CS5412: TIER 2 OVERLAYS

Lecture VI

Ken Birman

Recap

- A week ago we discussed RON and Chord: typical examples of P2P network tools popular in the cloud

- They were invented purely as content indexing systems, but 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 datacenter technologies turn out to use a DHT "concept" and would be build on a DHT
  - But one important point arises: inside a data center the DHT can be optimized to take advantage of "known membership"

CS5412 DHT "Road map"

Lecture and Topic		Lecture and Topic
First the big picture (Tuesday 2/9: RON and Chord)		A DHT can support many things! (Thursday 2/11: BigTable, …)
Can a WAN DHT be equally flexible? (Thursday 2/17: Beehive, Pastry)		Another kind of overlay: BitTorrent (Tuesday 2/22)

Some key takeaways to deeply understand, likely to show up on tests.

1. The DHT get/put abstraction is simple, scalable, extremely powerful.
2. DHTs are very useful for finding content ("indexing") and for caching data
3. A DHT can also ***mimic*** other functionality but in such cases, keep in mind that DHT guarantees are weak: ***a DHT is not an SQL database.***
4. DHTs work best inside the datacenter, because accurate membership information is available. This allows us to completely avoid "indirect routing" and just talk directly to the DHT member(s) we need to.
5. In a WAN we can approximate this kind of membership tracking, but less accurately. This limits WAN DHTs. Thus we can't use WAN DHTs with the same degree of confidence as we do inside a datacenter.

DHT in a WAN setting

- Some cloud companies are focused on Internet of Things scenarios that need DHTs in a WAN.
- Content hosting companies like Akamai often have a decentralized pull infrastructure and use a WAN overlay to find content (to avoid asking for the same thing again and again from the origin servers)
- Puzzle: Nobody can tolerate log(N) delays

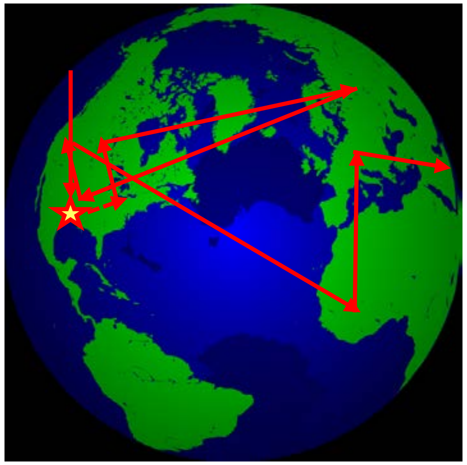
Native Chord wouldn't be fast enough

- Internal to a cloud data center a DHT can be pretty reliable and blindingly fast
  - Nodes don't crash often, and RPC is quite fast
  - Get/Put operation runs in 1RPC directly to the node(s) where the data is being held. Typical cost? 100us or less.
- But to get this speed every application needs access to a copy of the table of DHT member nodes
- In a WAN deployment of Chord with 1000 participants, we lack that table of members. With "Chord style routing", the get/put costs soar to perhaps 9 routing hops: maybe 1.5-2s. *<u>This overhead is unacceptable</u>*

Why was 1-hop 100us but 9 1.5s?

- Seems like 9 hops should be 900us?

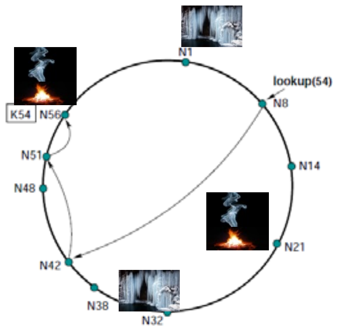
- Actually not: not all hops are the same!
  - Inside a data center, a hop really is directly over the network and only involves optical links and optical routers. So these are fast local hops.
  - In a WAN setting, each hop is over the Internet and for Chord, to a global destination! So each node-to-node hop could easily take 75-150ms. 9 such hops add up.



Churn

- We use this term when a WAN system has many nodes that come and go dynamically
- Common with mobile clients who try to have a WAN system right on their handheld devices, but move in and out of 3G network coverage areas
- Their nodes leave and join frequently, so Chord ends up with very inaccurate pointer tables

Hot spots



- In heavily used systems
  - 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

- 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

WAN structures on overlays

- We won’t have time to discuss how better overlays are used in the WAN, but CDN search in a system with a large number of servers is a common case.

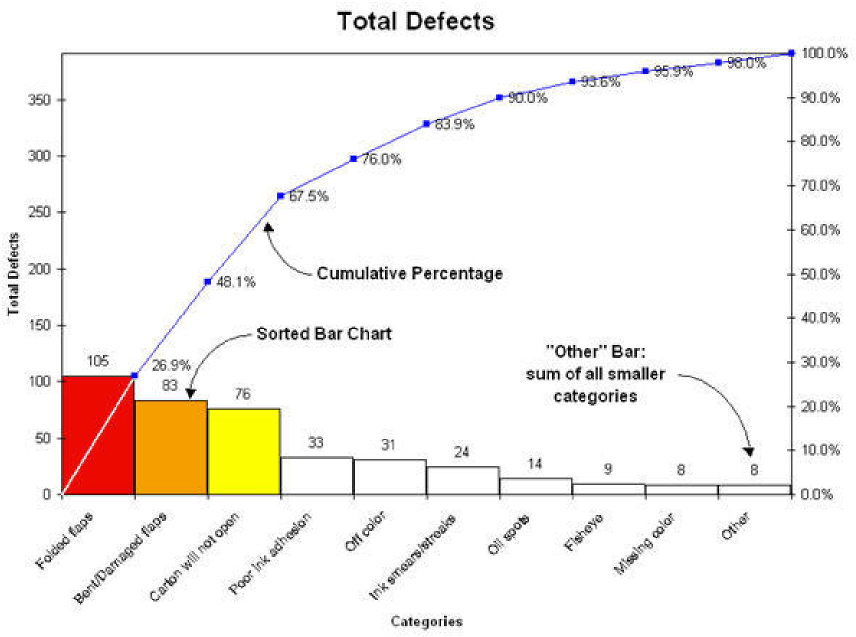
- In settings where privacy matters, a DHT can support privacy-preserving applications that just keep all data on the owner’s device. With a standard cloud we would have to trust the cloud operator/provider.

