CS5412 / LECTURE 12: GOSSIP PROTOCOLS

Ken Birman

Spring, 2022

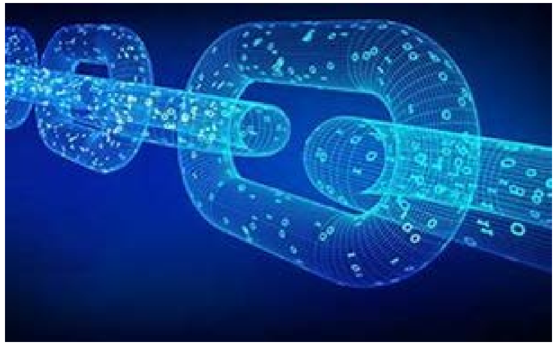
BUILDING SCALABLE INFRASTRUCTURES

Within a cloud computing environment we often need to manage very large pools of computers or services.

What is the best way to monitor and manage this kind of deployment?

In lectures 11 and 12 we will discuss the concept of using “gossip” as the basis for an unusually scalable style of solution. Amazon uses it in S3!

GOSSIP IS USED IN BLOCKCHAIN!

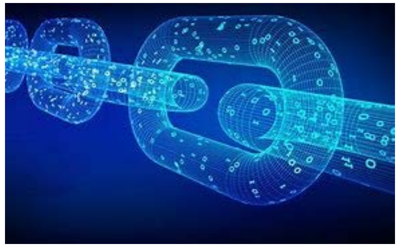


Hard to deny that Blockchain is big business!

Gossip is used to distribute updates in *permissionless* blockchain, where the participants are anonymous but might not be cooperating

Gossip makes it is very hard for attackers to disrupt updates.

BYZANTINE PARTICIPANTS



Blockchain must tolerate (a few) Byzantine participants.

Byzantine participant

They seek to damage or disrupt the chain or to favor or delay transactions from certain parties, attempt to DDoS links, etc.

Even a Byzantine participant can’t modify or forge messages: messages are cryptographically signed.

GOSSIP 101

Push-Pull in Action!

Suppose that Anne tells me something.

I’m sitting next to Fred, and I tell him

Later, he tells Mimi and I tell Frank

Not cooperating

Each round doubles the number of people who know the secret.

This is an example of a *push* epidemic

Push-pull occurs if we exchange data

PUSH/PULL GOSSIP

Push-Pull in Action!

Combines push and pull, but requires an RPC-style of interaction:

- Process P decides to gossip with process Q
- P sends Q some form of concise "digest" of information available.
- Q sends back its own digest, plus a list of items it wants from P.
- P responds by sending those items, plus a list of items it wants from Q.
- Q sends the requested items.

This avoids sending large duplicate objects

EACH MESSAGE IS TREATED SEPARATELY

We send requests but don’t actually “wait” for a response

This way if our peer turns out to be faulty or uncooperative, nothing bad can happen – we sent a first stage push-pull message but got not reply and don’t have any associated state that lingers, etc

If it works, great. If not, well, the next message will go to someone else.

LIMITED WORK PER ROUND

Think about maximum size gossip messages – there is always a limit.

All of these patterns have a fixed maximum number of messages that will be sent and received.

So, each process has a limit on how many bytes it will need to send for each gossip round.

REASONABLE NETWORK ASSUMPTIONS

Gossip doesn’t assume all-to-all connectivity: outside the datacenter we often see more of a graph of peering connections.

Instead, we tend to assume an “expander” graph: every node is reachable from every other node within O(log(N)) hops. We also assume that the network is robust to link failures or Byzantine DDoS attack.

These are considered to be very practical, reasonable assumptions

SIZE CONSTRAINT

For gossip to really have constant cost at each participant, we need to decide on a maximum message size.

Messages can grow to that maximum, but not beyond

But with unlimited numbers of processes, even if the events we gossip about are rare, the amount of information to share could grow as O(N)

TRICKS FOR LIMITING MESSAGE SIZE

Many systems only gossip about “recent” information.

The theory is that older data is probably stale or wrong in any case.

Then the issue is “how many events can happen in Δ time?” This may be more manageable

GOSSIP SCALES VERY NICELY BUT PURE PUSH OR PURE PULL ISN’T IDEAL

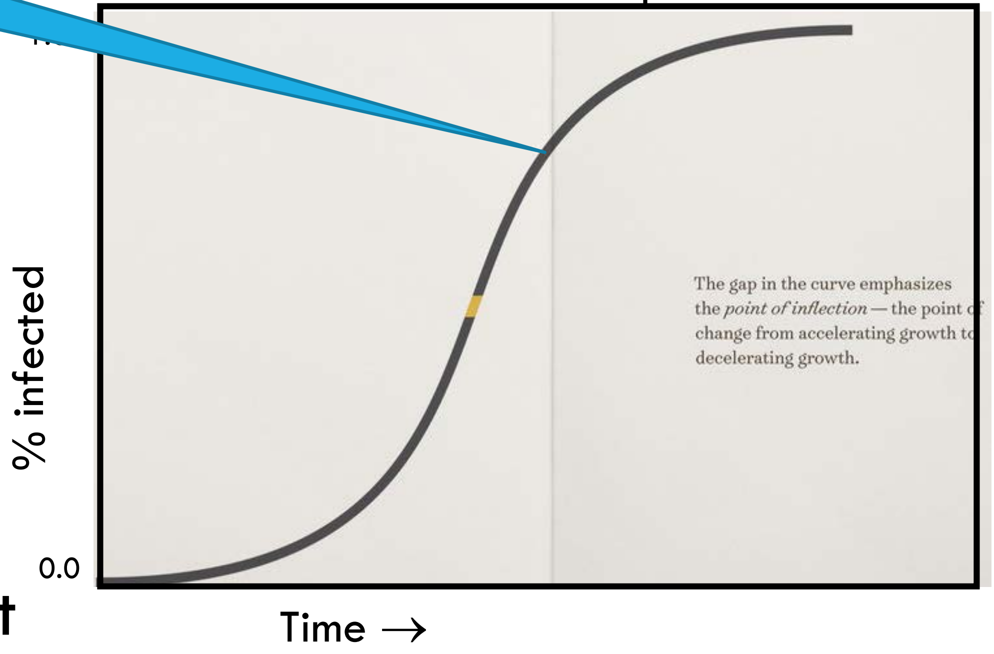
“Reinfection” eventually becomes dominant

Participant load is <u>constant</u>, independent of size of the system.

Total network load linear in system size.

Information spreads in log(N) time, yet that limit on work per process remains in effect!

