Using Gossip to Build Scalable Services

Ken Birman, CS514

Dept. of Computer Science

Cornell University

Our goal today

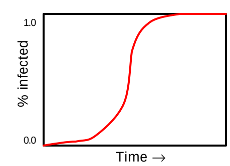
- Bimodal Multicast (last week) offers unusually robust event notification. It combines UDP multicast+ gossip for scalability
- What other things can we do with gossip?
- Today:
  - Building a "system status picture"
  - Distributed search (Kelips)
  - Scalable monitoring (Astrolabe)

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
  - Now what?

Dealing with full messages

- One option: pick random data
  - Randomly sample in the two sets
  - Now, a typical message contains 1/k of the "live" information. How does this impact the epidemic?
- Works for medium sizes, say 10,000 nodes…
  - K side-by-side epidemics… each takes log(N) time
  - Basically, we just slow things down by a factor of k… If a gossip message talks about 250 processes at a time, for example, and there are 10,000 in total, 40x slowdown

Really big systems

- With a huge system, instead of a constant delay, the slowdown will be a factor more like O(N)
- For example:
  - 1 million processes. Can only gossip about 250 at a time…
  - … it will take 4000 "rounds" to talk about all of them even once!

Now would need hierarchy

- Perhaps the million nodes are in centers of size 50,000 each (for example)
- And the data centers are organized into five corridors each containing 10,000
  - Then can use our 10,000 node solution on a per-corridor basis
  - Higher level structure would just track "contact nodes" on a per center/per corridor basis.

Generalizing

- We could generalize and not just track "last heard from" times
  - Could also track, for example:
    - IP address(es) and connectivity information
    - System configuration (e.g. which services is it running)
    - Does it have access to UDP multicast?
    - How much free space is on its disk?
- Allows us to imagine an annotated map!

Google mashup

- Google introduced idea for the web
  - You take some sort of background map, like a road map
  - Then build a table that lists coordinates and stuff you can find at that spot
    - Corner of College and Dryden, Ithaca: Starbucks…
- Browser shows pushpins with popups

Our “map” as a mashup

- We could take a system topology picture
  - And then superimpose “dots” to represent the machines
  - And each machine could have an associated list of its properties
- It would be a live mashup! Changes visible within log(N) time

Uses of such a mashup?

- Applications could use it to configure themselves
  - For example, perhaps they want to use UDP multicast within subsystems that can access it, but build an overlay network for larger-scale communication where UDP multicast isn’t permitted
- Would need to read mashup, think, then spit out a configuration file “just for me”

Let’s look at a second example

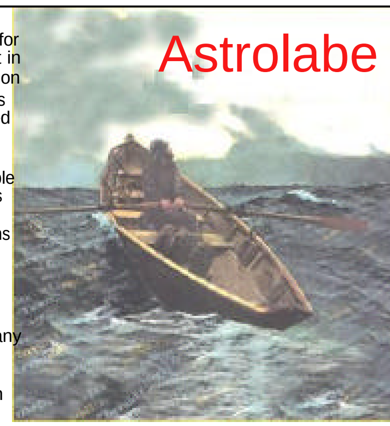
- Astrolabe system uses gossip to build a whole distributed database
- Nodes are given an initial location – each knows its “leaf domain”
- Inner nodes are elected using gossip and “aggregation” (we’ll explain this)
- Result is a self-defined tree of tables…

Astrolabe

- Intended as help for applications adrift in a sea of information
- Structure emerges from a randomized gossip protocol
- This approach is robust and scalable even under stress that cripples traditional systems

Developed at RNS, Cornell

- By Robbert van Renesse, with many others helping…
- Today used extensively within Amazon.com

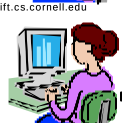


Astrolabe is a flexible monitoring overlay

swift.cs.cornell.edu

Name	Time	Load	Weblogic?	SMTP?	Word Version
swift	2271	1.8	0	1	6.2
falcon	1971	1.5	1	0	4.1
cardinal	2004	4.5	1	0	6.0

Periodically, pull data from monitored systems



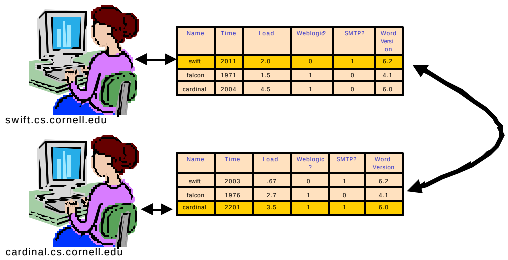
cardinal.cs.cornell.edu

Name	Time	Load	Weblogic?	SMTP?	Word Version
swift	2003	.67	0	1	6.2
falcon	1976	2.7	1	0	4.1
cardinal	2231	1.7	1	1	6.0