Goals

- So… we want to support a DHT (get, put)
- Want it to have fast lookups, like inside a data center, but in a situation where we can't just have a membership managing service
- Need it to be tolerant of churn, hence "adaptive"

Insight into adaptation

- Many “things” in computer networks exhbit 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



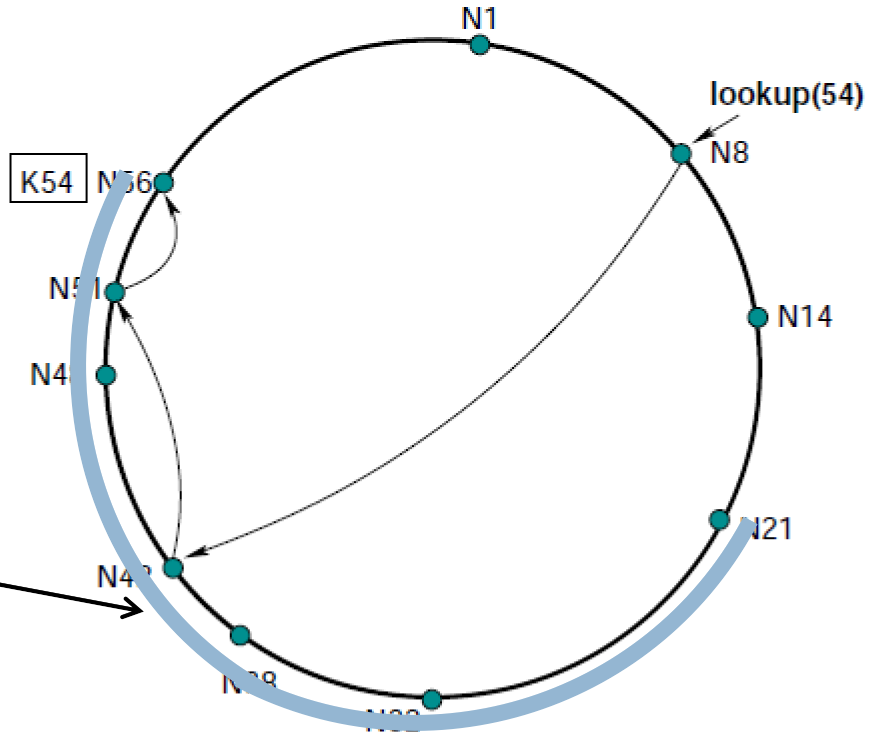
Beehive insight

- 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

- 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

- 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

- 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!