Pure push case: Needs a long time to reach the last few processes



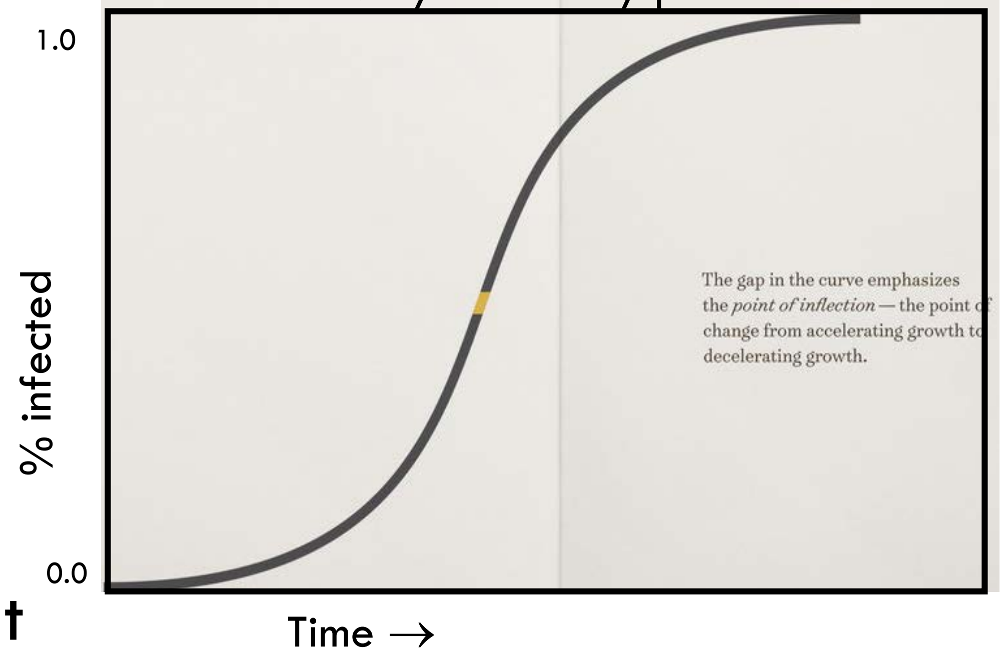
GOSSIP WITH PUSH-PULL IS BEST

Participant load is constant, independent of size of the system.

Total network load linear in system size.

Information spreads in log(N) time, yet that limit on work per process remains in effect!

Push-Pull Terminates quickly and reaches every non-faulty process



ONE SMALLISH RISK

What if everyone decides to gossip to the same process all at once?

Selection of the target is random… it could happen.

But it is very unlikely and in fact the receiver could just ignore some messages. *Gossip doesn't require reliable messaging.*

GOSSIP IN DISTRIBUTED SYSTEMS

We can even gossip about membership

- Need a bootstrap mechanism, but then discuss failures, new members
- This feature is *not* used in permissionless blockchain, but it is used in permissioned blockchain. In fact this is what distinguishes those models.

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

WHY "IF WE AREN'T IN A HURRY?"

Gossip is very robust, but log(N) time might not be fast.

Normally we run one round every second or so.

A data center with 100,000 computers would have log(N) = 17, so when something important happens, it would take 17 seconds to reach all nodes.

Size limit of messages can also be an issue

WHAT IF WE ACTUALLY ARE IN A HURRY?

One option is to mix gossip with a second mechanism.

UDP multicast can be useful here. This is an old and not-often used feature of the Internet UDP protocol (user datagram protocol, sometimes called “unreliable datagrams” to contrast with TCP).

- Instead of having just one server for each IP address, UDP datagrams allow multiple servers to attach to the same shared IP address
- With this feature, the UDP multicast will go to all receivers

SOME WARNINGS…

Many datacenters disable the router feature UDP multicast requires.

If they do this, it won’t work even though Linux might allow you to bind to that shared class-D multicast IP address, and to send to it – the messages just won’t reach other machines.

Also, because UDP multicast isn’t reliable, some receivers could receive the message, but others might drop it – silently. No retransmissions occur.

BIMODAL MULTICAST

This was a protocol that uses UDP multicast as a first step, then “fills any gaps” using gossip.

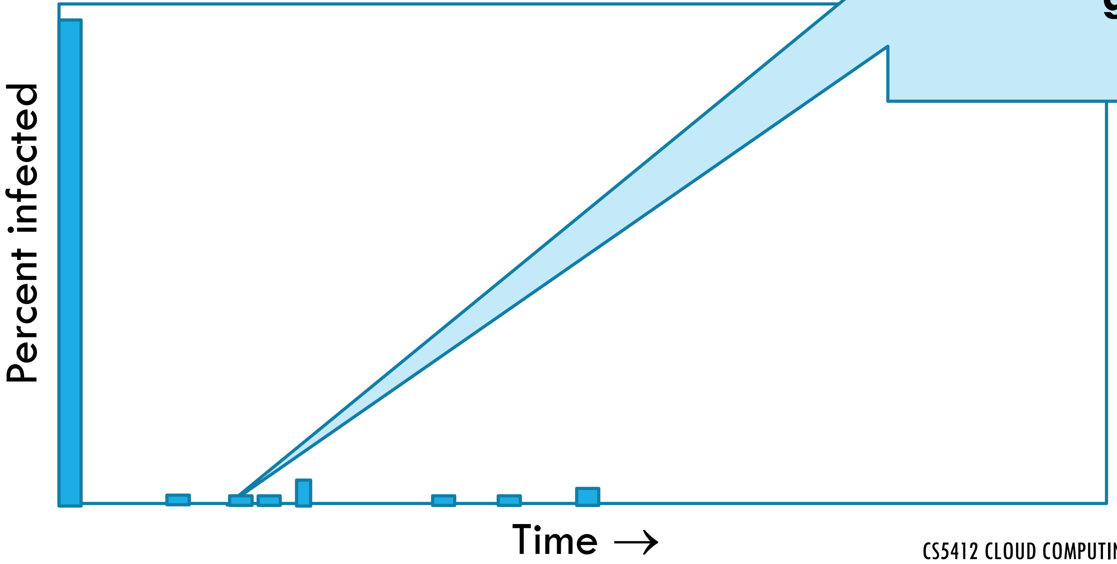
Cool trick: If some node is asked to share the same message twice via gossip, instead it resends the UDP multicast. That way if a few processes seem to have missed some message, it gets retransmitted soon.

We get two delivery delay “curves”: one for UDP multicast, the second for gossip to fill the tiny number of remaining gaps.

A BIMODAL DELIVERY CURVE

In this picture, 99.9% of the messages are received via UDP multicast. So in a datacenter with 100,000 machines, only 100 or so miss it.

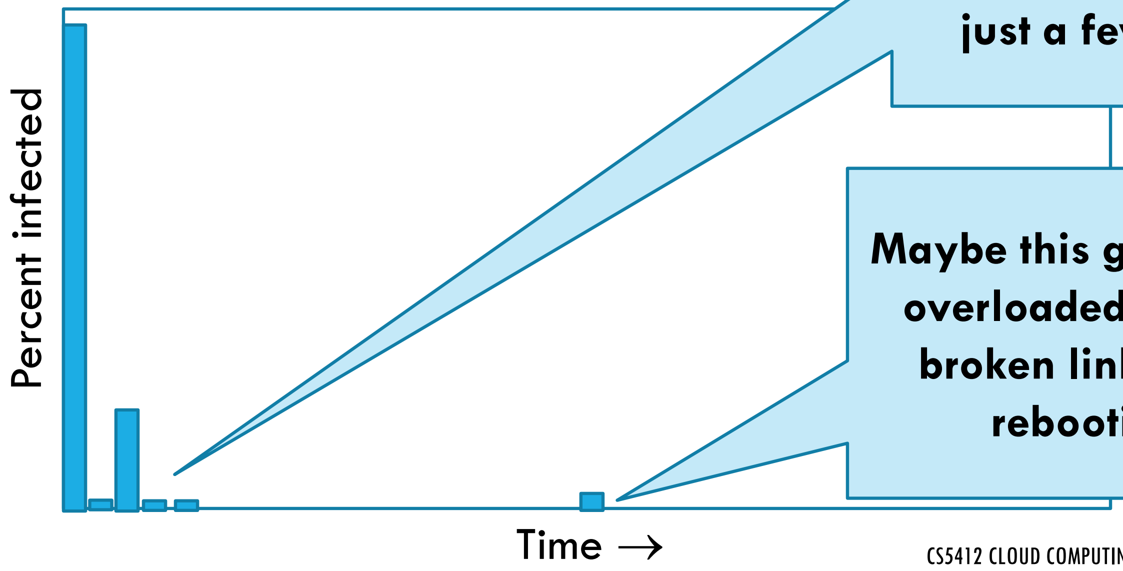
But then the gossip spread could be a bit slow



Gossip kicks in but takes time to fill the gaps

A BIMODAL DELIVERY CURVE

With a second UDP multicast, our curve looks much better!



