Algorithms for Cloud Computing

Nokia Wrocław, Poland

Miotacze Piorunów Team:
Sławek Zborowski, Marek Zdonek, Andrzej Albin, Bartek 'BaSz' Szurgot

December 29, 2015

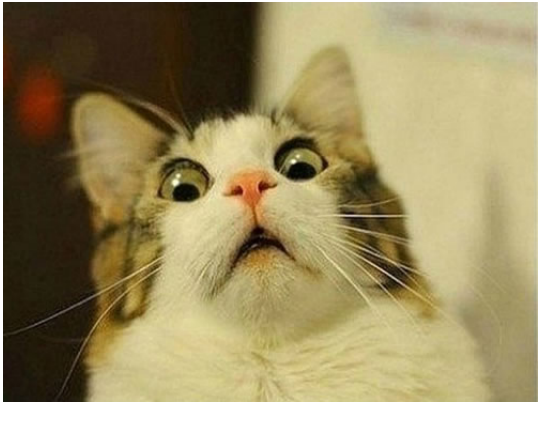
Miotacze Piorunów Team :-)

Miotacze Piorunów - the flag :-)

Roadmap

http://dailynewsdig.com/wp-content/uploads/2013/06/20-Funny-Shocked-Cat-Memes-3.jpg

Roadmap



http://dailynewsdig.com/wp-content/uploads/2013/06/20-Funny-Shocked-Cat-Memes-3.jpg

Cloud blitzkrieg! :-)

http://upload.wikimedia.org/wikipedia/commons/1/10/Bundesarchiv_Bild_101I-646-5188-17%2C_Flugzeuge_Junkers_Ju_87.jpg

Part 1

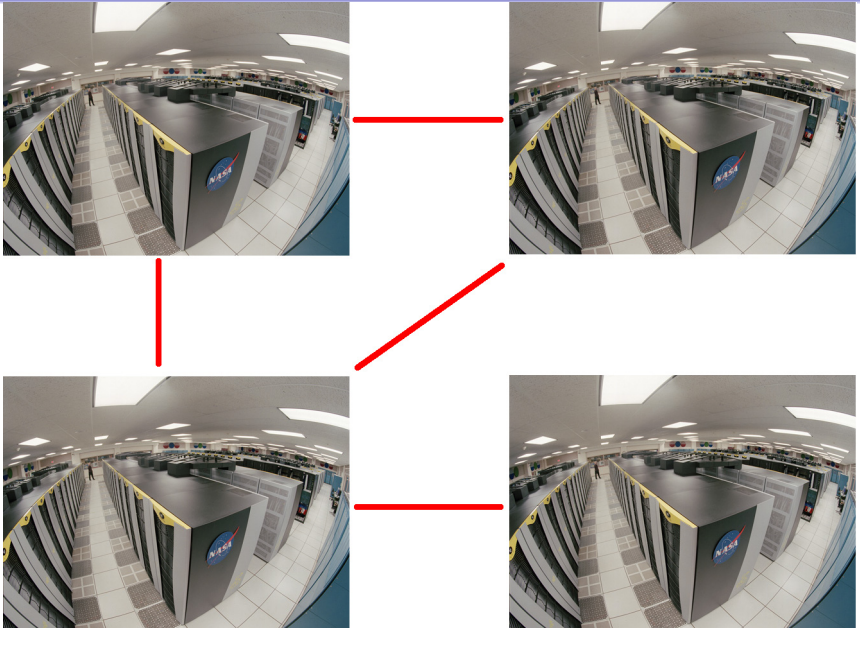
Supercomputers - ’60-’80

http://upload.wikimedia.org/wikipedia/commons/c/c4/CDC_6600.jc.jpg

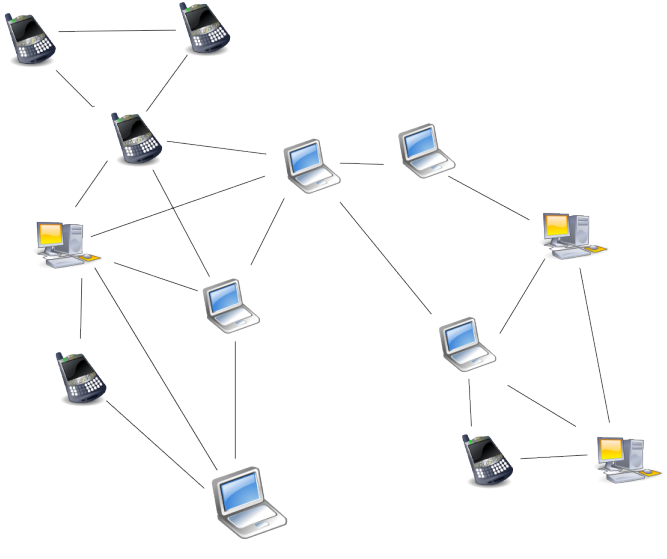
Clusters - since ’70

http://upload.wikimedia.org/wikipedia/commons/3/3d/Us-nasa-columbia.jpg

Grids - since ’90



P2P - since end of ’90



http://upload.wikimedia.org/wikipedia/en/f/fa/Unstructured_peer-to-peer_network_diagram.png

Clouds - since ’00

What is cloud?

- No single definition
  - A bit grid-like
  - With massive storage
  - CPUs + data put close

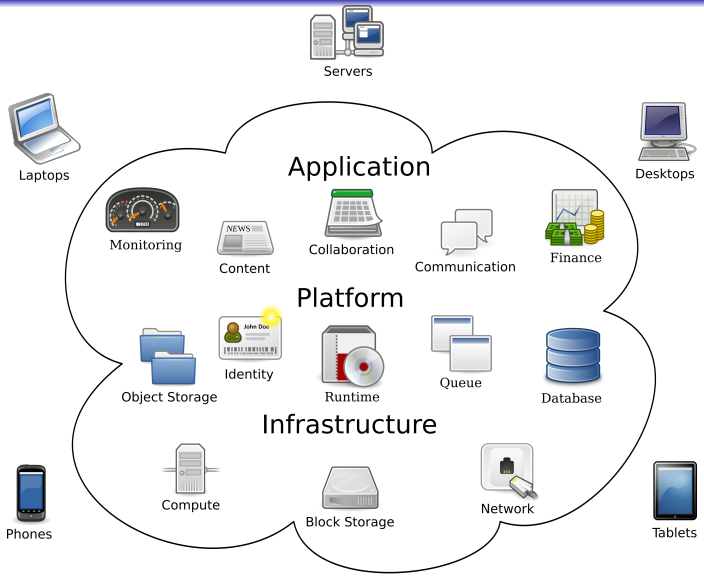
What is cloud?

- No single definition
  - A bit grid-like
  - With massive storage
  - CPUs + data put close
- Commonly agreed
  - Data intensive
  - Lot of CPU power

What is cloud?

- No single definition
  - A bit grid-like
  - With massive storage
  - CPUs + data put close
- Commonly agreed
  - Data intensive
  - Lot of CPU power
  - **S C A L E!**

Cloud types



http://upload.wikimedia.org/wikipedia/commons/b/b5/Cloud_computing.svg

What makes it different?

- Scale bring problems. . .

What makes it different?

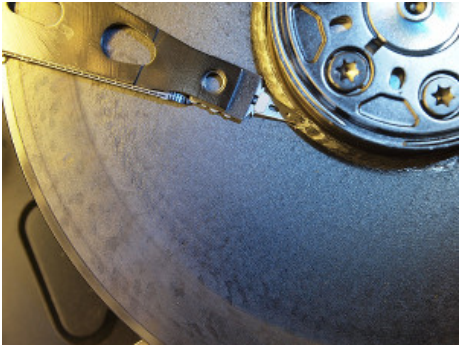
- Scale bring problems...
- Mean Time Between Failures (MTBF):
  - Server: 6 years
  - Disk: 4 years
- Good results!

What makes it different?

- Scale bring problems...
- Mean Time Between Failures (MTBF):
  - Server: 6 years
  - Disk: 4 years
- Good results!
- 10k servers (2 disk each)
- Every day:
  - 5 servers die
  - 14 disks fail

What makes it different?

- Scale bring problems. . .
- Mean Time Between Failures (MTBF):
  - Server: 6 years
  - Disk: 4 years
- Good results!
- $10k$ servers (2 disk each)
- Every day:
  - 5 servers die
  - 14 disks fail



Lesson learned

At scale incredibly rare is a commonplace.

http://upload.wikimedia.org/wikipedia/commons/b/b9/Hard_disk_failure.jpeg

How to deal with it?

- Errors **do happen**
  - Any time
  - Any place

How to deal with it?

- Errors **do happen**
  - Any time
  - Any place
- Massive scale
  - Lots of communication
  - Unreliable networks
  - Data intensive
  - ...

How to deal with it?

- Errors **do happen**
  - Any time
  - Any place
- Massive scale
  - Lots of communication
  - Unreliable networks
  - Data intensive
  - . . .
- Addressed by software
- Algorithms

http://upload.wikimedia.org/wikipedia/commons/1/19/Train_wreck_at_Montparnasse_1895.jpg

Part 2

The problem

- Massive scale system
- Node has new data
- How to inform others?

The problem

- Massive scale system
- Node has new data
- How to inform others?
  - One-to-one?

The problem

- Massive scale system
- Node has new data
- How to inform others?
  - One-to-one?
    - Will take ages to propagate!
    - What about packet loss?
    - What about node loss?

The problem

- Massive scale system
- Node has new data
- How to inform others?
  - One-to-one?
    - Will take ages to propagate!
    - What about packet loss?
    - What about node loss?
  - All-to-all?

The problem

- Massive scale system
- Node has new data
- How to inform others?
  - One-to-one?
    - Will take ages to propagate!
    - What about packet loss?
    - What about node loss?
  - All-to-all?
    - $O(N^2)$ messages!
    - Sending will kill any network!
    - Receiving will kill every node!

The problem

- Massive scale system
- Node has new data
- How to inform others?
  - One-to-one?
    - Will take ages to propagate!
    - What about packet loss?
    - What about node loss?
  - All-to-all?
    - $O(N^2)$ messages!
    - Sending will kill any network!
    - Receiving will kill every node!
- We can do better than that...