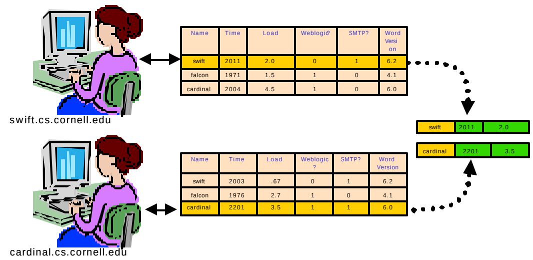
Astrolabe in a single domain

- Each node owns a single tuple, like the management information base (MIB)
- Nodes discover one-another through a simple broadcast scheme (“anyone out there?”) and gossip about membership
  - Nodes also keep replicas of one-another’s rows
  - Periodically (uniformly at random) merge your state with some else…

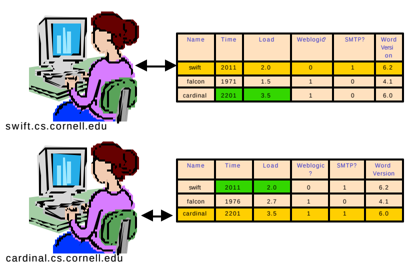
State Merge: Core of Astrolabe epidemic



State Merge: Core of Astrolabe epidemic



State Merge: Core of Astrolabe epidemic



Observations

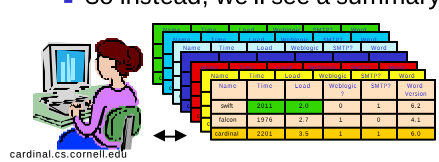
- Merge protocol has constant cost
  - One message sent, received (on avg) per unit time.
  - The data changes slowly, so no need to run it quickly – we usually run it every five seconds or so
  - Information spreads in O(log N) time
- But this assumes bounded region size
  - In Astrolabe, we limit them to 50-100 rows

Big systems…

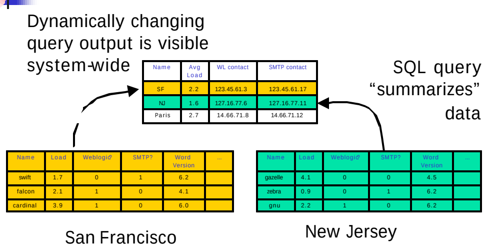
- A big system could have *many* regions
  - Looks like a pile of spreadsheets
  - A node only replicates data from its neighbors within its own region

Scaling up… and up…

- With a stack of domains, we don't want every system to "see" every domain
  - Cost would be huge
- So instead, we'll see a summary



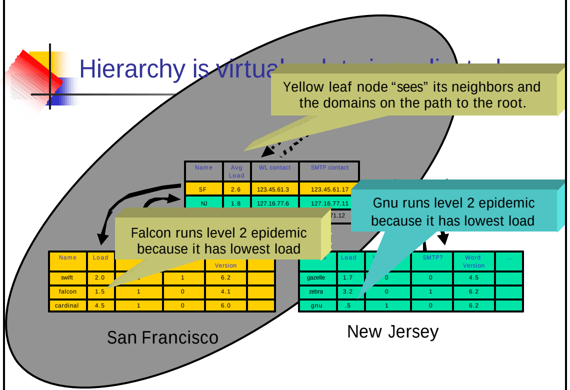
Astrolabe builds a hierarchy using a P2P protocol that "assembles the puzzle" without any servers



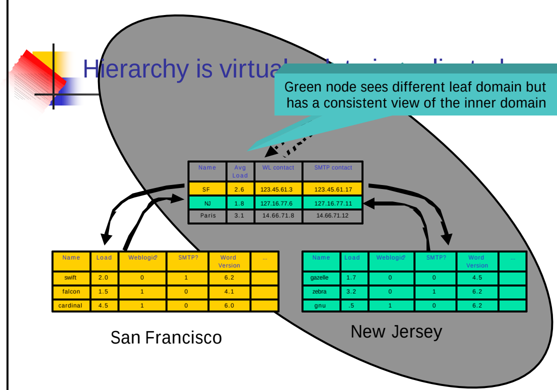
Large scale: "fake" regions

- These are
  - Computed by queries that summarize a whole region as a single row
  - Gossiped in a read-only manner within a leaf region
- But who runs the gossip?
  - Each region elects "k" members to run gossip at the next level up.
  - Can play with selection criteria and "k"

Hierarchy is virtual



Hierarchy is virtual



Worst case load?

- A small number of nodes end up participating in $O(\log_{fanout} N)$ epidemics
  - Here the fanout is something like 50
  - In each epidemic, a message is sent and received roughly every 5 seconds
- We limit message size so even during periods of turbulence, no message can become huge.