Gossip kicks in, then a second UDP multicast occurs, and this leaves just a few stagglers

Maybe this guy is badly overloaded, or had a broken link, or was rebooting…

IS GOSSIP ATOMIC MULTICAST?

Not really. It lacks a total order guarantee.

But some systems add a timestamp and deliver multicasts in timestamp order, breaking ties using the IP address of the senders.

At time τ, they have some estimated δ such that no messages are expected with timestamp $\leq \tau\text{-}\delta$ (if one were to turn up, they would silently discard it). When messages become “stable” in this sense, they deliver them.

STRAGGLERS CAN MISS MESSAGES

A machine that ran very slow for a while and missed messages will see them via gossip, but it may be too late to deliver them in correct order.

In effect, a mistake was already made and it is too late to fix it.

So… bimodal multicast can approximate atomic multicast, but only to some probabilistic limit. It won't (can't) be flawless.

GOSSIP ABOUT MEMBERSHIP

Start with a bootstrap protocol

- For example, processes go to some web site. On it they find a dozen nodes where the system has been stable for a long time
- Pick one at random

It sends back a membership list. Now your node is up and running!

- Then track "processes I've heard from recently" and "processes other nodes have heard from recently"
- Generally, use push gossip to spread the word

EXAMPLE: THE KELIPS DHT

The goal is similar to the goal for any DHT: Support **put** and **get**. Kelips does this entirely using gossip!

It ends up being very robust, but a bit slow – we don't really use Kelips, but it does illustrate how gossip can let us build sophisticated data structures that "feel" like things where consensus would normally be used!

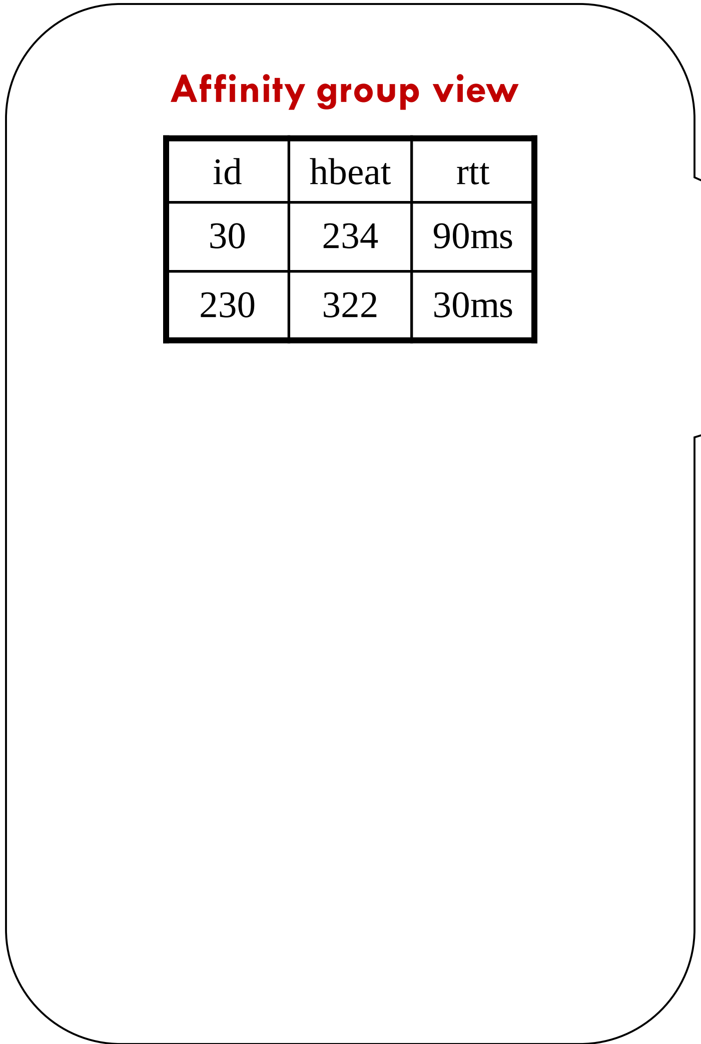
Note: Kelips does not support the "versioned" style of **put**. In fact with gossip, that type of functionality ("compare and swap") is hard!

MAIN ASSUMPTIONS

Kelips was created for wide-area computing.

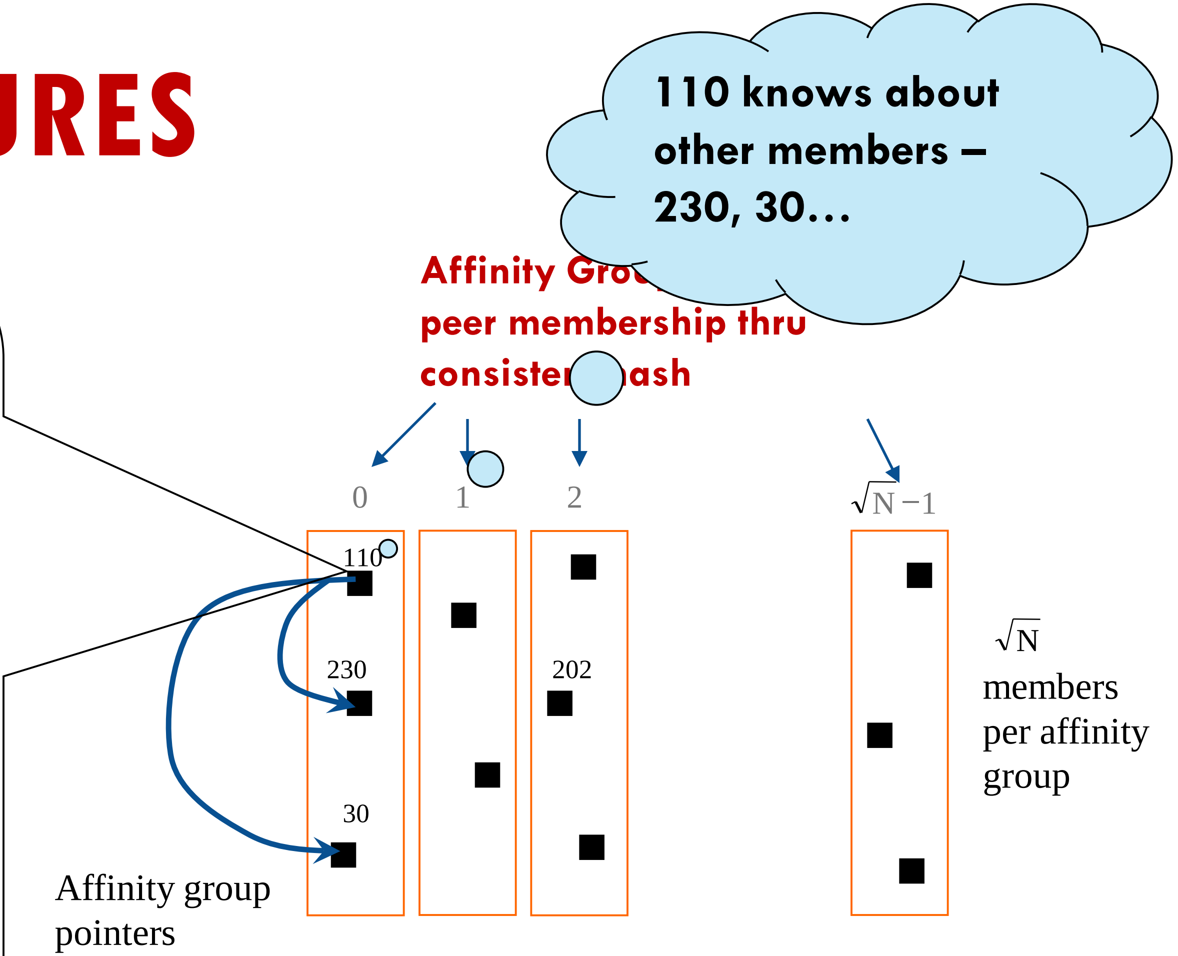
No node is sure who else is running the protocol, although there is a way to send messages to random other nodes.