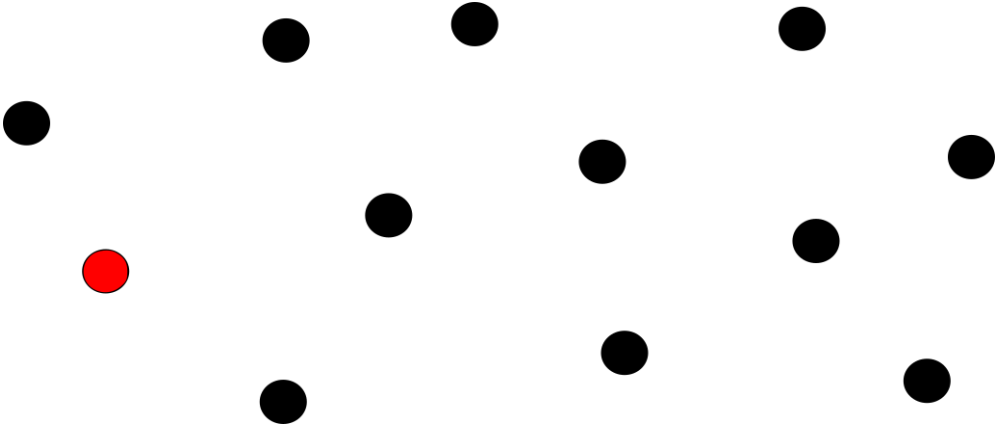
The solution

1. Each node:
    1. Pick random node
    2. Send update to it
    3. Receive incoming messages / wait
    4. Repeat periodically

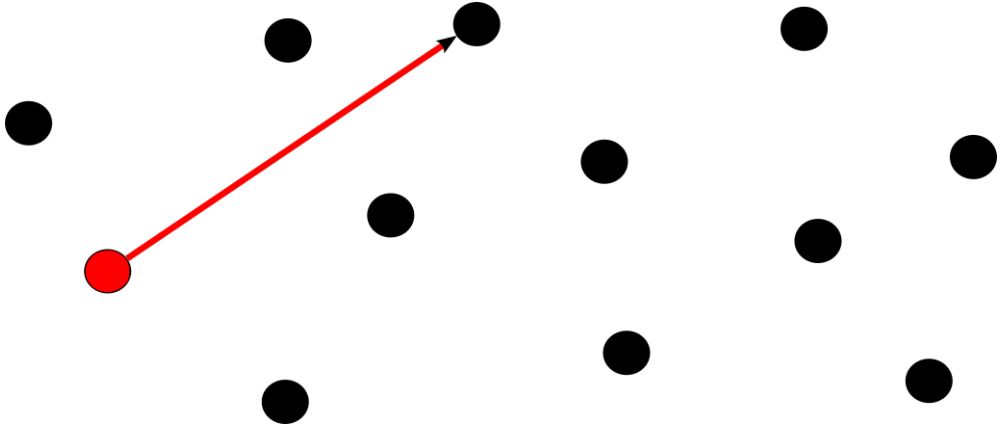
The solution

1. Each node:
    1. Pick random node
    2. Send update to it
    3. Receive incoming messages / wait
    4. Repeat periodically
2. Run on all nodes

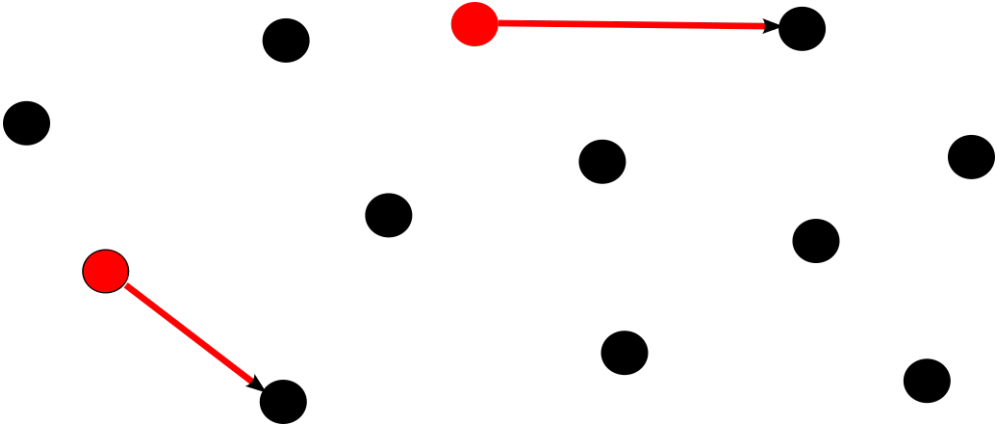
Example run



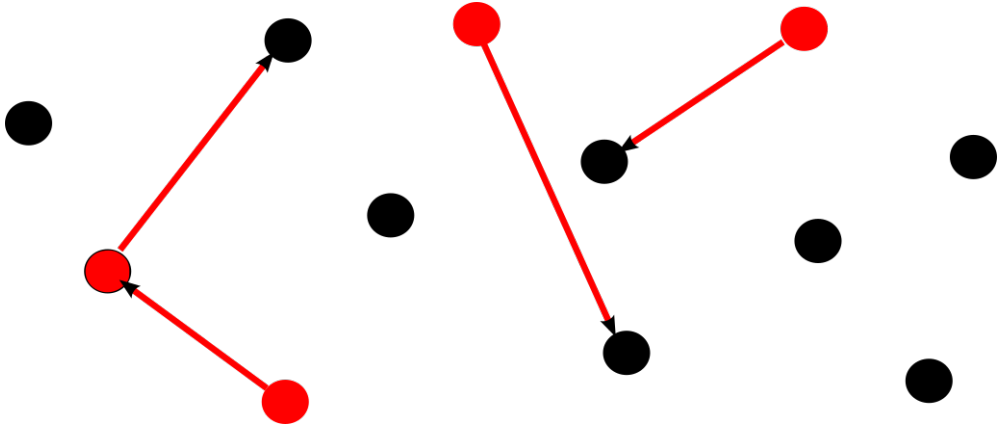
Example run



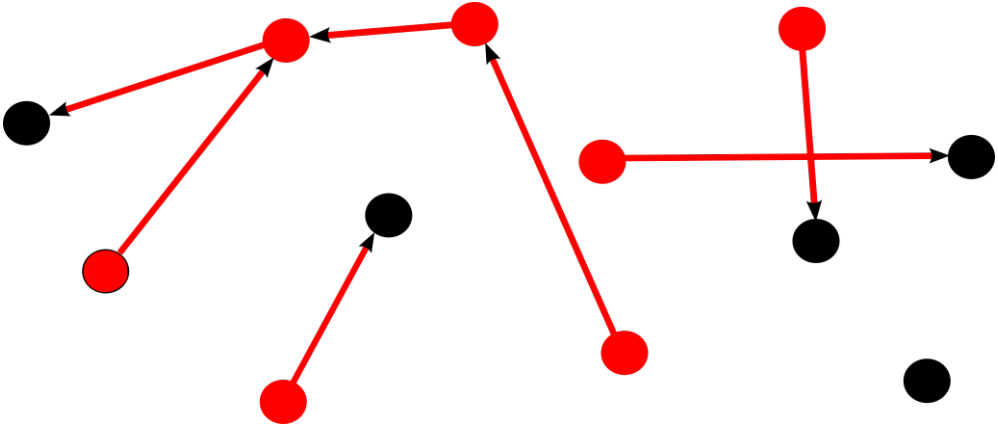
Example run



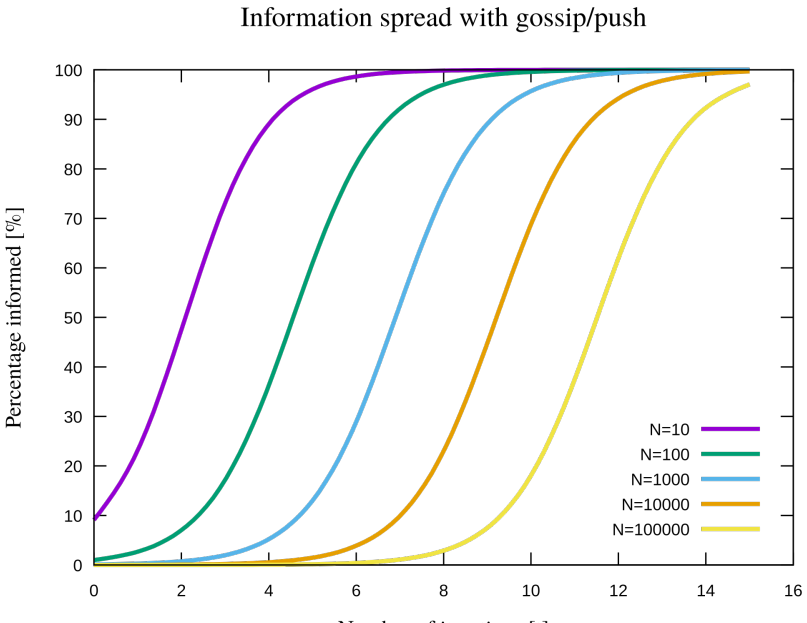
Example run



Example run



Spreads like a disease!



Gossip at Amazon S3

- Amazon S3:
  - **S**imple **S**torage **S**ervice
  - Online file storage
  - Part of *Amazon Web Services*
- Gossiping:
  - Spreading state info
  - Base component of storage
- Some numbers:
  - Over $2 * 10^{12}$ objects (2013)
  - 99.9% availability/month
  - 99.99% availability/year
  - 99.999999999% durability/year

https://upload.wikimedia.org/wikipedia/commons/e/ed/AWS_Simple_Icons_Storage_Amazon_S3_Bucket_with_Objects.svg
https://upload.wikimedia.org/wikipedia/commons/1/1d/AmazonWebservices_Logo.svg

Part 3

Basic concepts

- Aka. "failure detection"

Basic concepts

- Aka. "failure detection"
- Properties:
  - **Accuracy** == no mistakes during judgment
  - **Completeness** == every failure is detected

Basic concepts

- Aka. "failure detection"
- Properties:
  - **Accuracy** == no mistakes during judgment
  - **Completeness** == every failure is detected
  - *Speed* == time to first detection
  - *Scale* == distributing load on nodes uniformly

Basic concepts

- Aka. "failure detection"
- Properties:
  - **Accuracy** == no mistakes during judgment
  - **Completeness** == every failure is detected
  - *Speed* == time to first detection
  - *Scale* == distributing load on nodes uniformly
- **Accuracy** vs. **completeness**
- Cannot have both (over unreliable networks)!

The idea

- Inform others:
  - You're alive
  - Others known to be:
    - Alive
    - Dead

The idea

- Inform others:
  - You're alive
  - Others known to be:
    - Alive
    - Dead
- That is:
  - Propagate dead/alive info...
  - Over large number of nodes...

The idea

- Inform others:
  - You're alive
  - Others known to be:
    - Alive
    - Dead
- That is:
  - Propagate dead/alive info...
  - Over large number of nodes...
- Sounds familiar? :-)

Yup!