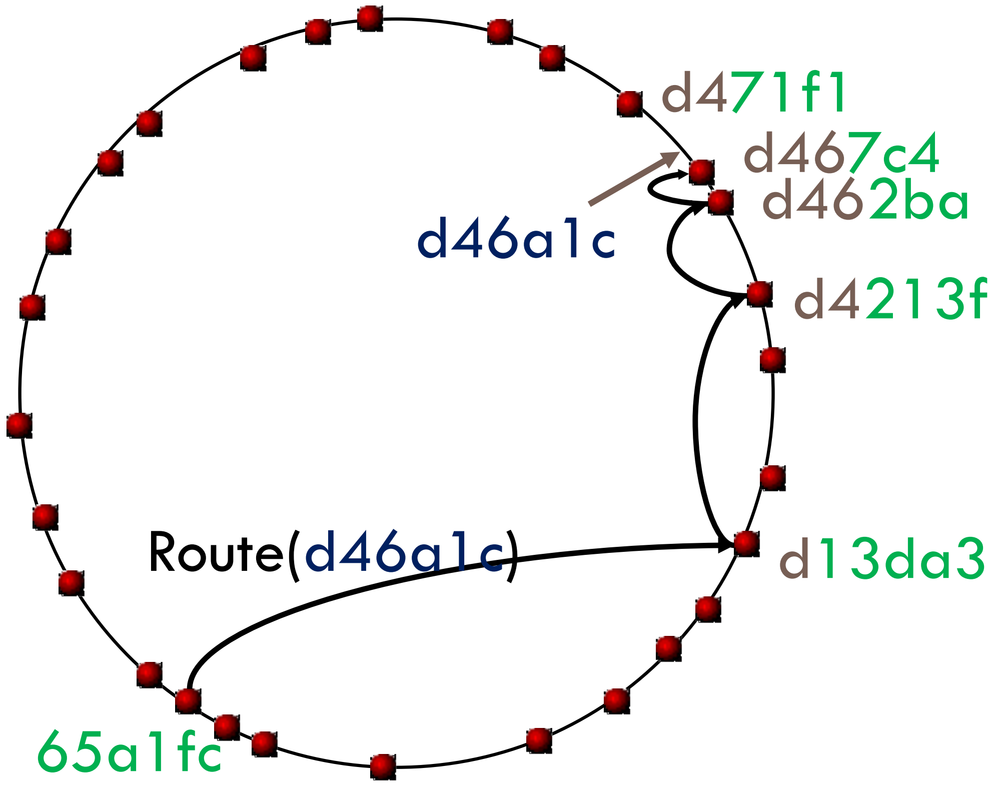
Notice interplay of ideas here

- 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 CS5412

Pastry

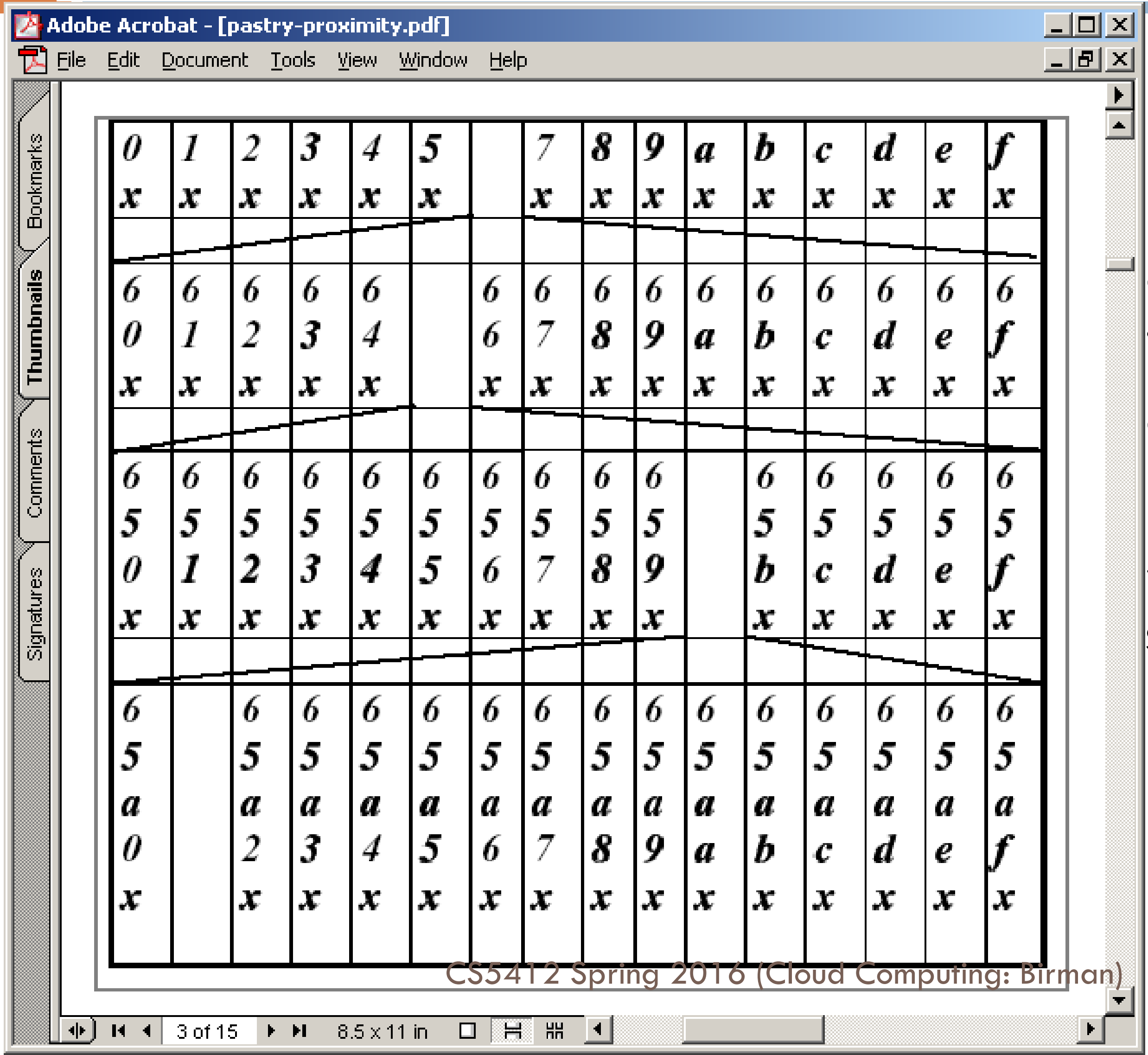
- A DHT much like Chord or Beehive
- But the goal here is to have more flexibility in picking finger links
  - In Chord, the node with hashed key H must look for the nodes with keys H/2, H/4, etc….
  - In Pastry, there are a *set* of possible target nodes and this allows Pastry flexibility to pick one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

Pastry also uses a circular number space



- 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)


Adobe Acrobat - [pastry-proximity.pdf]
File Edit Document Tools View Window Help
Bookmarks
Thumbnails
Comments
Signatures
3 of 15
8.5 x 11 in