KELIPS IN PICTURES



Affinity group view

id	hbeat	rtt
30	234	90ms
230	322	30ms



110 knows about other members – 230, 30…

Affinity Gro

peer membership thru

consisten ash

Affinity group pointers

$\sqrt{N}$ members per affinity group

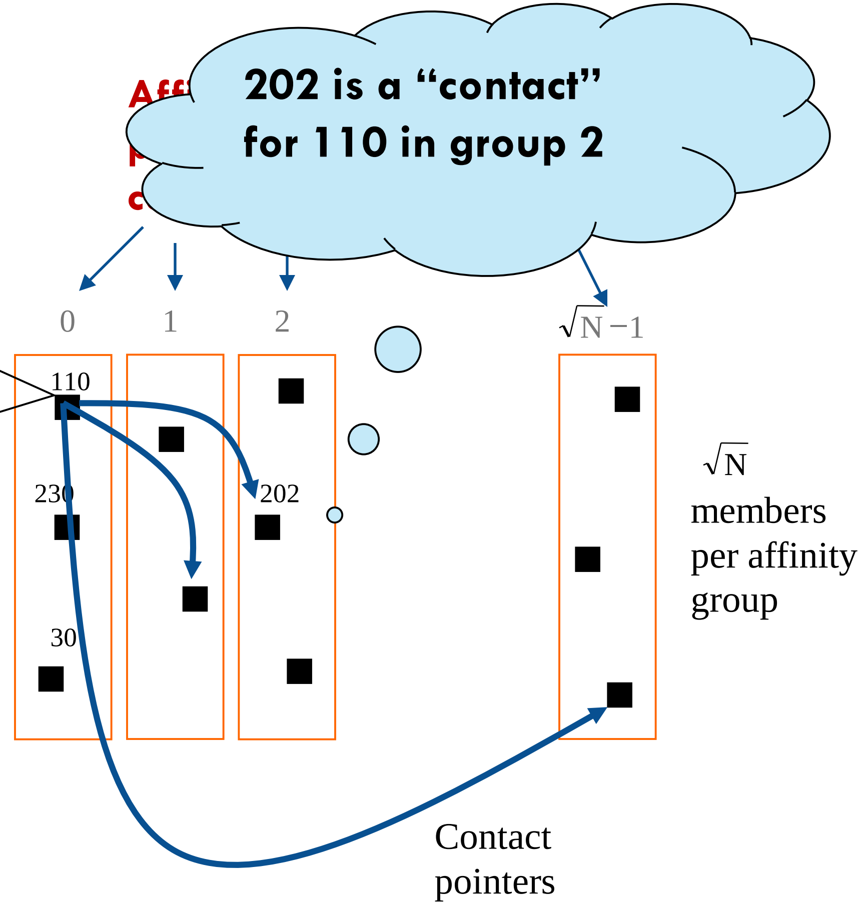
KELIPS

Affinity group view

id	hbeat	rtt
30	234	90ms
230	322	30ms

Contacts

group	contactNode
…	…
2	202


Aff
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

Affinity group view