Local entries table

- Each node keeps table
- Table for node **A**:

1	2	3
A	5	
B	4	-41
C	5	-13

Local entries table

- Each node keeps table
- Table for node **A**:

1	2	3
A	5	
B	4	-41
C	5	-13

- Three columns
  1. Node name
  2. Heartbeat count
  3. Entry timeout

Local entries table

- Each node keeps table
- Table for node **A**:

1	2	3
A	5	
B	4	-41
C	5	-13

- Three columns
  1. Node name
  2. Heartbeat count
  3. Entry timeout
- Timeout is local
- Timed out entries are dead
- Heartbeat incremented on send

Merging entries

A

A	5	
B	4	-41
C	5	-13

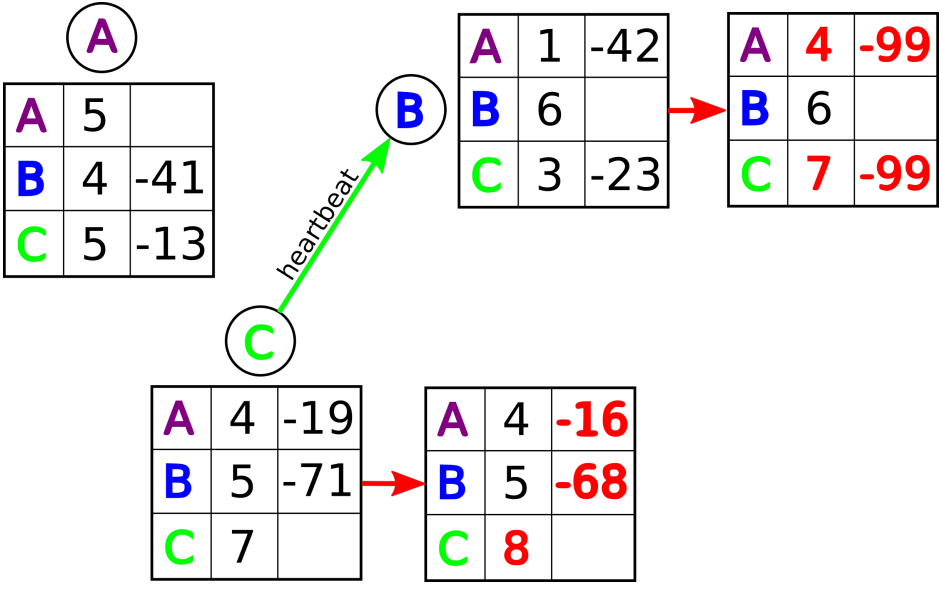
B

A	1	-42
B	6	
C	3	-23

C

A	4	-19
B	5	-71
C	7	

Merging entries



Amazon DynamoDB

- Amazon DynamoDB:
  - Database-like
  - Distributed Hash Table (DHT)
  - Powers Amazon Web Services
  - Used by Amazon S3
- Membership + failure detection
- No central register
- Excellent scaling

https://upload.wikimedia.org/wikipedia/commons/e/ed/AWS_Simple_Icons_Storage_Amazon_S3_Bucket_with_Objects.svg
https://upload.wikimedia.org/wikipedia/commons/1/1d/AmazonWebservices_Logo.svg

Part 4

Overview

- **S**calable, **W**eakly-consistent, **I**nfection-style **M**embership protocol
- Pings instead of heartbeats

Overview

- **S**calable, **W**eakly-consistent, **I**nfection-style **M**embership protocol
- Pings instead of heartbeats
- Algorithm:
  1. Ping 1, random host (H)
  2. If got pong == ok (done)

Overview

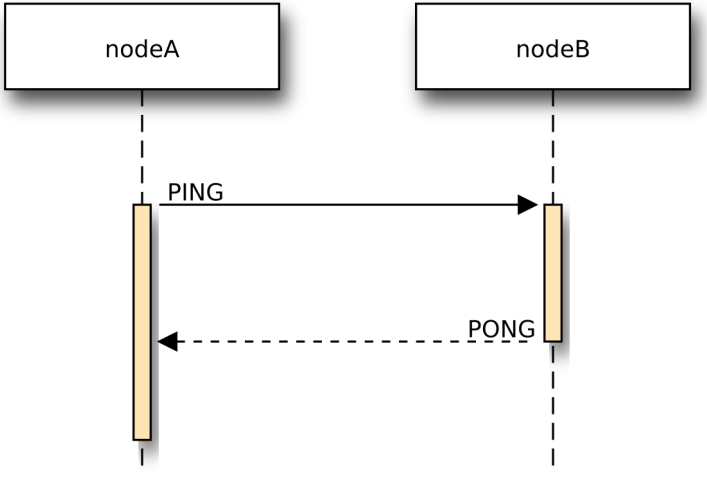
- **S**calable, **W**eakly-consistent, **I**nfection-style **M**embership protocol
- Pings instead of heartbeats
- Algorithm:
  1. Ping 1, random host (H)
  2. If got pong == ok (done)
  3. Else
     1. Ask N, random hosts to ping (H)
     2. 1 "ack" == ok
     3. Else mark as failed

Overview

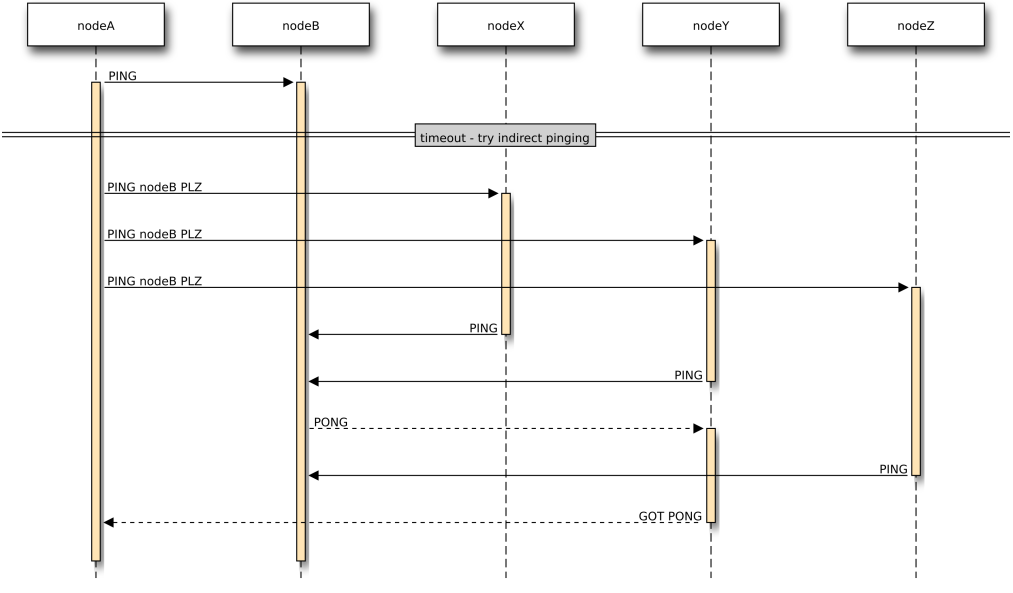
- **S**calable, **W**eakly-consistent, **I**nfection-style **M**embership protocol
- Pings instead of heartbeats
- Algorithm:
  1. Ping 1, random host (H)
  2. If got pong == ok (done)
  3. Else
     1. Ask N, random hosts to ping (H)
     2. 1 "ack" == ok
     3. Else mark as failed
- Extra attempt to verify:
  - Special (more network paths)
  - Temporal (more time to response)

Example - direct ping succeeded

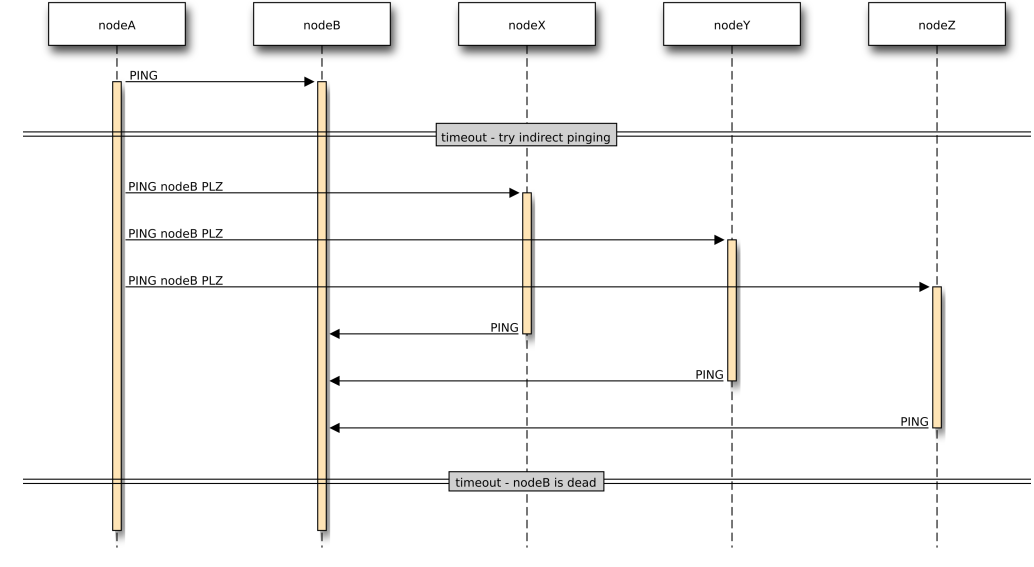
Example - direct ping succeeded



Example - direct ping failed

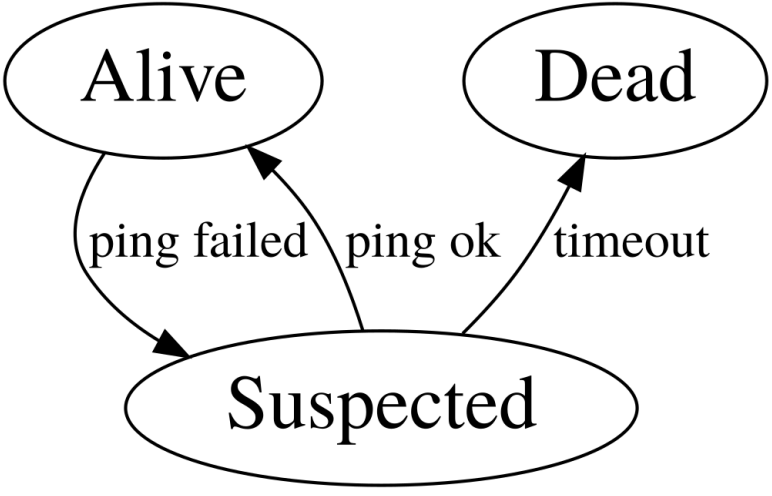


Example - all pings failed



Suspicion mechanism

Suspicion mechanism



Part 5

Overview

- Classic algorithm
- Proposed by Google
- Sourcing from functional languages

Overview

- Classic algorithm
- Proposed by Google
- Sourcing from functional languages
- Two stages:
  - Map
  - Reduce

Overview

- Classic algorithm
- Proposed by Google
- Sourcing from functional languages
- Two stages:
  - Map
  - Reduce
- Mapping:
  - Input: raw source data
  - Output: (key -> value) pairs

Overview

- Classic algorithm
- Proposed by Google
- Sourcing from functional languages
- Two stages:
  - Map
  - Reduce
- Mapping:
  - Input: raw source data
  - Output: (key -> value) pairs
- Reducing:
  - Input: (key -> values[])
  - Output: (key -> merged-values)

Cars example

- Input: database of cars registered, per year

Cars example

- Input: database of cars registered, per year
  - 1991:
    - Opel Vectra, ABS
    - Mercedes W124, ABS, airbag
  - 1995:
    - Opel Vectra, AC, TC
    - Dodge Viper, ABS, airbag
  - ...