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

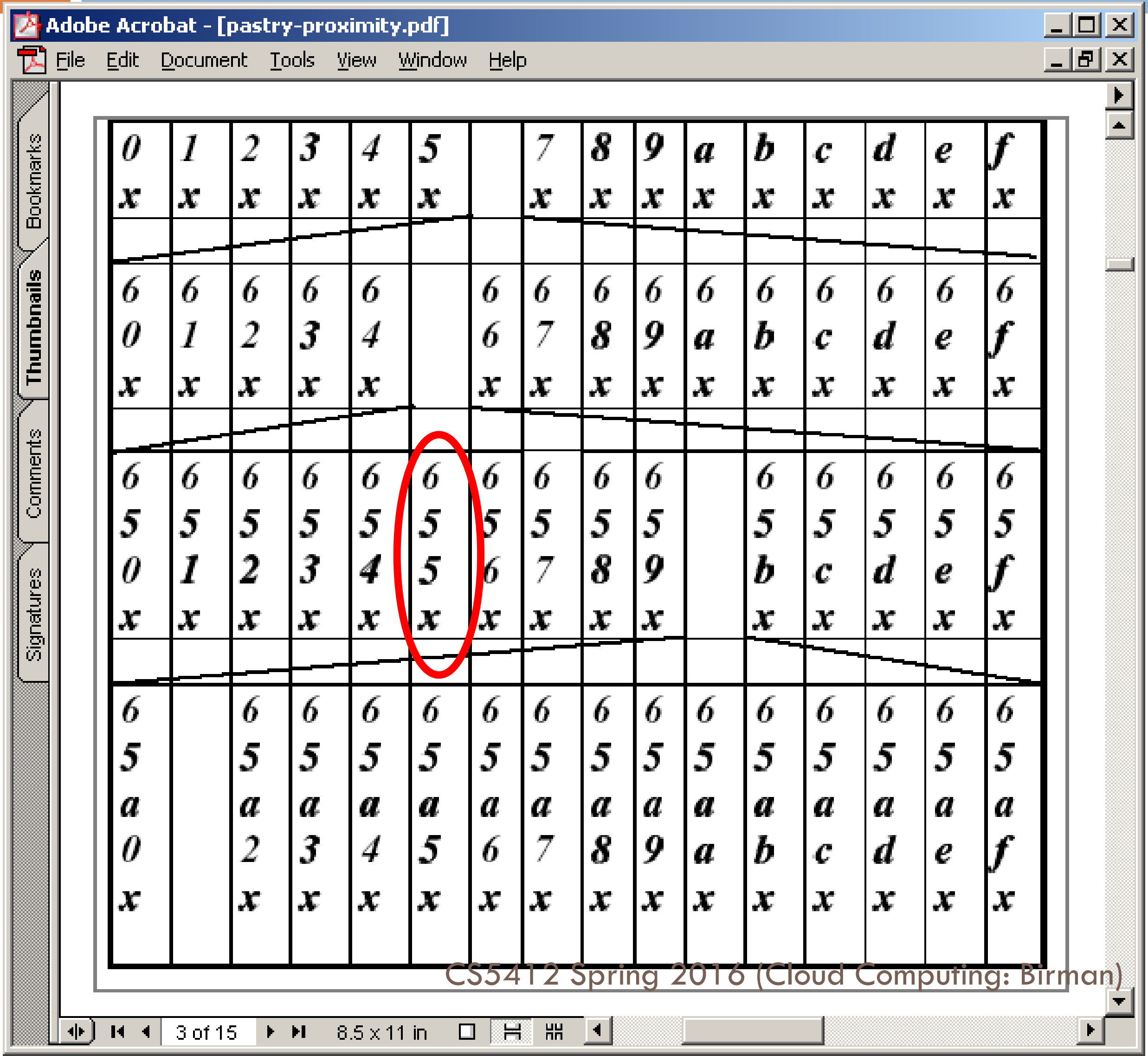
Pastry join

- 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

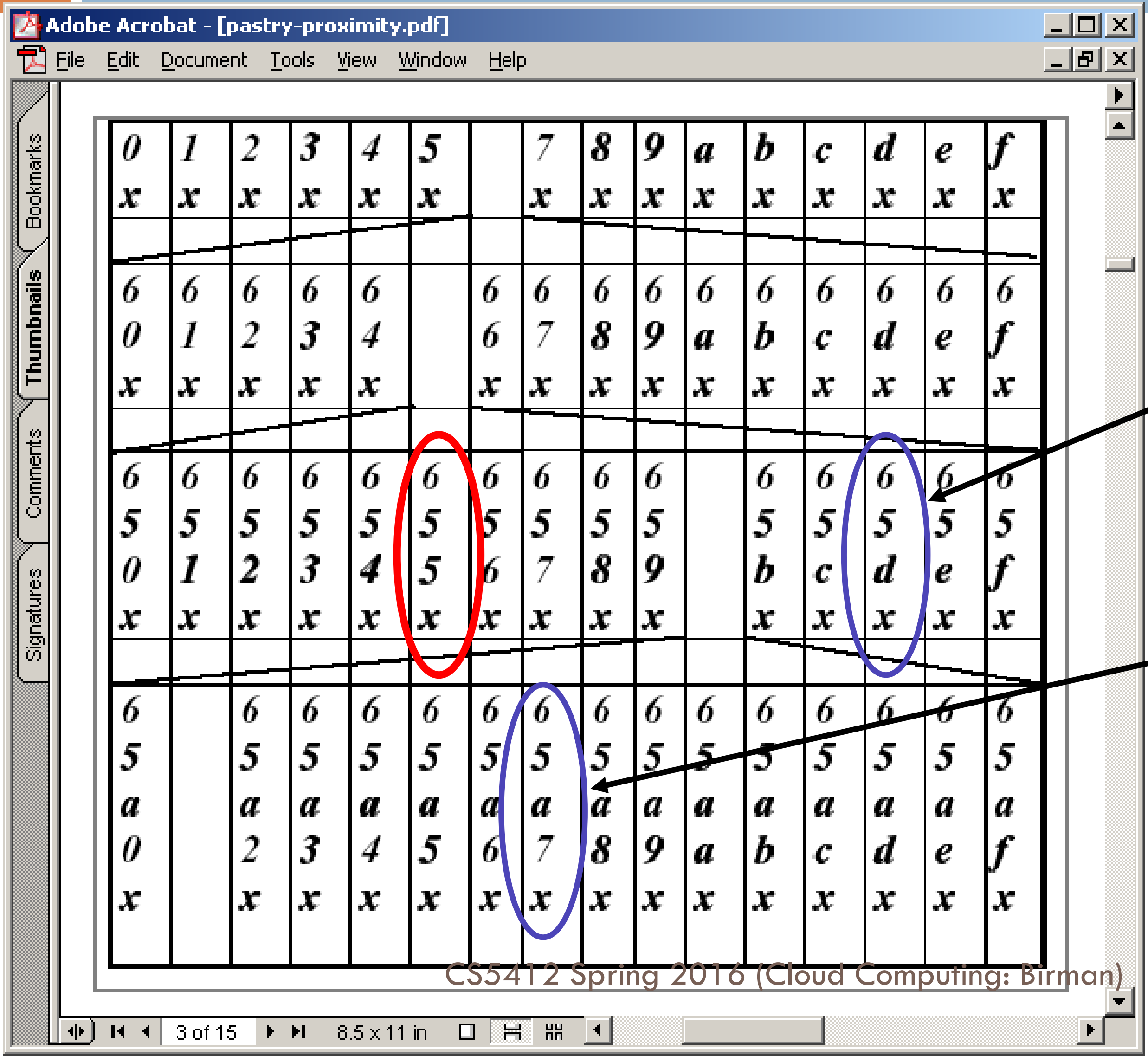
- Noticed by leaf set neighbors when leaving node doesn't respond
  - Neighbors ask highest and lowest nodes in leaf set for new leaf set
- Noticed by routing neighbors when message forward fails
  - Immediately can route to another neighbor
  - Fix entry by asking another neighbor in the same "row"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

For instance, this neighbor fails



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

Ask other neighbors



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

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
    - How might an attacker try 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…