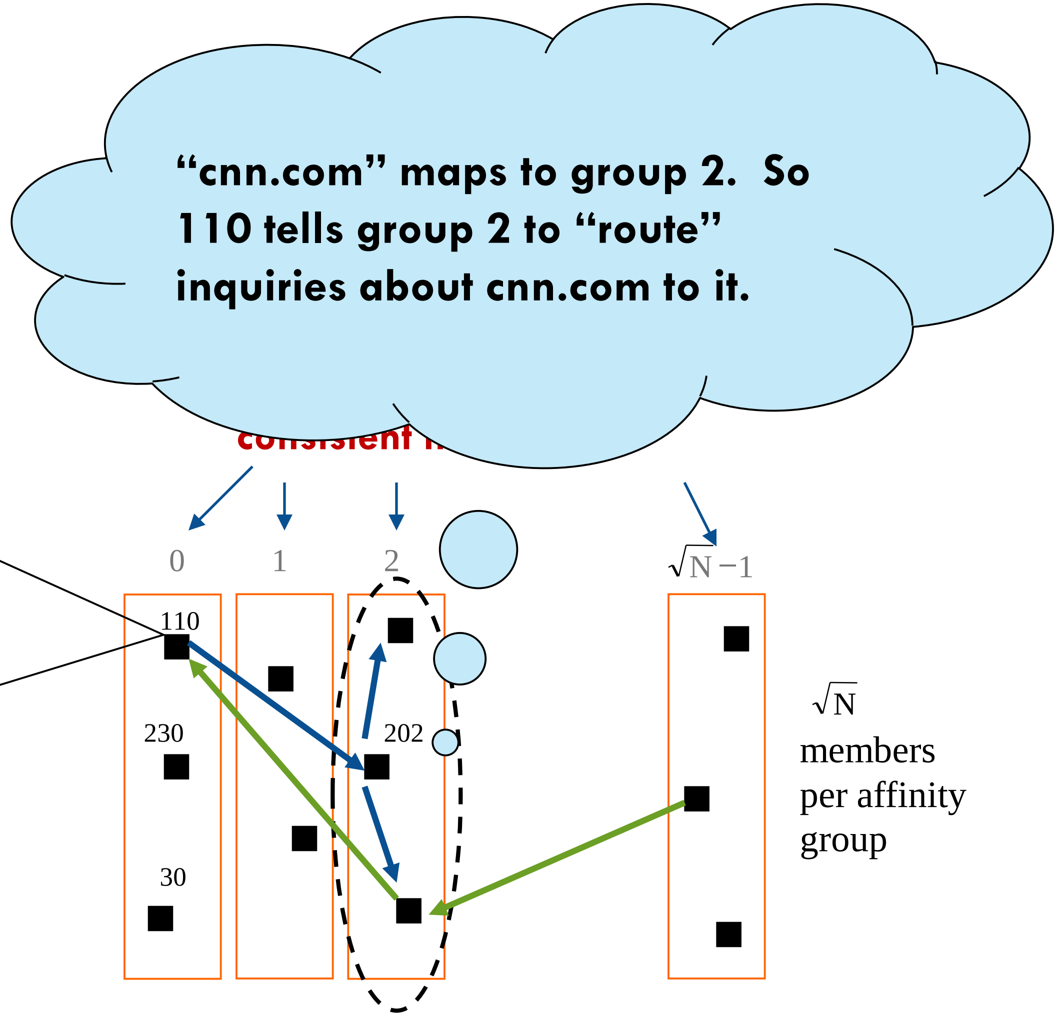
id	hbeat	rtt
30	234	90ms
230	322	30ms

Contacts

group	contactNode
…	…
2	202

Resource Tuples

resource	info
…	…
cnn.com	110



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

$\sqrt{N}$ members per affinity group

Gossip protocol replicates data cheaply

HOW IT WORKS

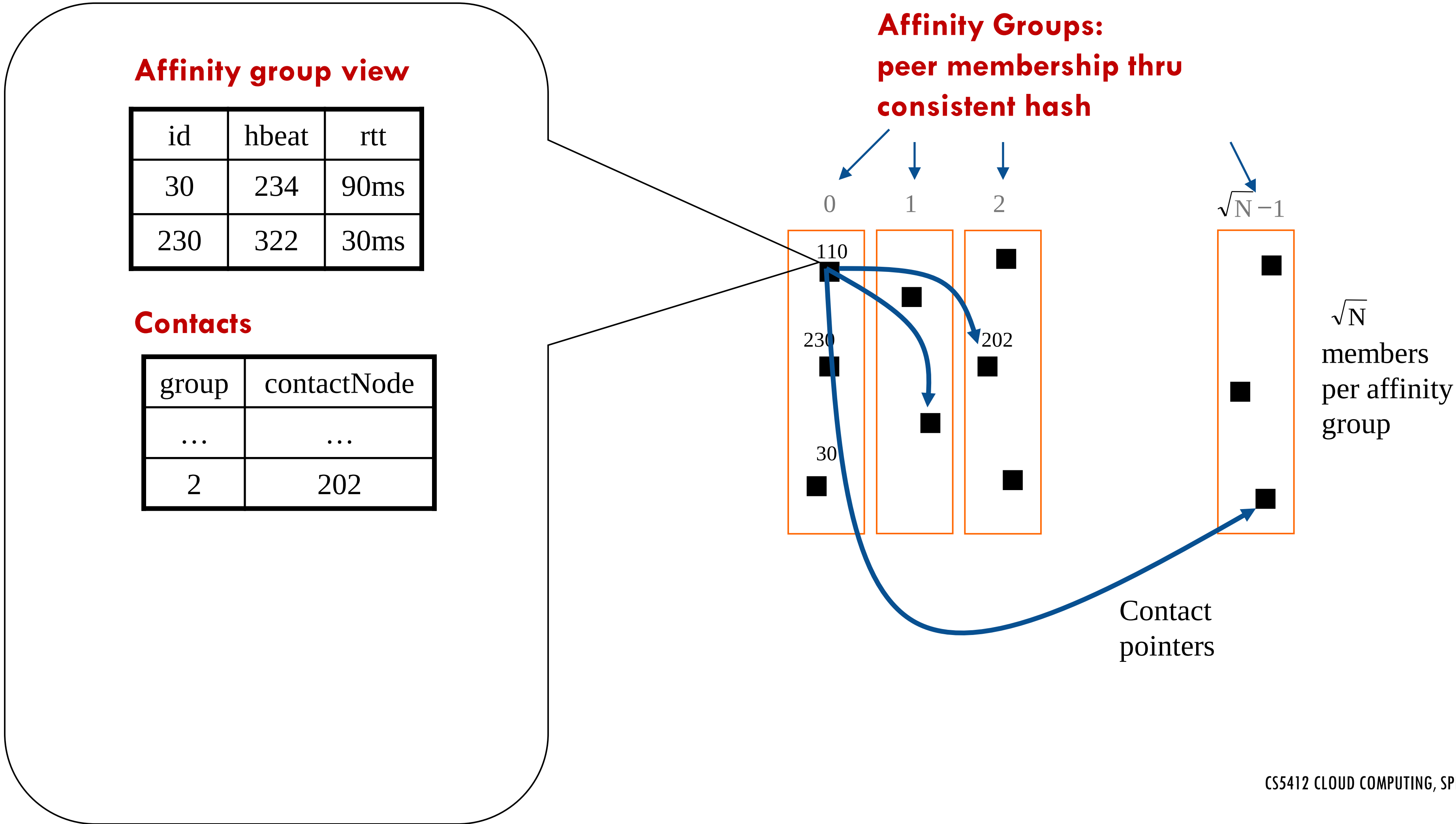
Kelips is *entirely* gossip based!

- 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

SO, HOW DOES IT WORK?