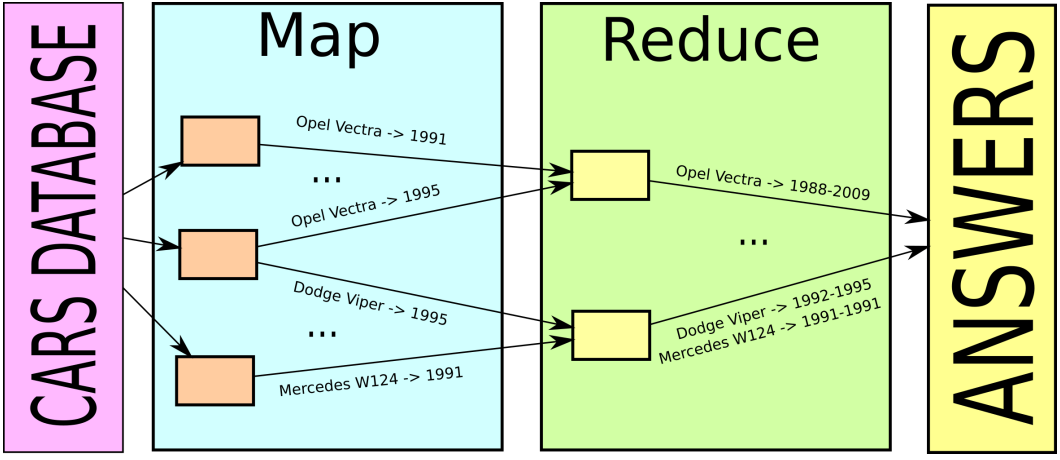
Cars example

- Input: database of cars registered, per year
  - 1991:
    - Opel Vectra, ABS
    - Mercedes W124, ABS, airbag
  - 1995:
    - Opel Vectra, AC, TC
    - Dodge Viper, ABS, airbag
  - ...
- Query: which cars where produced when?

Cars example

- Input: database of cars registered, per year
  - 1991:
    - Opel Vectra, ABS
    - Mercedes W124, ABS, airbag
  - 1995:
    - Opel Vectra, AC, TC
    - Dodge Viper, ABS, airbag
  - ...
- Query: which cars where produced when?
  - Opel Vectra: 1988-2009
  - Dodge Viper: 1992-1995
  - ...

Computations



Algorithm properties

- Mapping:
  - Generation is independent
  - Perfect parallel task

Algorithm properties

- Mapping:
  - Generation is independent
  - Perfect parallel task
- Reducing:
  - Can be run in parallel
  - Good distribution of mapped entries needed!

Algorithm properties

- Mapping:
  - Generation is independent
  - Perfect parallel task
- Reducing:
  - Can be run in parallel
  - Good distribution of mapped entries needed!
- Algorithm:
  - Highly parallel
  - Widely adopted
  - Common solution for batch tasks

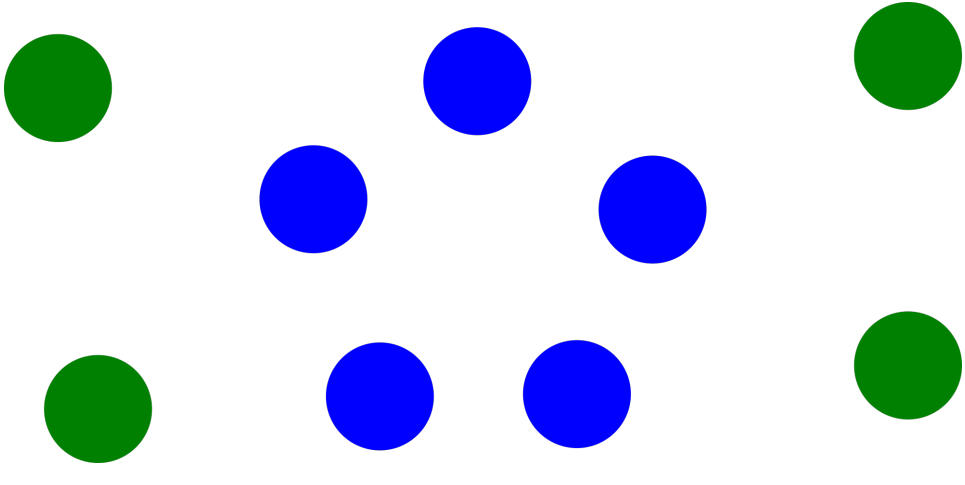
Practical applications

https://upload.wikimedia.org/wikipedia/commons/a/aa/Logo_Google_2013_Official.svg
https://upload.wikimedia.org/wikipedia/commons/0/0e/Hadoop_logo.svg
https://upload.wikimedia.org/wikipedia/en/f/f8/CouchDB.svg

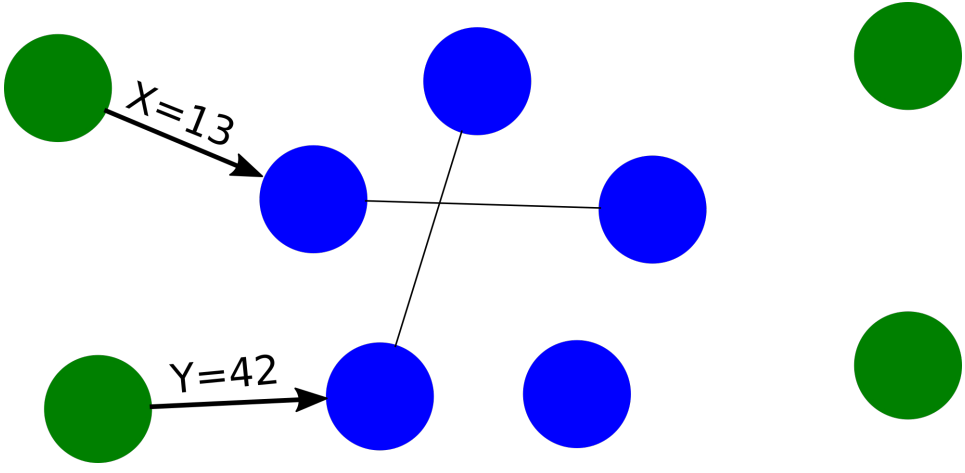
https://upload.wikimedia.org/wikipedia/en/e/eb/MongoDB_Logo.png
https://upload.wikimedia.org/wikipedia/en/8/8e/Riak_distributed_NoSQL_key-value_data_store_logo.png

Part 6

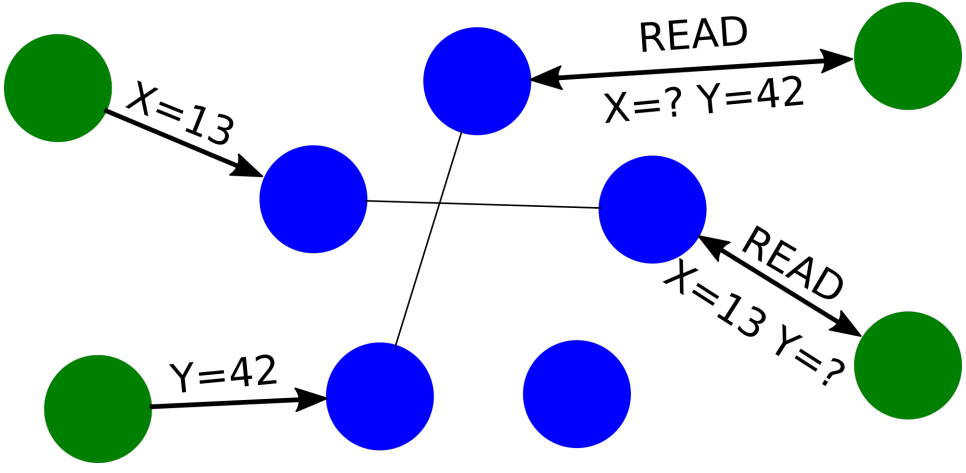
Once upon a time...



Once upon a time...



Once upon a time...



Consensus

Consensus problem

Reaching an agreement upon a single value/state.

- Fundamental problem in distributed systems

Consensus

Consensus problem

Reaching an agreement upon a single value/state.

- Fundamental problem in distributed systems
- How to:
  - Agree upon single value?
  - Tolerate failures?

Consensus

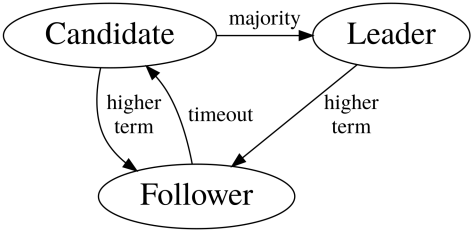
Consensus problem

Reaching an agreement upon a single value/state.