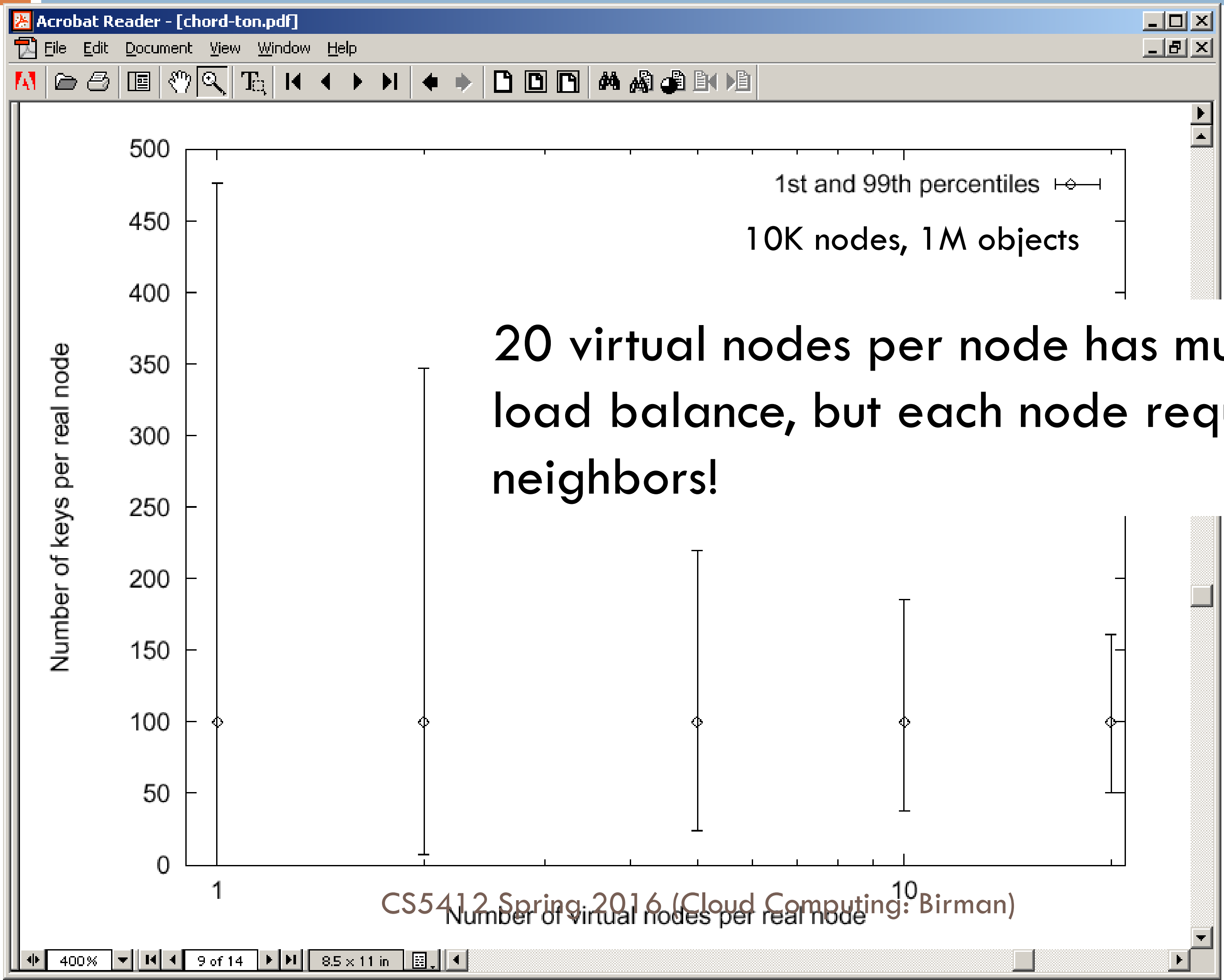
Load balancing

- 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!

Fireflies

- 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 repell many kinds of disruptive attacks
- We won't have time to look at the details today

Another major 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)!*

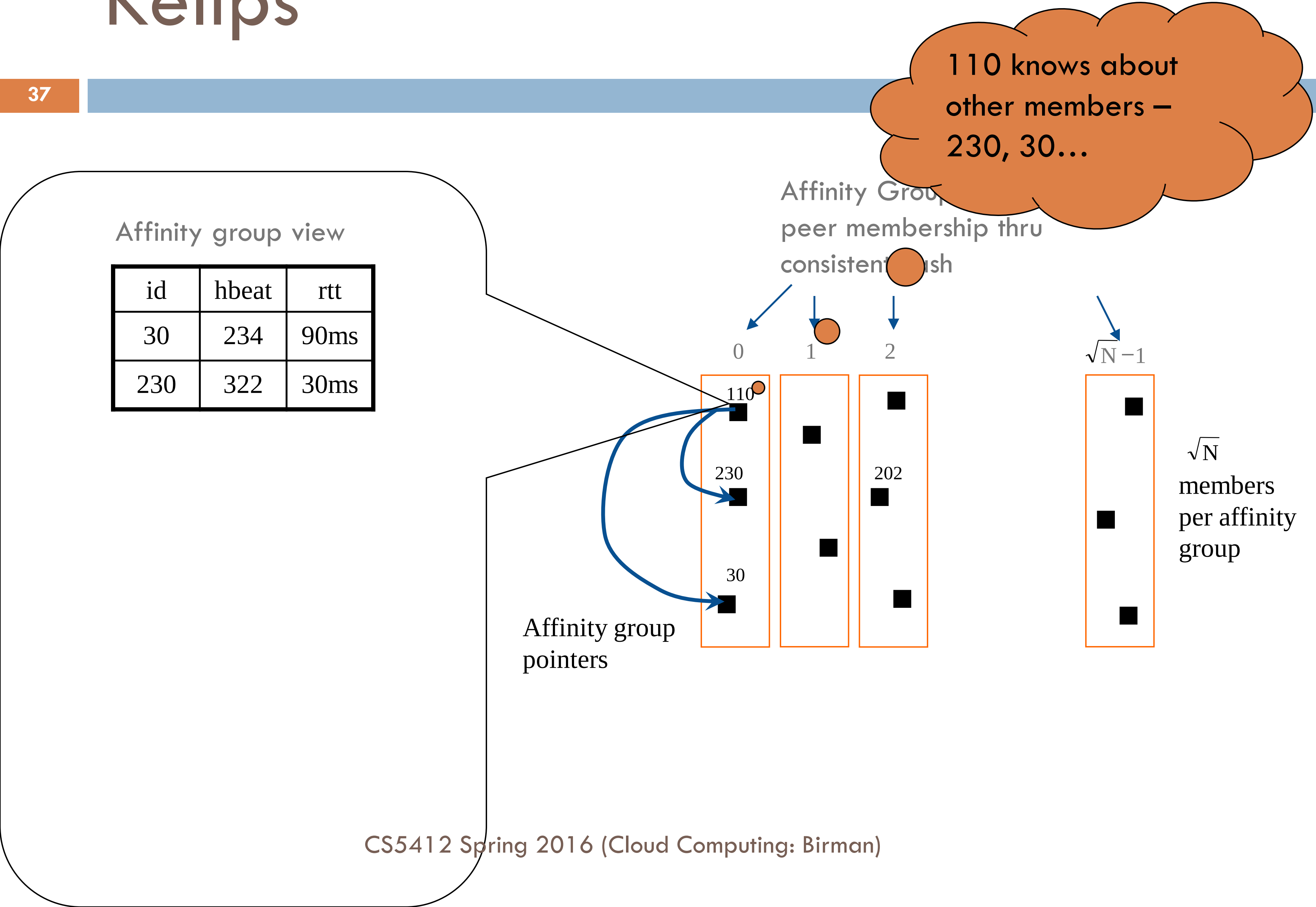
Rationale?