Affinity group view

id	hbeat	rtt
30	234	90ms
230	322	30ms

Contacts

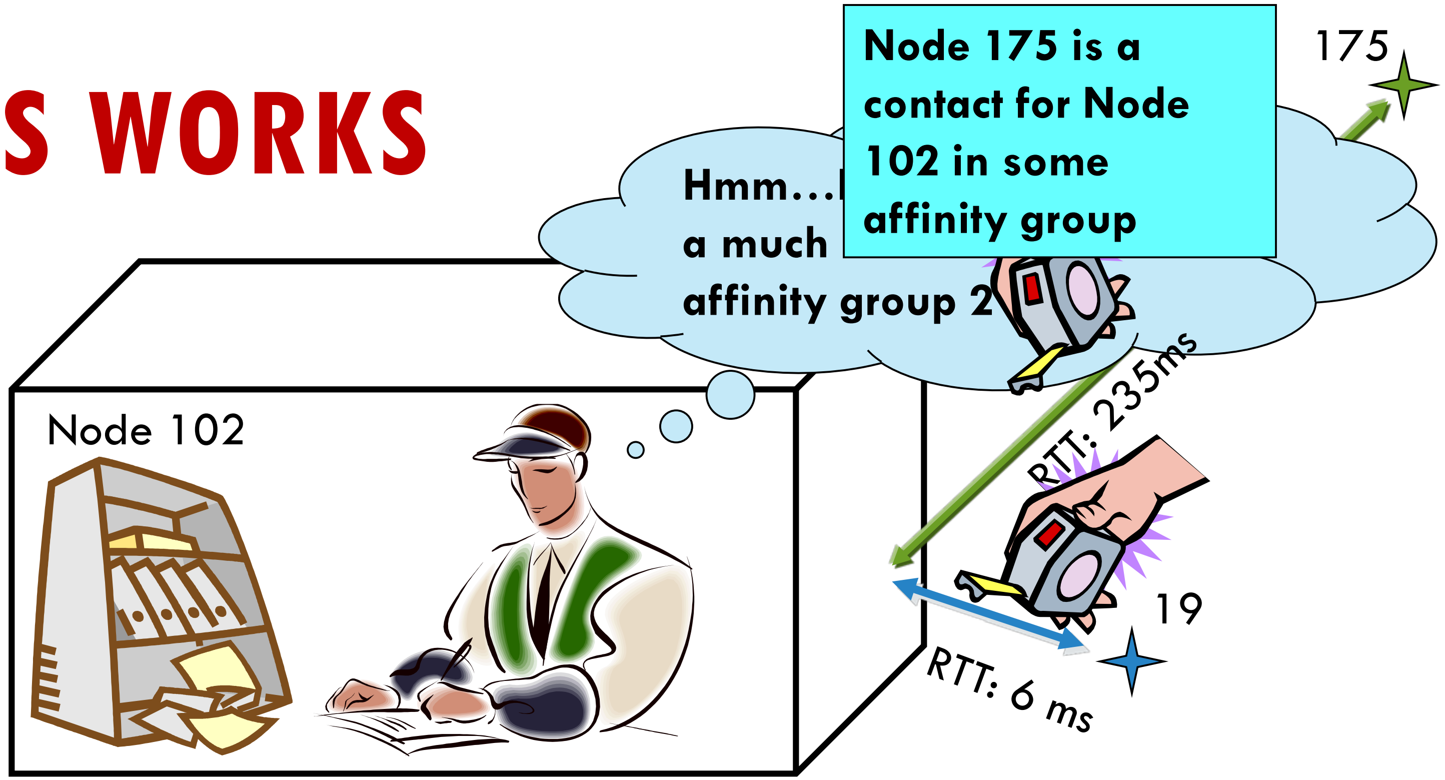
group	contactNode
…	…
2	202

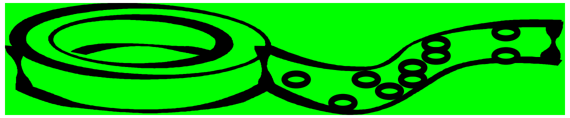
Affinity Groups: peer membership thru consistent hash

$\sqrt{N}$ members per affinity group

Contact pointers

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

INTERESTING THINGS ABOUT KELIPS

The actual DHT is not really “encoded” anywhere.

Instead, Kelips emerges from the mix of gossip and the hashing rule.

If all nodes have the same value of $\sqrt{N}$ then Kelip will be “self-stabilizing”

- They do have the same value for N because they gossip about membership!

WHO USES KELIPS?

Nobody!

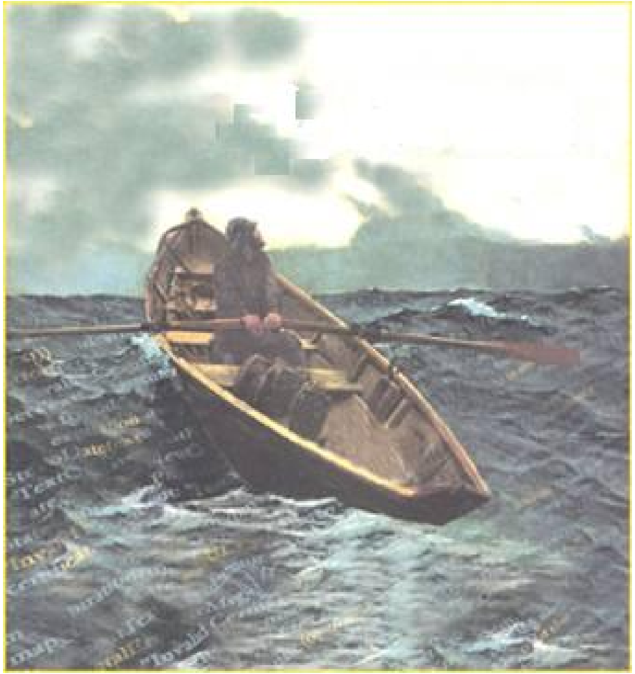
In a datacenter we generally can track membership with perfect accuracy. We can then create a DHT just using hashing.

Kelips only makes sense if nodes don’t know the list of members.

ASTROLABE

Intended as help for applications adrift in a sea of information

Structure emerges from a randomized gossip protocol



An actual Astrolabe

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

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

swift.cs.cornell.edu

Name	Time	Load	Weblogic?	SMTP?	Word Version
swift	2011	2.0	0	1	6.2
falcon	1971	1.5	1	0	4.1
cardinal	2004	4.5	1	0	6.0

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	2201	3.5	1	1	6.0

STATE MERGE: CORE OF ASTROLABE EPIDEMIC

swift.cs.cornell.edu

Name	Time	Load	Weblogic?	SMTP?	Word Version
swift	2011	2.0	0	1	6.2
falcon	1971	1.5	1	0	4.1
cardinal	2004	4.5	1	0	6.0

swift	2011	2.0
cardinal	2201	3.5

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	2201	3.5	1	1	6.0

STATE MERGE: CORE OF ASTROLABE EPIDEMIC

swift.cs.cornell.edu

Name	Time	Load	Weblogic?	SMTP?	Word Version
swift	2011	2.0	0	1	6.2
falcon	1971	1.5	1	0	4.1
cardinal	2201	3.5	1	0	6.0

cardinal.cs.cornell.edu

Name	Time	Load	Weblogic ?	SMTP?	Word Version
swift	2011	2.0	0	1	6.2
falcon	1976	2.7	1	0	4.1
cardinal	2201	3.5	1	1	6.0

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

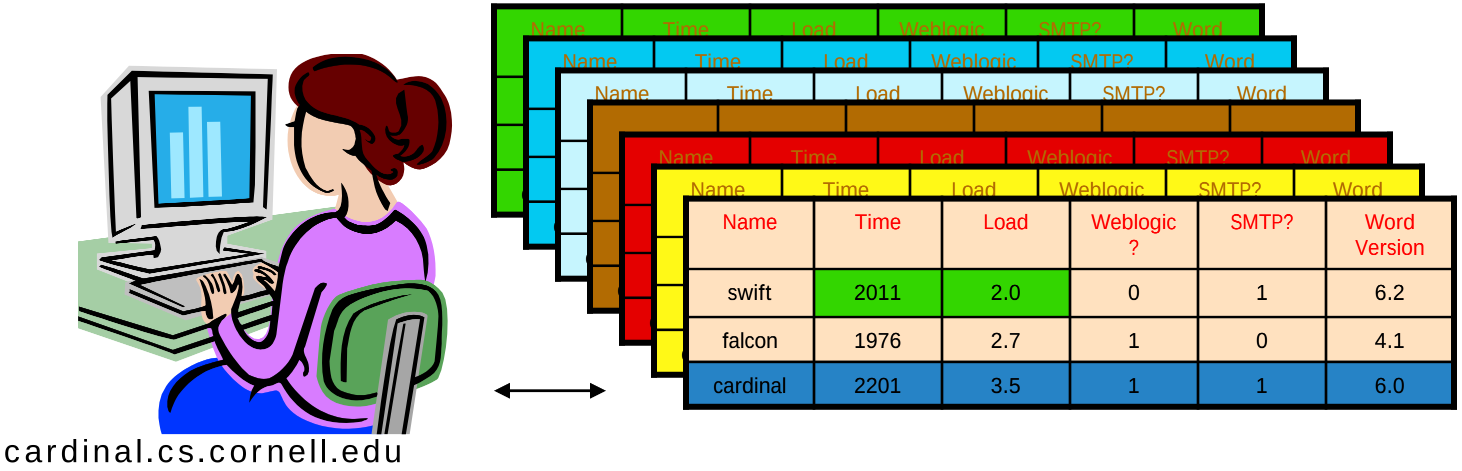
- Looks like a pile of spreadsheets
- A node only replicates data from its neighbors within its own region