- Fundamental problem in distributed systems
- How to:
  - Agree upon single value?
  - Tolerate failures?
- Example algorithms:
  - Paxos
  - Raft

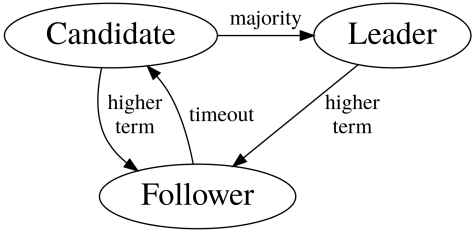
Raft's organization

- Node is in a given state



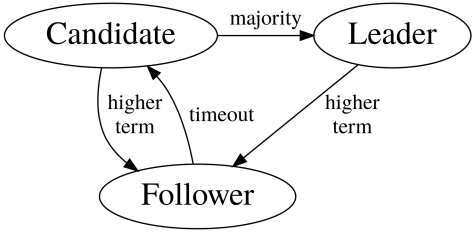
Raft's organization

- Node is in a given state
- Basic operations:
  - Leader election
  - Log replication ("*state machine*")



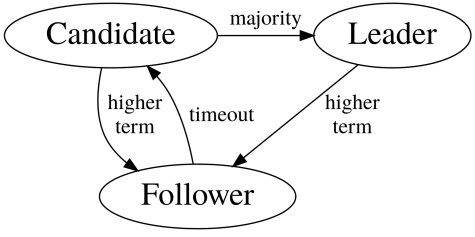
Raft's organization

- Node is in a given state
- Basic operations:
  - Leader election
  - Log replication ("*state machine*")
- Voting for a leader
- "Term counter":
  - Identify voting rounds
  - Incremented by candidate



Raft's organization

- Node is in a given state
- Basic operations:
  - Leader election
  - Log replication ("*state machine*")
- Voting for a leader
- "Term counter":
  - Identify voting rounds
  - Incremented by candidate
- Leader:
  - Send heartbeats
  - Synchronizes followers
- Clients talk with leader

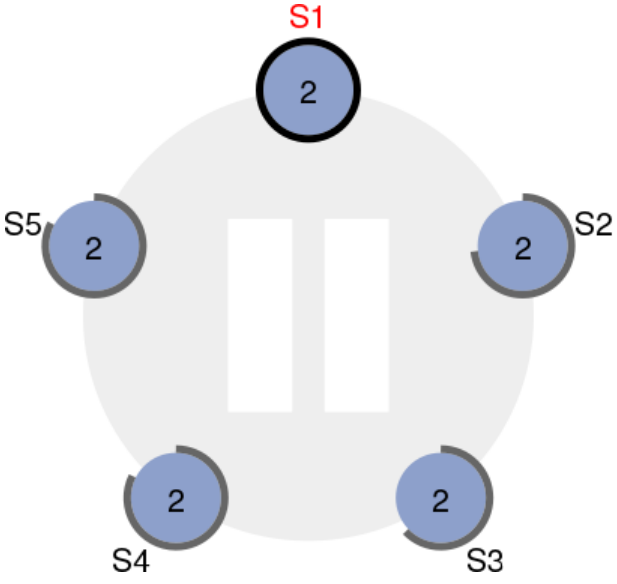


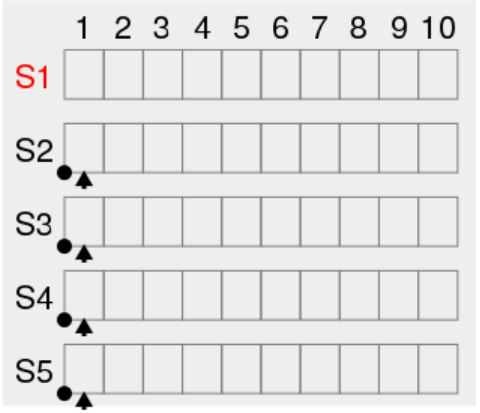
Log replication

https://fiftytwomedia.files.wordpress.com/2011/05/cloned-sheep.jpg

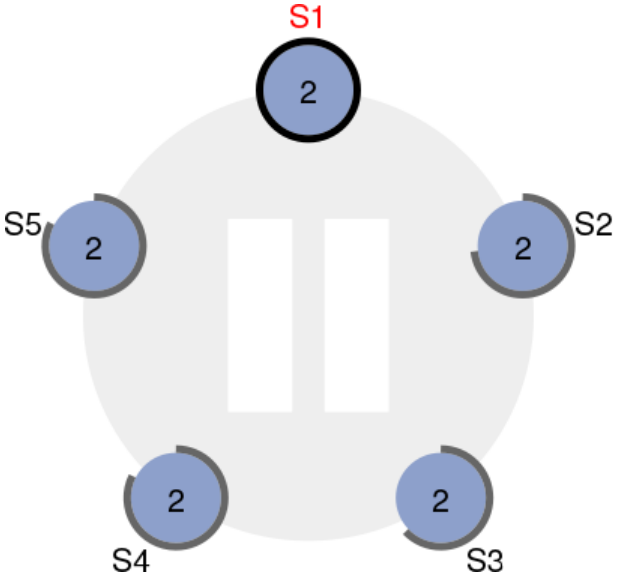
How log (state machine) is replicated?