Who uses stuff like this?

- Amazon uses Astrolabe throughout their big data centers!
  - For them, Astrolabe plays the role of the mashup we talked about earlier
  - They can also use it to automate reaction to temporary overloads

Example of overload handling

- Some service S is getting slow…
  - Astrolabe triggers a "system wide warning"
- Everyone sees the picture
  - "Oops, S is getting overloaded and slow!"
  - So everyone tries to reduce their frequency of requests against service S

Another use of gossip: Finding stuff

- This is a problem you've probably run into for file downloads
  - Napster, Gnutella, etc
  - They find you a copy of your favorite Red Hot Chili Peppers songs
  - Then download from that machine
- At MIT, the Chord group turned this into a hot research topic!

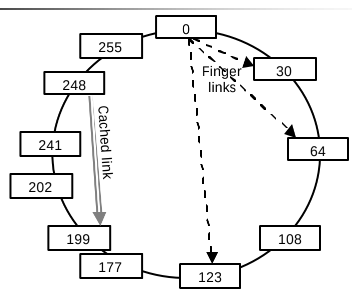
Chord (MIT group)

- The MacDonald's of DHTs
- A data structure mapped to a network
  - Ring of nodes (hashed id's)
  - Superimposed binary lookup trees
  - Other cached "hints" for fast lookups
- Chord is *not* convergently consistent

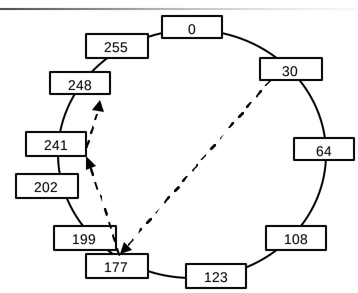
How Chord works

- Each node is given a random ID
  - By hashing its IP address in a standard way
- Nodes are formed into a ring ordered by ID
- Then each node looks up the node ½ across, ¼ across, $1/8^{th}$ across, etc
- We can do binary lookups to get from one node to another now!

Chord picture



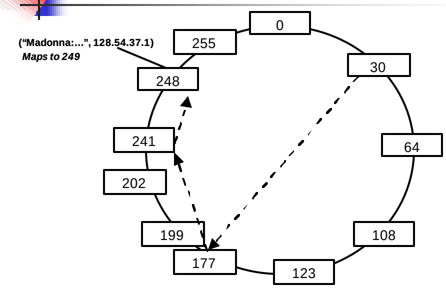
Node 30 searches for 249



OK... so we can look for nodes

- Now, store information in Chord
  - Each "record" consists of
    - A keyword or index
    - A value (normally small, like an IP address)
  - E.g:
    - ("Madonna:I'm not so innocent", 128.74.53.1)
- We map the index using the hash function and save the tuple at the closest Chord node along the ring

Madonna "maps" to 249



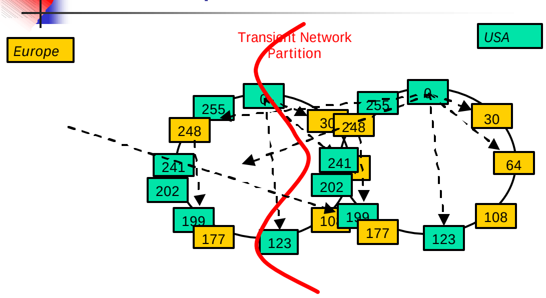
Looking for Madonna

- Take the name we're searching for (needs to be the *exact* name!)
- Map it to the internal id
- Lookup the closest node
- It sends back the tuple (and you cache its address for speedier access if this happens again!)

Some issues with Chord

- Failures and rejoins are common
  - Called the "churn" problem and it leaves holes in the ring
  - Chord has a self-repair mechanism that each node runs, independently, but it gets a bit complex
  - Also need to replicate information to ensure that it won't get lost in a crash

Chord can malfunction if the network partitions...



... so, who cares?

- Chord lookups can fai… it suffers from high overhea… en nodes churn
  - Loads surge i… n things are already disrupted… often, because of loads
  - And c… dict how long Chord might re… rupted once it gets that way
- W… case scenario: *Chord can become in… nsistent and stay that way*

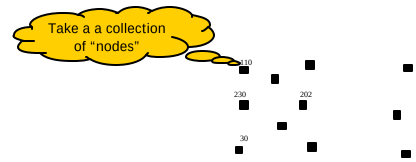
The Fine Print

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.

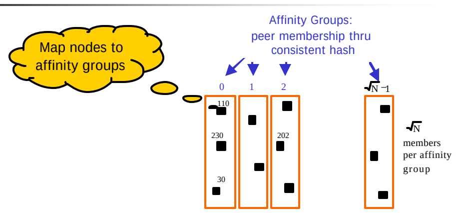
Can we do better?