- 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

Affinity group view

id	hbeat	rtt
30	234	90ms
230	322	30ms


110 knows about other members – 230, 30…
Affinity Group: peer membership thru consistent hash
0
1
2
$\sqrt{N}-1$
110
230
30
202
$\sqrt{N}$ members per affinity group
Affinity group pointers

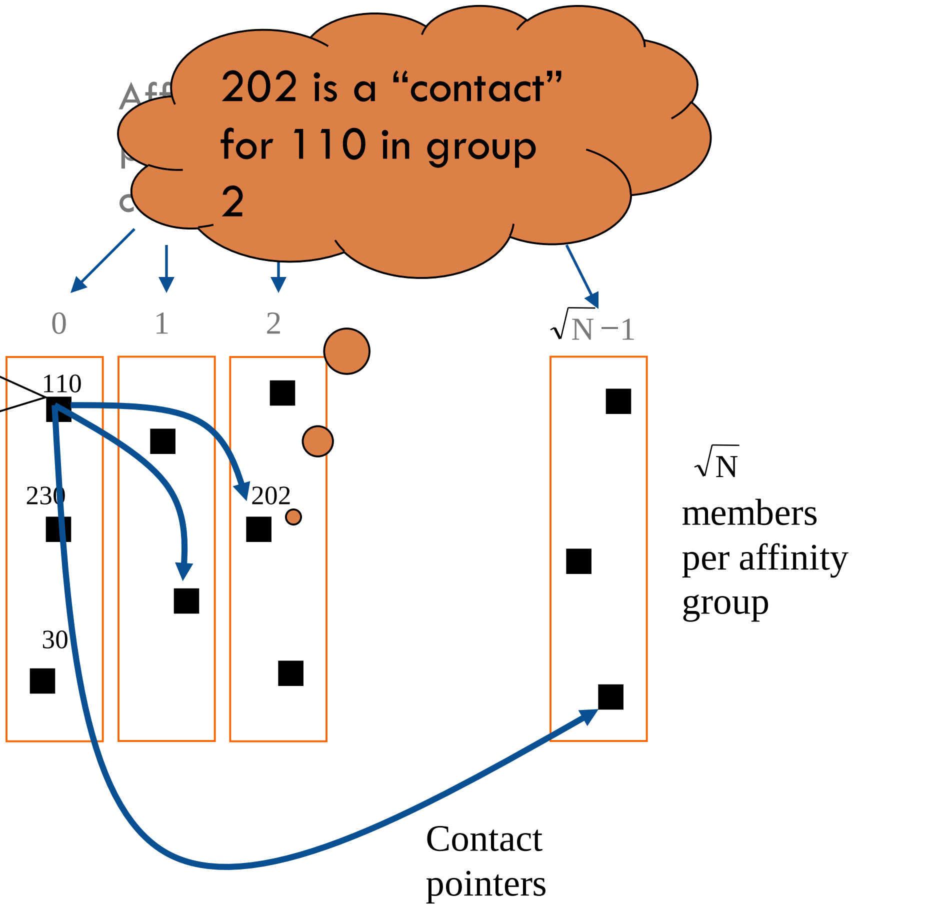
Kelips

Affinity group view

id	hbeat	rtt
30	234	90ms
230	322	30ms

Contacts

group	contactNode
…	…
2	202


202 is a "contact" for 110 in group 2
0
1
2
$\sqrt{N}-1$
110
230
30
202
$\sqrt{N}$ members per affinity group
Contact pointers