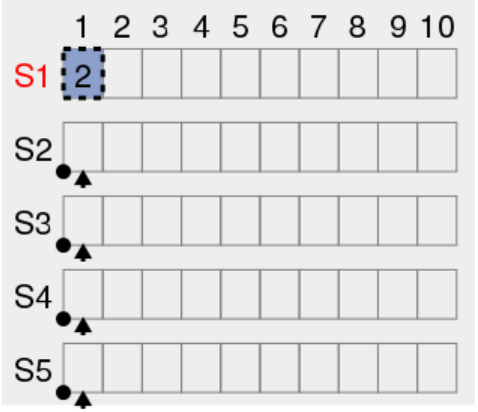
Log replication



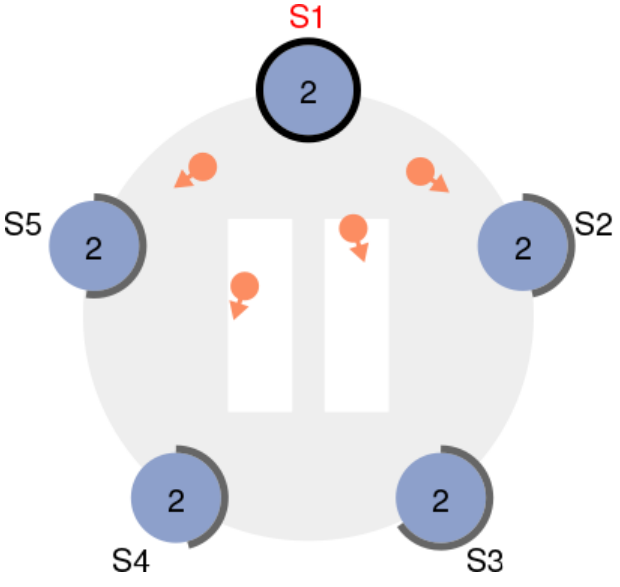


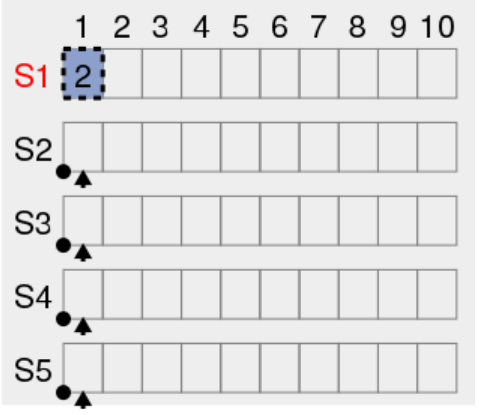
Log replication



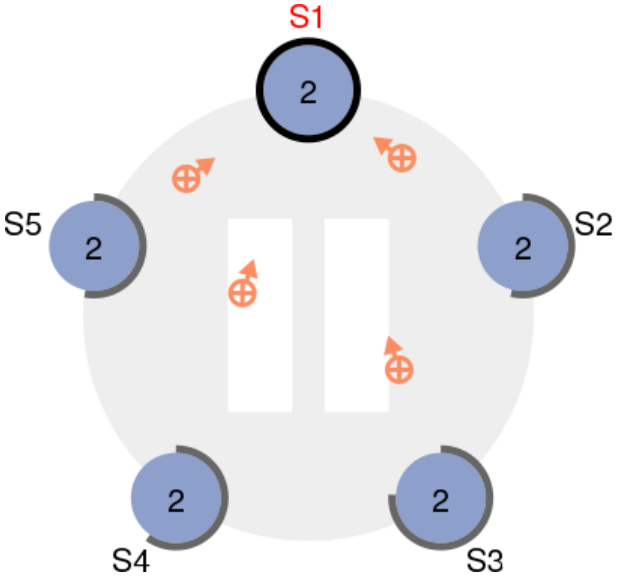


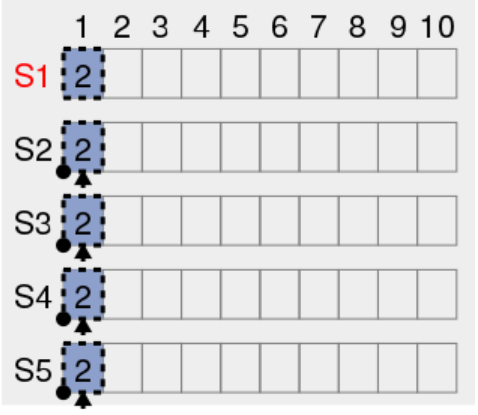
Log replication



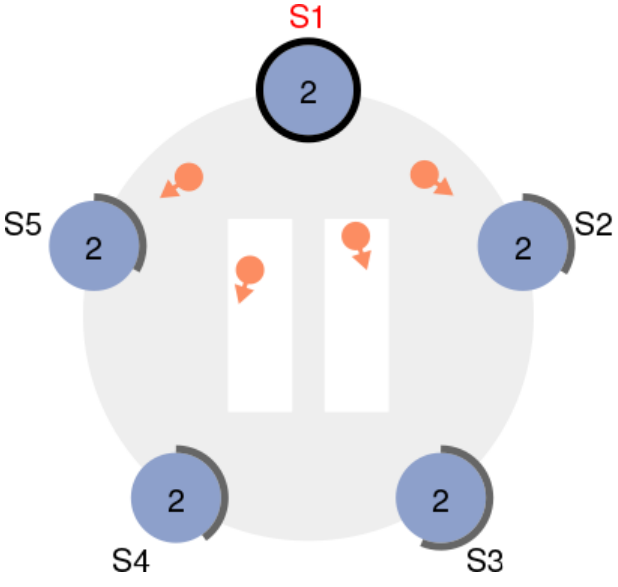


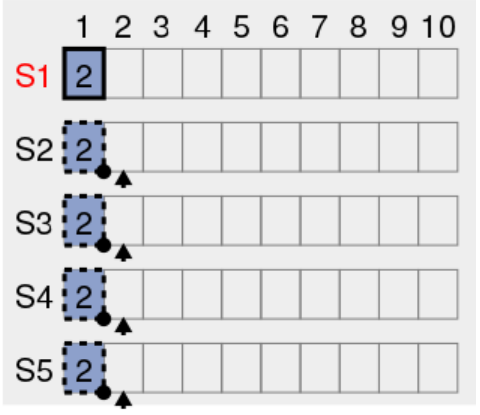
Log replication



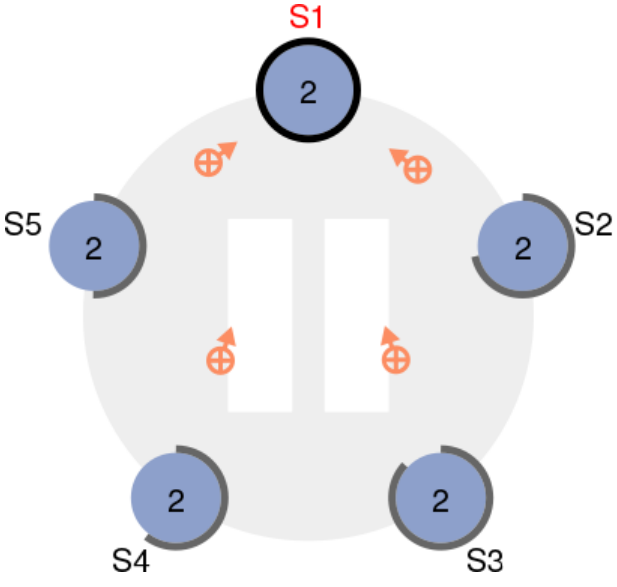


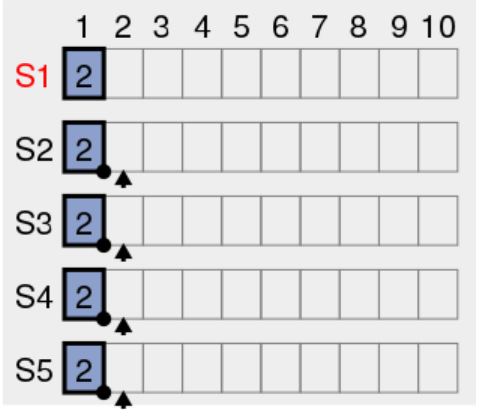
Log replication





Log replication



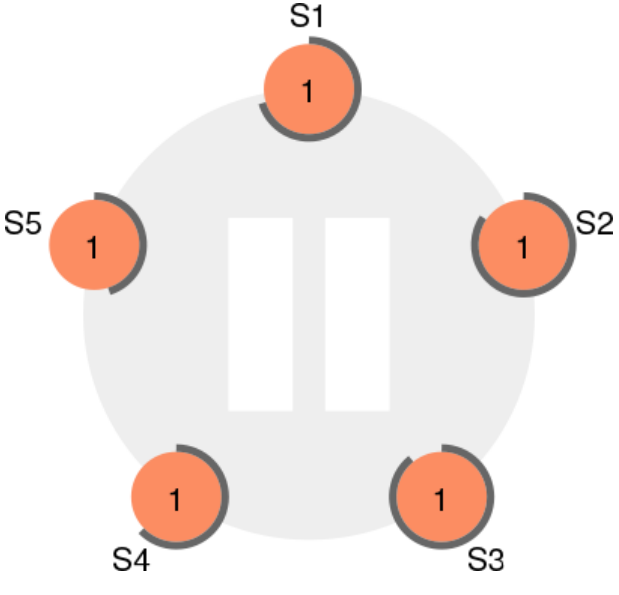


Leader election

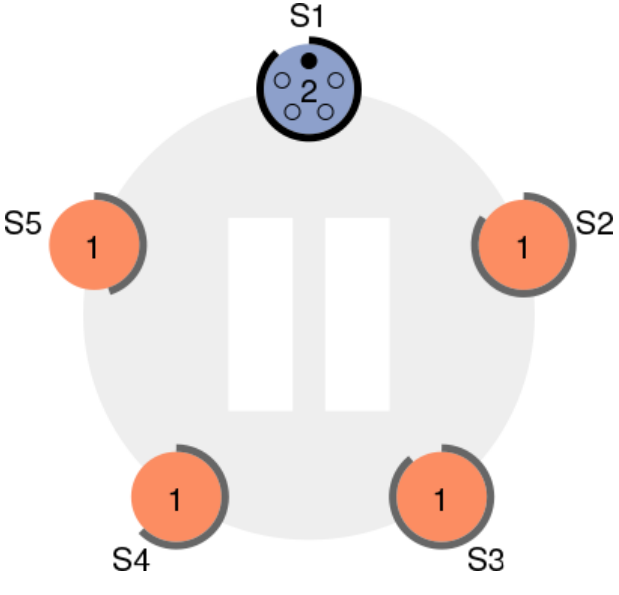
How leader election works?

http://i.huffpost.com/gen/1849024/thumbs/o-ELECTION-BALLOT-facebook.jpg

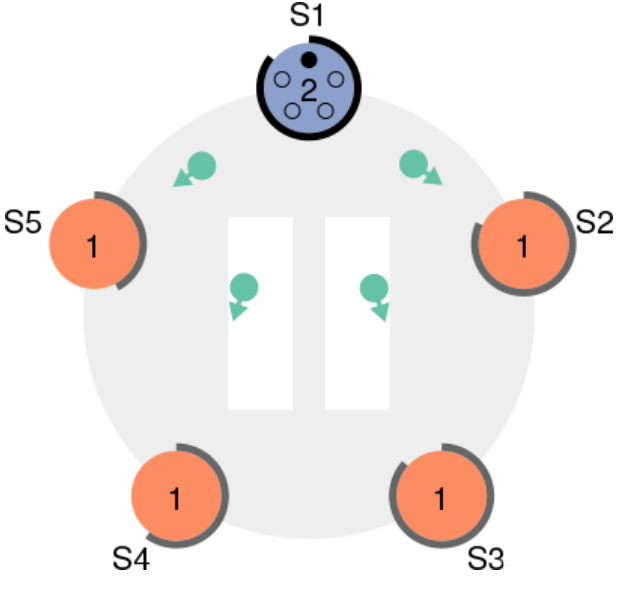
Leader election



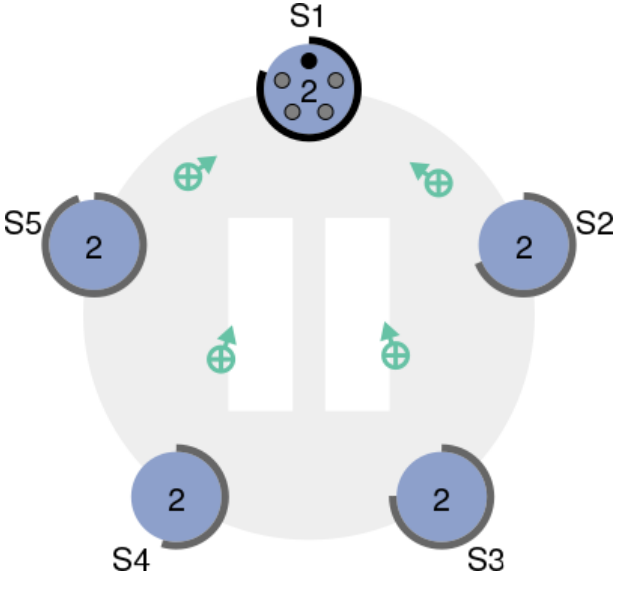
Leader election



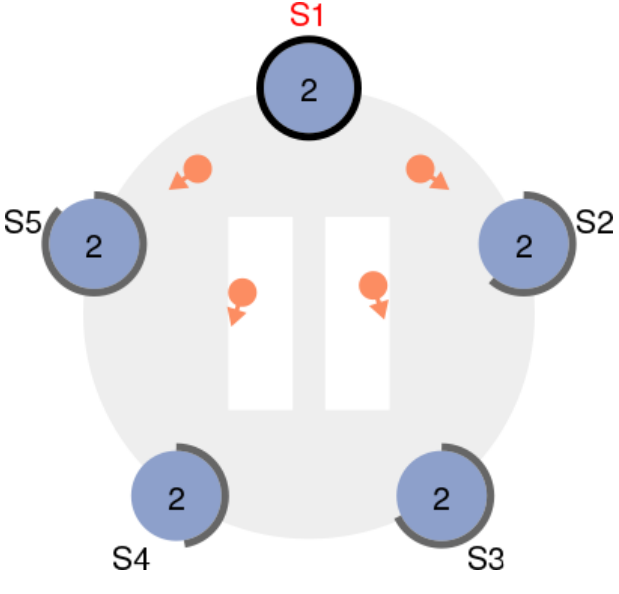
Leader election



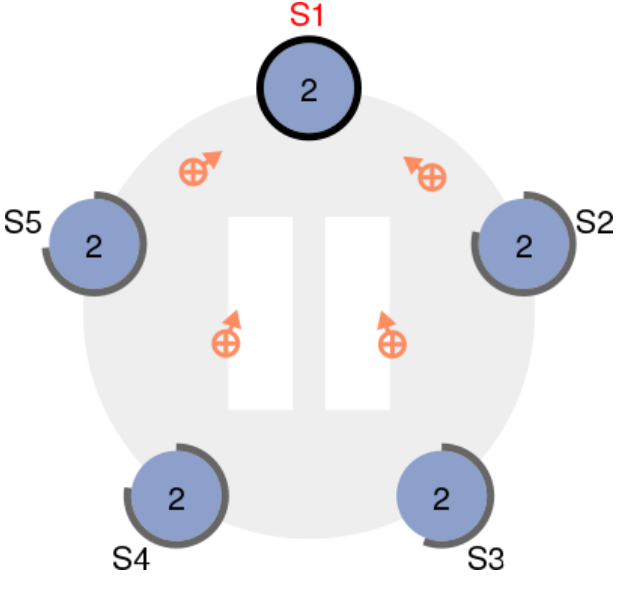
Leader election



Leader election



Leader election

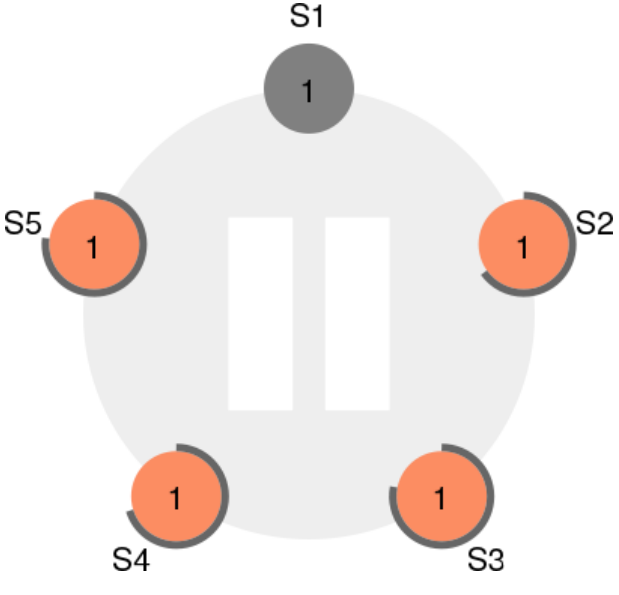


That simple?