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

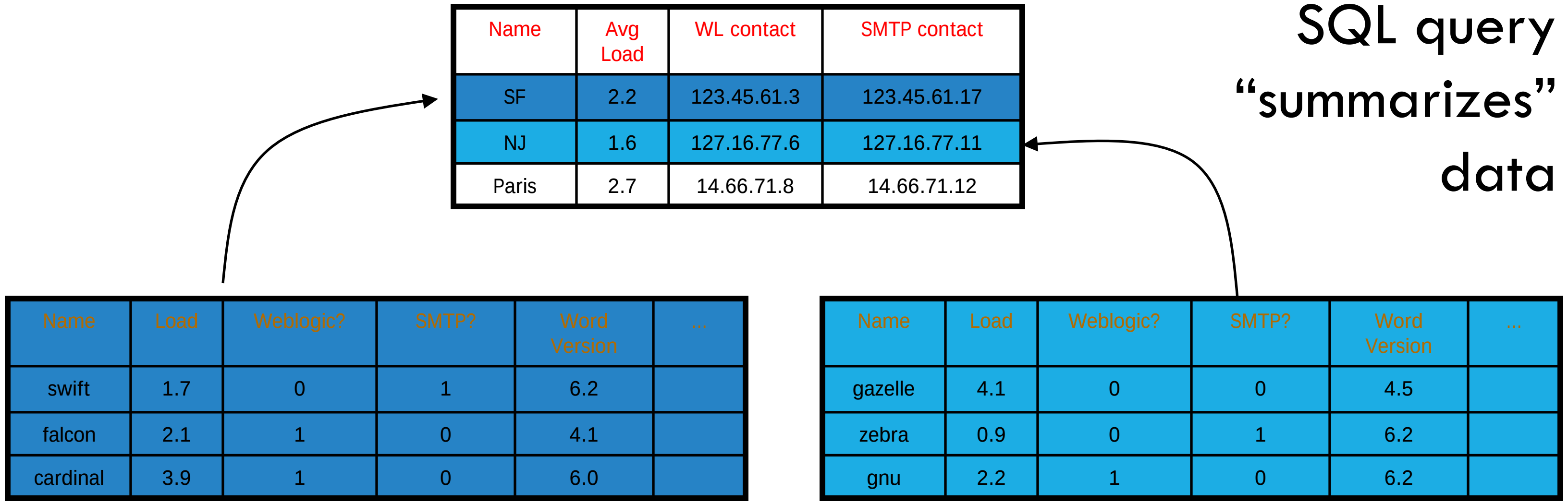


Name	Time	Load	Weblogic ?	SMTP?	Word Version
swift	2011	2.0	0	1	6.2
falcon	1976	2.7	1	0	4.1
cardinal	2201	3.5	1	1	6.0

ASTROLABE BUILDS A HIERARCHY WITHOUT ANY SERVERS

Dynamically changing query output is visible system-wide

SQL query "summarizes" data



Name	Avg Load	WL contact	SMTP contact
SF	2.2	123.45.61.3	123.45.61.17
NJ	1.6	127.16.77.6	127.16.77.11
Paris	2.7	14.66.71.8	14.66.71.12

Name	Load	Weblogic?	SMTP?	Word Version	…
swift	1.7	0	1	6.2	
falcon	2.1	1	0	4.1	
cardinal	3.9	1	0	6.0	

San Francisco

Name	Load	Weblogic?	SMTP?	Word Version	…
gazelle	4.1	0	0	4.5	
zebra	0.9	0	1	6.2	
gnu	2.2	1	0	6.2	

New Jersey

LARGE SCALE: "VIRTUAL" REGIONS

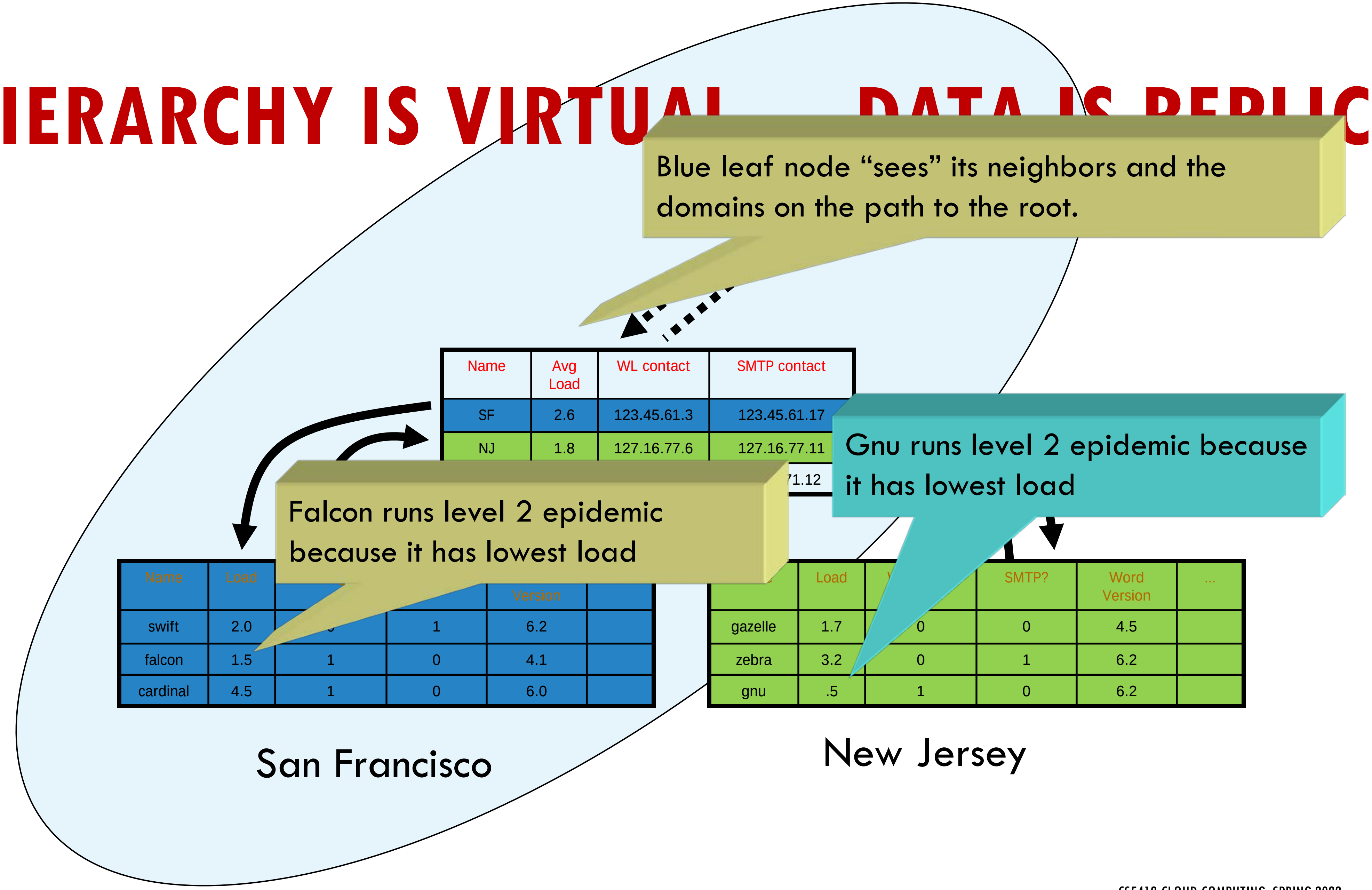
These are

- 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 DATA IS REPLICATED



Blue leaf node “sees” its neighbors and the domains on the path to the root.

