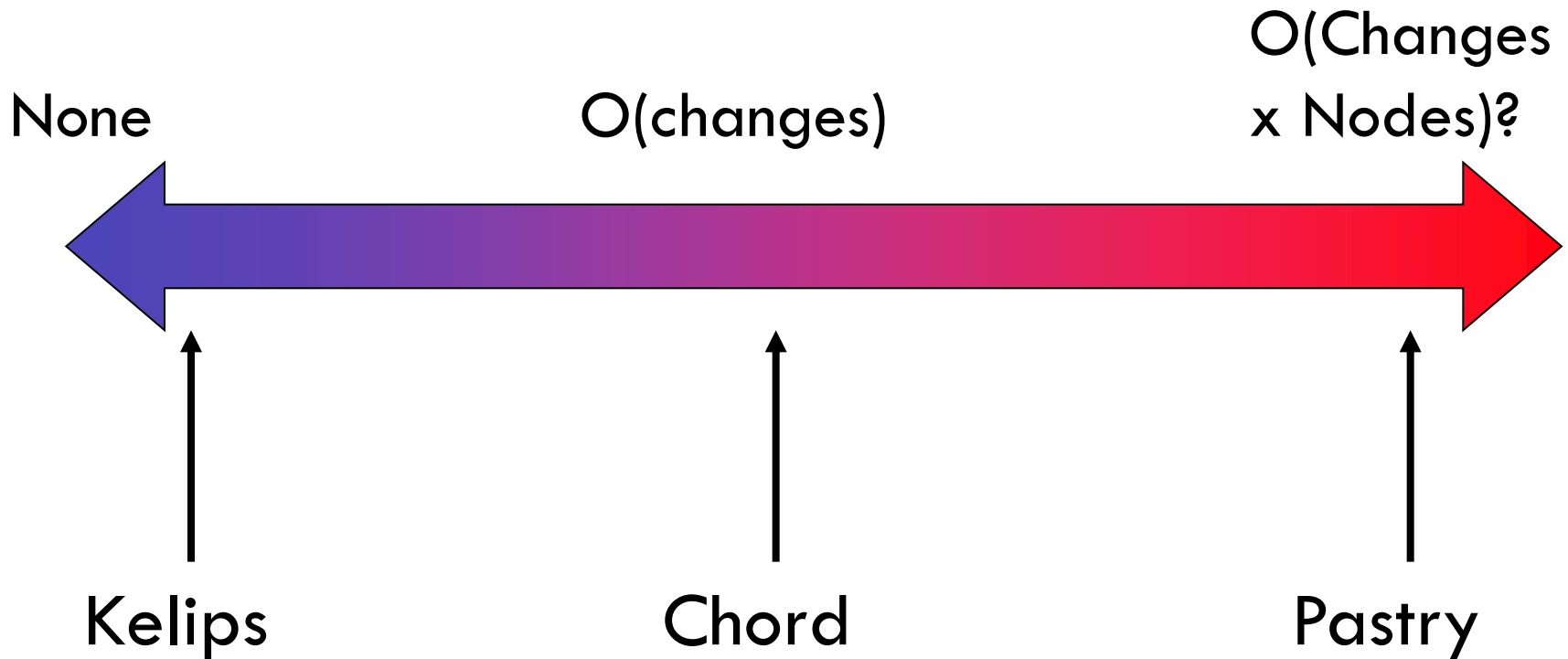
# Replication makes it robust

44

- 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

# Control traffic load generated by churn

45



# Summary

46

- Adaptive behaviors can improve overlays
  - ▣ Reduce costs for inserting or looking up information
  - ▣ Improve robustness to churn or serious disruption
- As we move from CAN to Chord to Beehive or Pastry one could argue that complexity increases
- Kelips gets to a similar place and yet is very simple, but pays a higher storage cost than Chord/Pastry