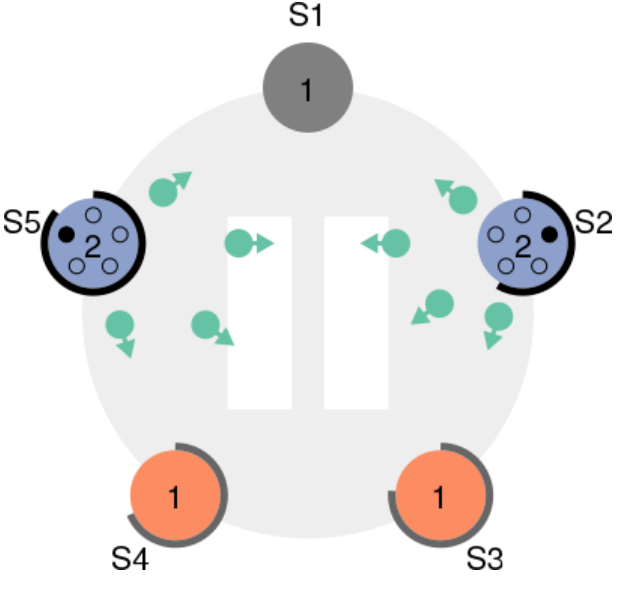
http://cache.gawkerassets.com/assets/images/4/2010/01/500x_hamshred.jpg

Does leader election always work?

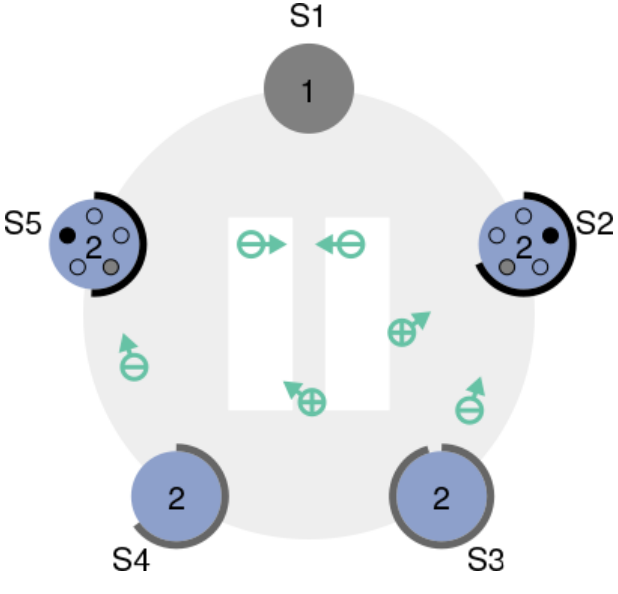
Split vote



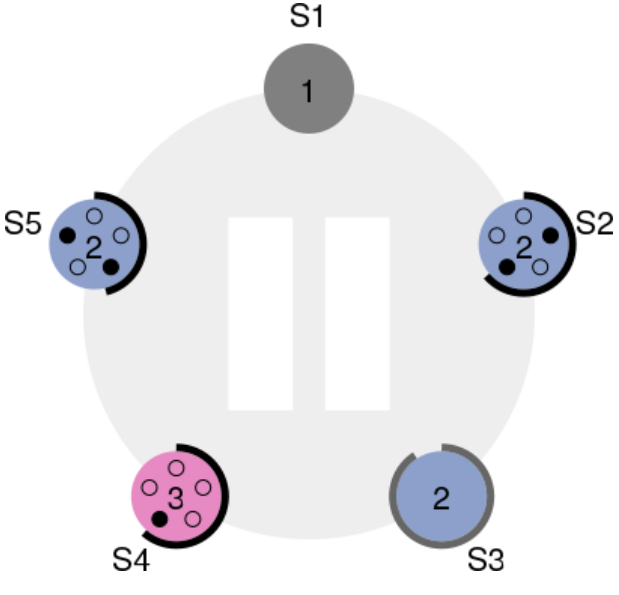
Split vote



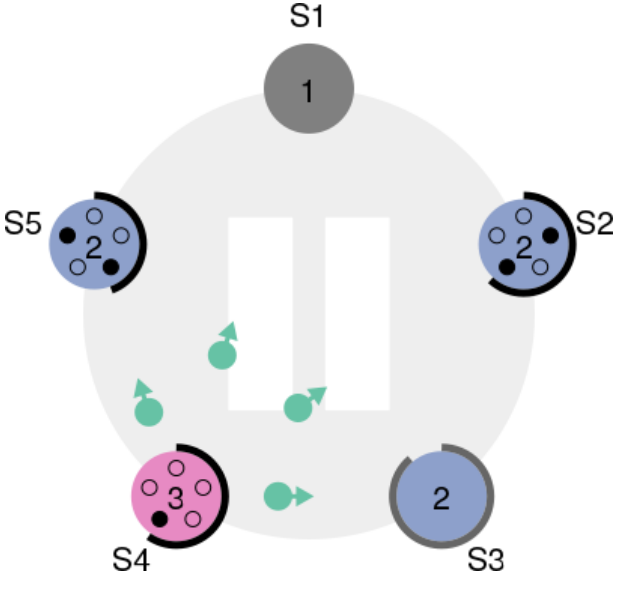
Split vote



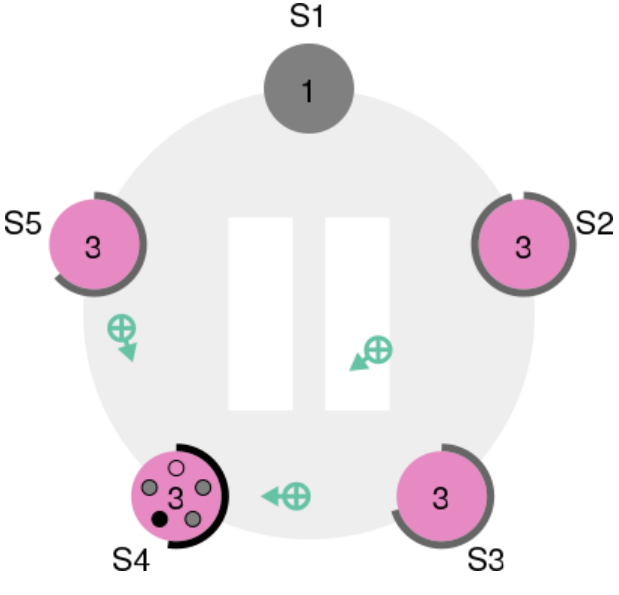
Split vote



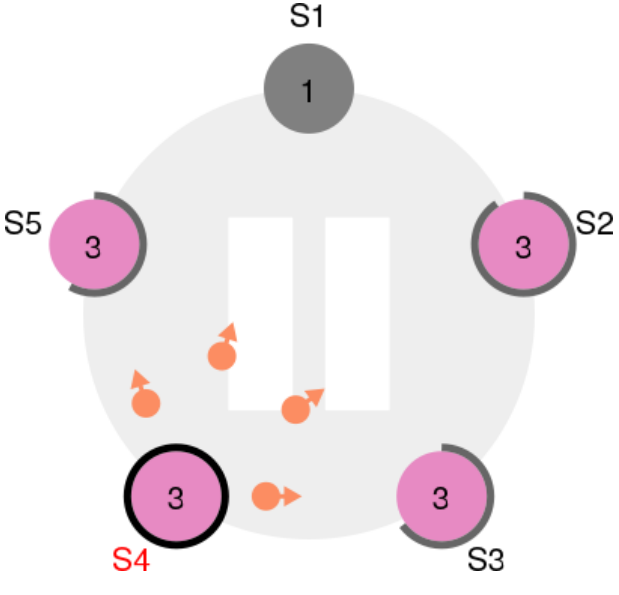
Split vote



Split vote



Split vote



Raft's ecosystem

- Base of CockroachDB

- Widespread use
- "Paxos made simple"

Raft's ecosystem

- Base of CockroachDB

- Widespread use
- "Paxos made simple"

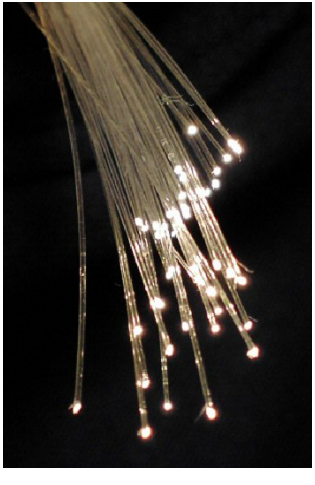
Raft's implementations:

cppa-raft, LogCabin, bspolley/raft, noeleo/raft, whitewater, willemt/raft, NRaft, dupdob/RAFTiNG, dinghy, melee, raft-clj, rodriguezvalencia/rafting, draft, zraft_lib, eraft, huckleberry, rafter, rafterl, Flotten, graft, go-raft, etcd/raft, hashicorp/raft, jpathy/raft, peterbourgon/raft, pontoon, seaturtles, kontiki, allengeorge/libraft, barge, chicm/CmRaft, copycat, drpicox/uoc-raft-2013p, jalvaro/raft, jgroups-raft, RaftKVDatabase/JSimpleDB, OpenDaylight, pvilas/raft, Raft4WS, Raft-JVM, r4j, liferaft, benbjohnson/raft.js, dannycoates/raft-core, kanaka/raft.js, skiff, ocaml-raft, py-raft, simpleRaft, floss, giraft, harryw/raft, zodiac-prime, hoverbear/raft, akka-raft, archie/raft, cb372/raft, chelan, ckite, scalaraft, lite-raft, C5, yora, srned/Prez, fxsjy/Ins

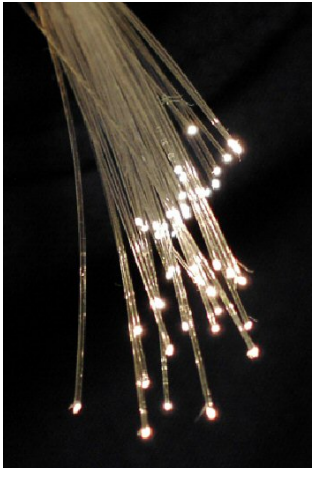
https://pbs.twimg.com/profile_images/592837647577587714/eNMItpUG_400x400.png

Part 7

Network partitioning



Network partitioning



+

http://upload.wikimedia.org/wikipedia/commons/b/b9/Kitchen-Scissors.jpg
http://upload.wikimedia.org/wikipedia/commons/4/49/Fibreoptic.jpg

CAP theorem

- Systems' properties:
  1. **C**onsistency

CAP theorem

- Systems' properties:
  1. **C**onsistency
  2. **A**vailability

CAP theorem

- Systems' properties:
  1. **C**onsistency
  2. **A**vailability
  3. **P**artition-tolerance

CAP theorem

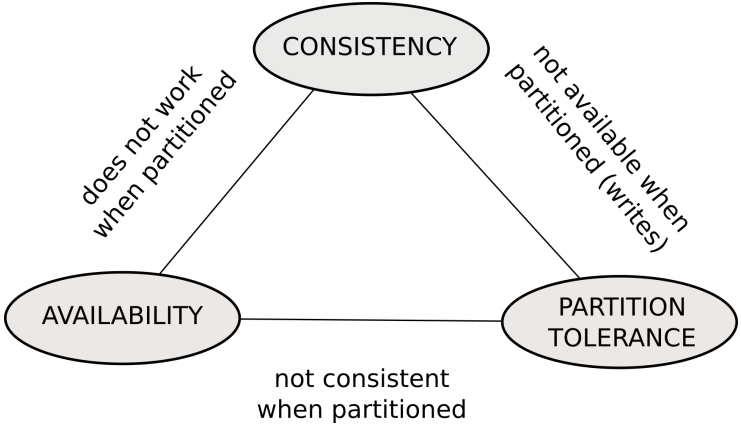
- Systems' properties:
  1. **C**onsistency
  2. **A**vailability
  3. **P**artition-tolerance

CAP theorem

Having C/A/P – choose (at most) 2 of 3!