Kelips

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

Affinity group view

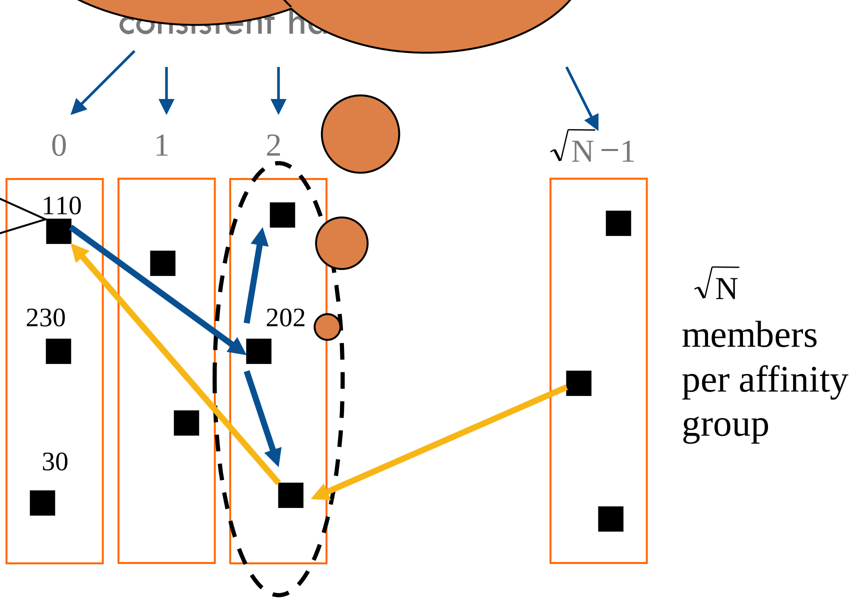
id	hbeat	rtt
30	234	90ms
230	322	30ms

Contacts

group	contactNode
…	…
2	202

Resource Tuples

resource	info
…	…
cnn.com	110



$\sqrt{N}$ members per affinity group

Gossip protocol replicates data cheaply

How it works

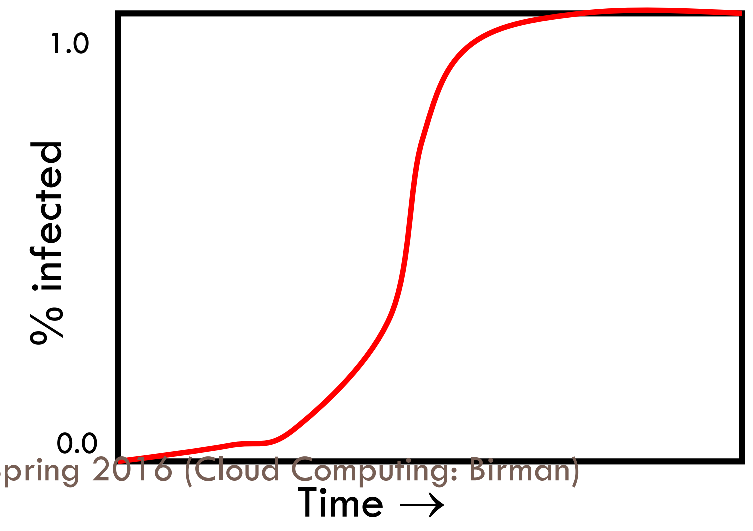
- Kelips is *entirely* gossip based!
  - Gossip about membership
  - Gossip to replicate and repair data
  - Gossip about "last heard from" time used to discard failed nodes
- Gossip "channel" uses fixed bandwidth
  - … fixed rate, packets of limited size