- Kelips is a Cornell-developed "distributed hash table", much like Chord
- But unlike Chord it heals itself after a partitioning failure
- It uses gossip to do this...

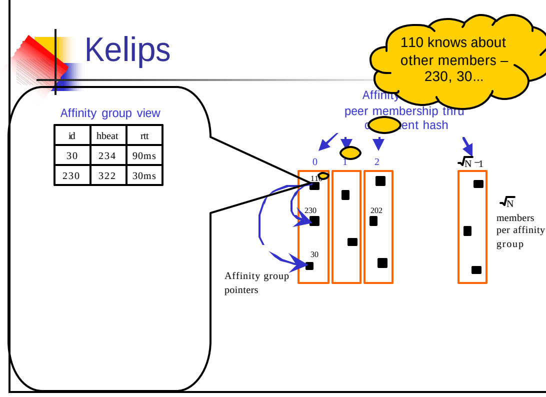
Kelips (Linga, Gupta, Birman)



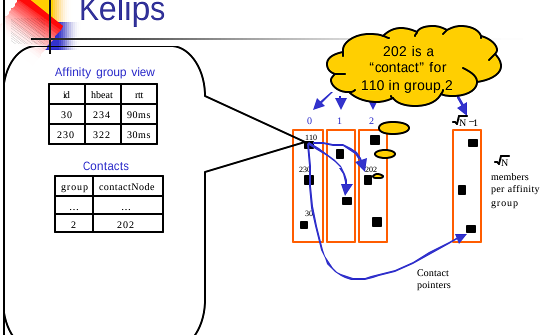
Kelips



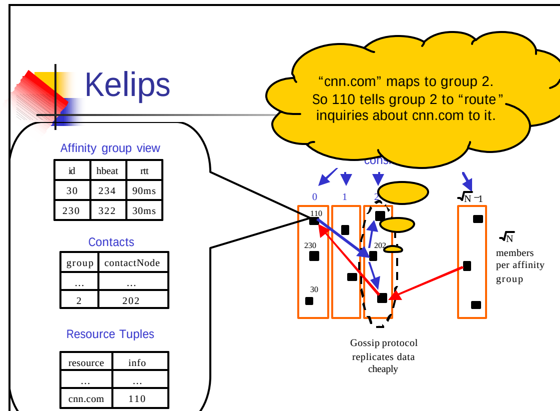
Kelips



Kelips



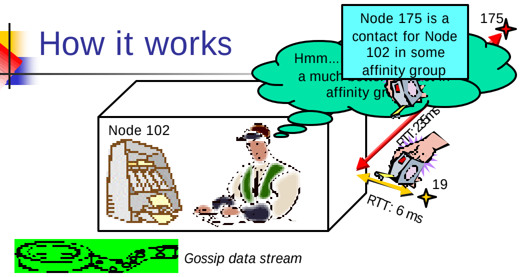
Kelips



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

How it works



- Heuristic: 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 √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*

Work in progress...

- Prakash Linga is extending Kelips to support multi-dimensional indexing, range queries, self-rebalancing
- Kelips has limited incoming "info rate"
  - Behavior when the limit is continuously exceeded is not well understood.
  - Will also study this phenomenon

Kelips isn't alone

- Back at MIT, Barbara Liskov built a system she calls Epichord
  - It uses a Chord-like structure
  - But it also has a background gossip epidemic that heals disruptions caused by crashes and partitions
- Epichord is immune to the problem Chord can suffer

Connection to self-stabilization

- Self-stabilization theory
  - Describe a system and a desired property
  - Assume a failure in which code remains correct but node states are corrupted
  - Proof obligations: property reestablished within bounded time
- Epidemic gossip: remedy for what ails Chord!
  - *c.f.* Epichord (Liskov)
  - *Kelips and Epichord are self-stabilizing!*

Beyond self-stabilization

- Tardos poses a related problem
  - Consider behavior of the system *while an endless sequence of disruptive events occurs*
  - System never reaches a quiescent state
  - Under what conditions will it still behave correctly?
- Results of form "if disruptions satisfy j then correctness property is continuously satisfied"
- Hypothesis: with convergent consistency we may be able to prove such things

Convergent consistency

- A term used for gossip algorithms that
  - Need log(N) time to "mix" new events into an online system
  - Reconverge to their desired state once this mixing has occurred
- They can be overwhelmingly robust because gossip can explore an exponential number of data routes!

Summary

- Gossip is a powerful concept!
  - We've seen gossip used for tracking membership and other data (live mashups)
  - Gossip for data mining and monitoring
  - Gossip for search in big peer-to-peer nets
  - Gossip used to "aggregate" system-wide properties such as load, health of services, etc.
- Coming next: Even MORE uses of gossip!