Name	Avg Load	WL contact	SMTP contact
SF	2.6	123.45.61.3	123.45.61.17
NJ	1.8	127.16.77.6	127.16.77.11
			71.12

Falcon runs level 2 epidemic because it has lowest load

Gnu runs level 2 epidemic because it has lowest load

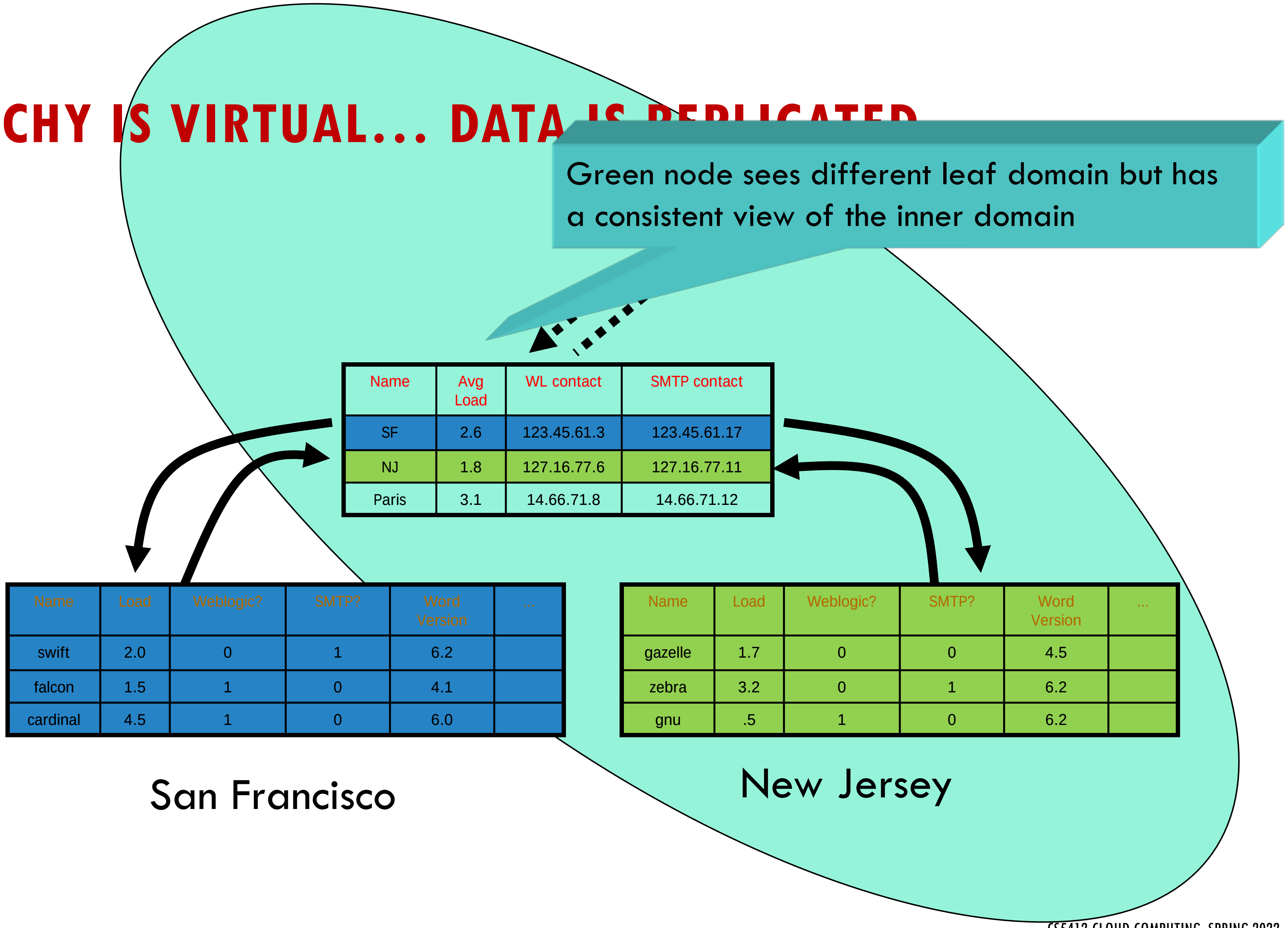
Name	Load			Version	
swift	2.0	0	1	6.2	
falcon	1.5	1	0	4.1	
cardinal	4.5	1	0	6.0	

San Francisco

	Load		SMTP?	Word Version	...
gazelle	1.7	0	0	4.5	
zebra	3.2	0	1	6.2	
gnu	.5	1	0	6.2	

New Jersey

HIERARCHY IS VIRTUAL… DATA IS REPLICATED



Green node sees different leaf domain but has a consistent view of the inner domain

Name	Avg Load	WL contact	SMTP contact
SF	2.6	123.45.61.3	123.45.61.17
NJ	1.8	127.16.77.6	127.16.77.11
Paris	3.1	14.66.71.8	14.66.71.12

Name	Load	Weblogic?	SMTP?	Word Version	…
swift	2.0	0	1	6.2	
falcon	1.5	1	0	4.1	
cardinal	4.5	1	0	6.0	

San Francisco

Name	Load	Weblogic?	SMTP?	Word Version	…
gazelle	1.7	0	0	4.5	
zebra	3.2	0	1	6.2	
gnu	.5	1	0	6.2	

New Jersey

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.

CRITICISM OF ASTROLABE

Uses complained that Astrolabe didn't feel natural.

People expect a database, not a distributed query system. However, they do like the idea of dynamically searching for the root cause of a problem.

Gossip is slow, and management systems may need to react quickly. So if they do use Astrolabe, they might ask for a feature like the bimodal multicast UDP "acceleration", which can speed things up when events occur.

WHO USES ASTROLABE?

When Werner Vogels joined Amazon, they adopted Astrolabe inside the S3 storage system.

It evolved substantially over time, but the gossip pattern was retained.

Today, many management systems use ideas similar to these, but the Astrolabe hierarchical approach is not currently seen even at Amazon.

BLOCKCHAINS

Created as a joke but by now has a $29BUS market value. Pronounced “doge” not “doggy”

Blockchains have emerged as a new application for gossip, but in wide-area settings – not inside datacenters.

The area was pioneered by Bitcoin, the cryptocurrency.

Bitcoin is defined over an append-only tamperproof log (like Paxos!). A gossip protocol is used to share proposed updates robustly.

… WE WON'T DIVE INTO BLOCKCHAIN TODAY

The Bitcoin log is a bit complicated because of the cryptographic model.

But the policy for disseminating updates (log appends) is definitely a gossip protocol. Every node talks to some neighbors in the Bitcoin network "overlay" and they exchange data using gossip techniques.

The idea is that even if a few nodes are Byzantine, the overall blockchain can route around their bad behavior.

HOW TO USE GOSSIP IN A PROJECT?

Lonnie Princehouse

I really like Lonnie Princehouse's MiCA platform.

https://github.com/mica-gossip/MiCA

MiCA uses a Java-based coding style. You create and compose gossip and can even configure them to run at different gossip rates.

SUMMARY

Gossip can be a powerful tool for building stable distributed protocols.

It is extremely easy to use, and robust!

But it can be challenging to create entire systems based on gossip. The technology works best as a tool for building subsystems with specific roles.