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

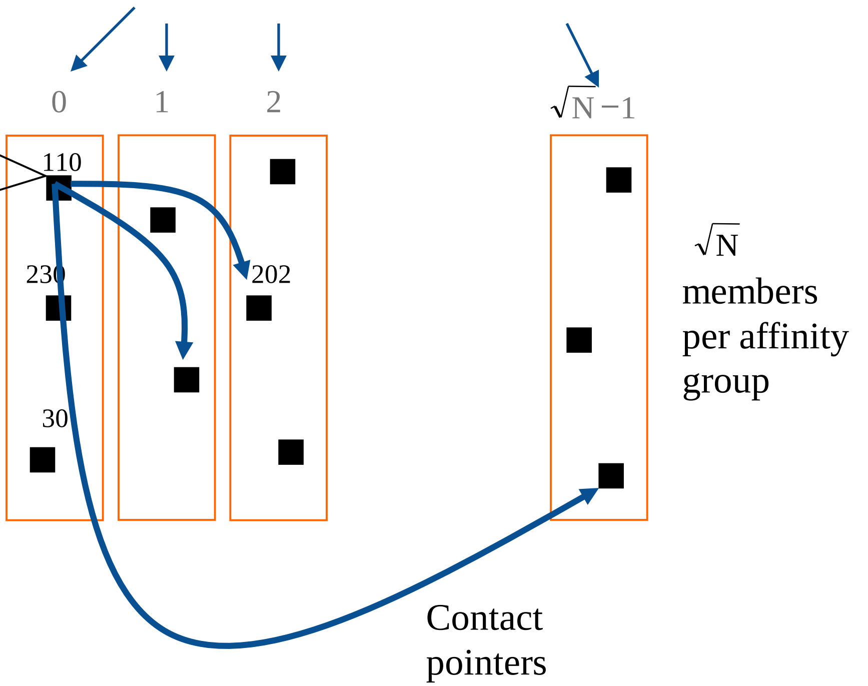
Affinity group view

id	hbeat	rtt
30	234	90ms
230	322	30ms

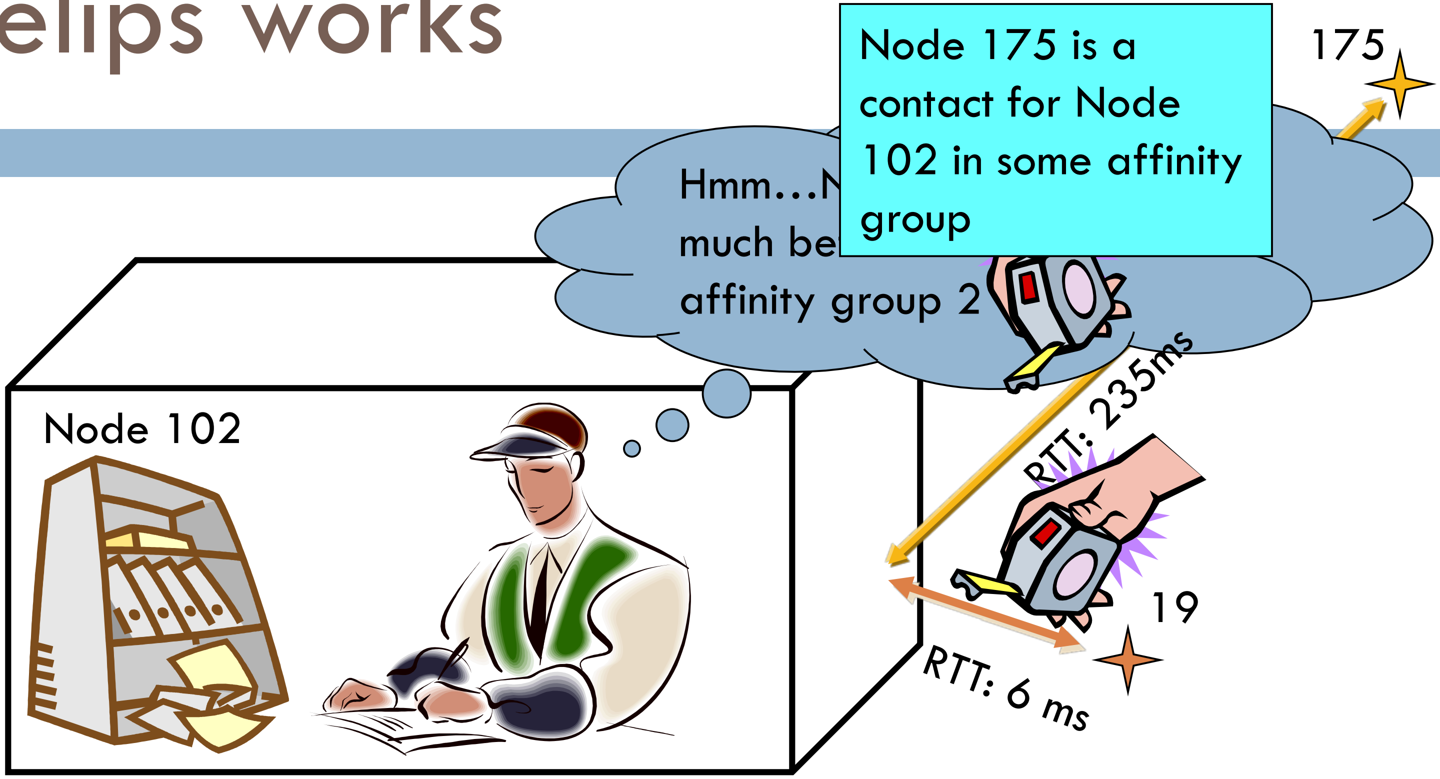
Contacts

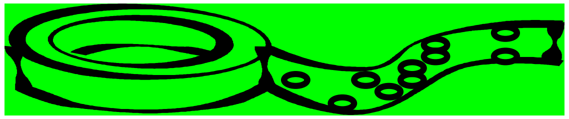
group	contactNode
…	…
2	202

Affinity Groups: peer membership thru consistent hash



How Kelips works


Node 175 is a contact for Node 102 in some affinity group
175
Hmm…N much be affinity group 2
Node 102
RTT: 235ms
19
RTT: 6 ms

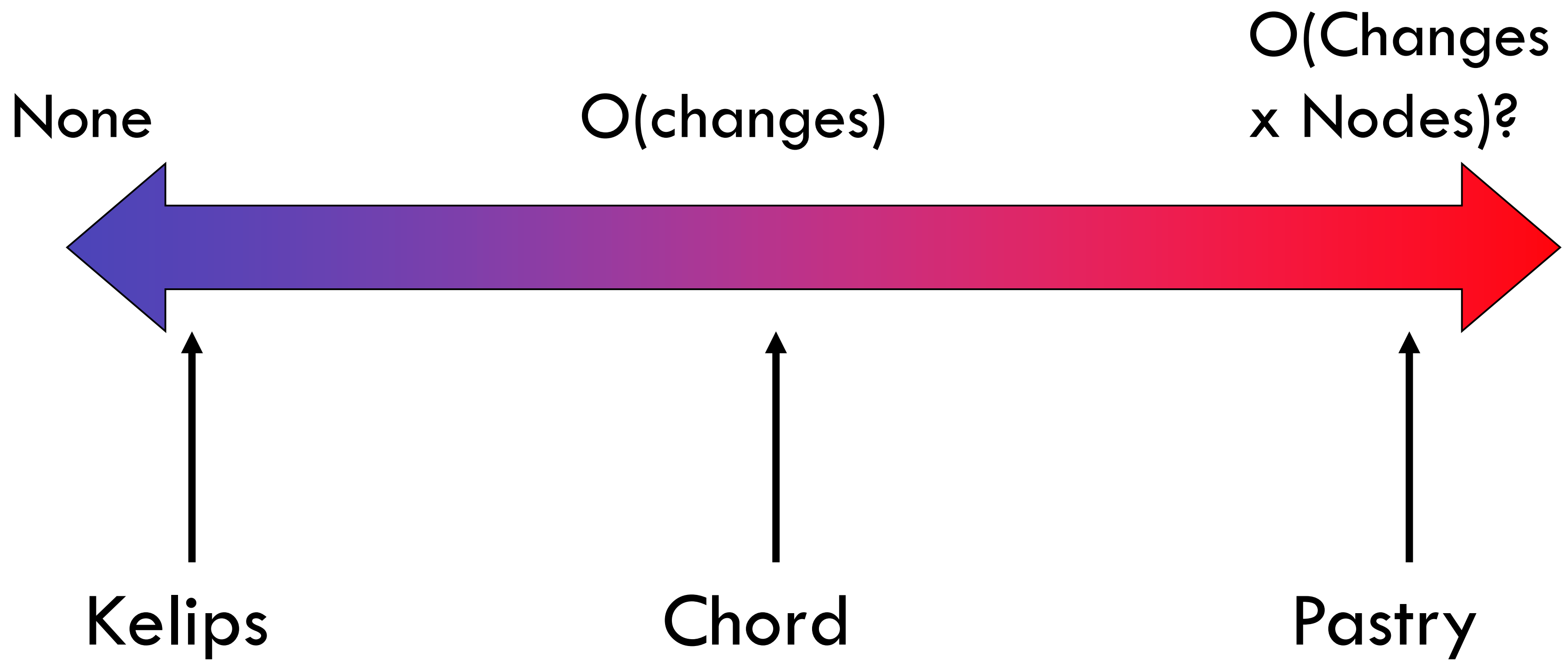

Gossip data stream

- 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

Control traffic load generated by churn



Summary

- 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