CAP theorem

- Systems' properties:
  1. **C**onsistency
  2. **A**vailability
  3. **P**artition-tolerance



CAP theorem

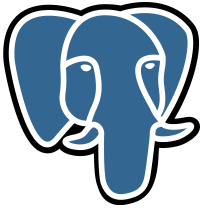
Having C/A/P – choose (at most) 2 of 3!

Example systems' classification

- Having:
  - **C**onsistency
  - **A**vailability
  - **P**artition-tolerance

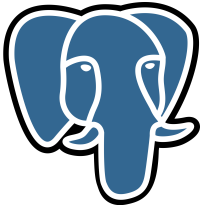
Example systems' classification

- Having:
  - **C**onsistency
  - **A**vailability
  - **P**artition-tolerance
- Examples:
  - PostgreSQL – **CA**P



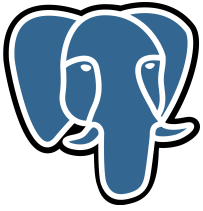
Example systems' classification

- Having:
  - **C**onsistency
  - **A**vailability
  - **P**artition-tolerance
- Examples:
  - PostgreSQL – **CA**P
  - MongoDB – **C**A**P**



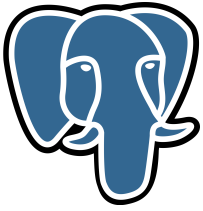
Example systems' classification

- Having:
  - **C**onsistency
  - **A**vailability
  - **P**artition-tolerance
- Examples:
  - PostgreSQL – **CA**P
  - MongoDB – **CP**
  - Cassandra – C**AP**



Example systems' classification

- Having:
  - **C**onsistency
  - **A**vailability
  - **P**artition-tolerance
- Examples:
  - PostgreSQL – **CA**P
  - MongoDB – **C**A**P**
  - Cassandra – C**AP**
- In reality:
  - Modulo bugs! :-)
  - Often configuration-dependent...
  - "*Call me maybe*" series



mongoDB

cassandra

https://upload.wikimedia.org/wikipedia/commons/2/29/Postgresql_elephant.svg
https://upload.wikimedia.org/wikipedia/commons/5/5e/Cassandra_logo.svg
https://upload.wikimedia.org/wikipedia/en/e/eb/MongoDB_Logo.png

Part 8

Synchronizing time

- Given events:
  - **A** (at **nodeA**)
  - **B** (at **nodeB**)
- Did **A** happened before **B**?

Synchronizing time

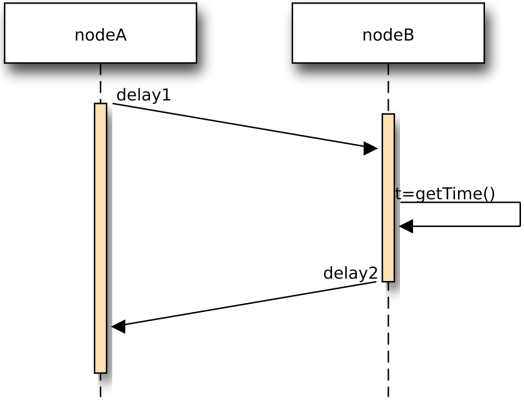
- Given events:
  - **A** (at **nodeA**)
  - **B** (at **nodeB**)
- Did **A** happened before **B**?
- Check their time!

Synchronizing time

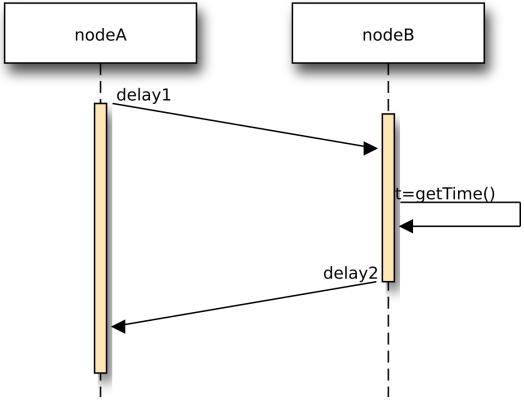
- Given events:
  - **A** (at **nodeA**)
  - **B** (at **nodeB**)
- Did **A** happened before **B**?
- Check their time!
  - Are clocks in sync? (*clock skew*)
    - How precise?
    - What about latencies?
  - Will clocks remain in sync? (*clock drift*)

http://1.bp.blogspot.com/-m4Wd15SQ3eY/T75YMmeYwrI/AAAAAAAAAaw/nrH2BxGSimw/s320/Clock+art-Salvador+Dali.jpg

Cristian's algorithm

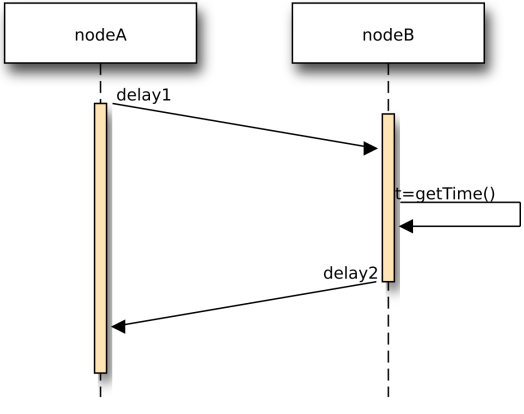


Cristian's algorithm



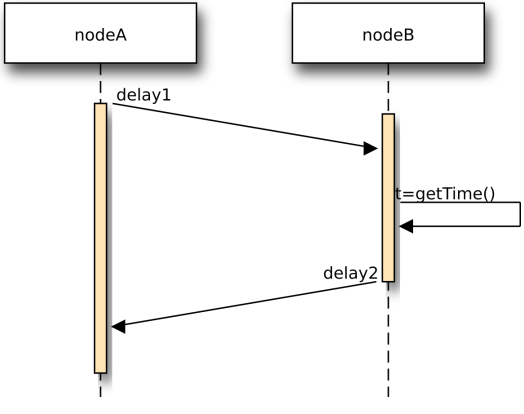
- Assuming $delay1 = delay2$
- New time: $t_1 = t + \frac{RTT}{2}$
- Where:
  - RTT – Round Trip Time
  - t_1 – new local time
  - t – time from remote party

Cristian's algorithm



- Assuming $delay1 = delay2$
- New time: $t_1 = t + \frac{RTT}{2}$
- Where:
  - RTT – Round Trip Time
  - t_1 – new local time
  - t – time from remote party
- Fairly Simple™;-)
- Often imprecise

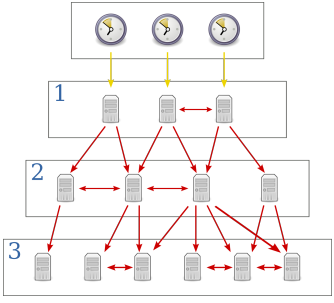
Cristian's algorithm



- Assuming $delay1 = delay2$
- New time: $t_1 = t + \frac{RTT}{2}$
- Where:
  - RTT – Round Trip Time
  - t_1 – new local time
  - t – time from remote party
- Fairly Simple™;-)
- Often imprecise
- Single point of synchronization
- Does not scale

NTP's algorithm

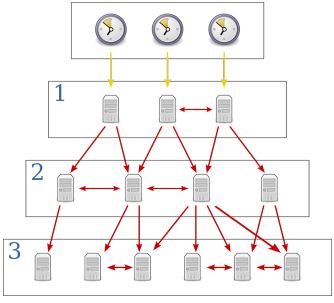
- **N**etwork **T**ime **P**rotocol

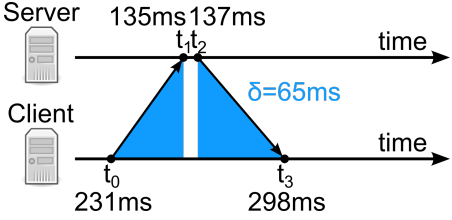


- More clock sources
- Better scalability

NTP's algorithm

- **N**etwork **T**ime **P**rotocol

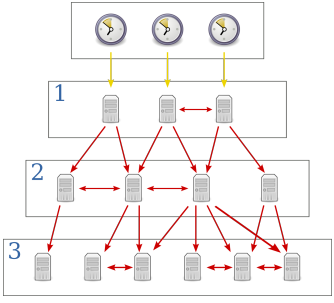




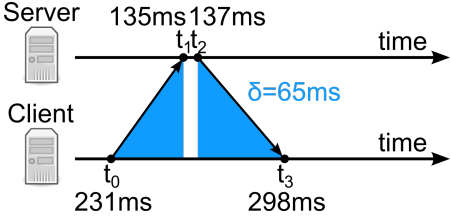
- More clock sources
- Better scalability

NTP's algorithm

- **N**etwork **T**ime **P**rotocol



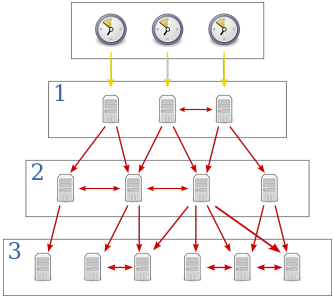
- More clock sources
- Better scalability



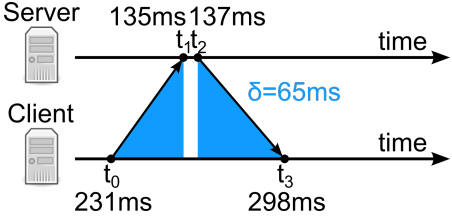
- Better *RTT* estimation
- Probabilistic analysis of data
- Continuous operation

NTP's algorithm

- **N**etwork **T**ime **P**rotocol



- More clock sources
- Better scalability



- Better *RTT* estimation
- Probabilistic analysis of data
- Continuous operation

- Commonly use in the Internet
- Still clocks skew and drift. . .

http://upload.wikimedia.org/wikipedia/commons/c/c9/Network_Time_Protocol_servers_and_clients.svg

http://upload.wikimedia.org/wikipedia/commons/8/8d/NTP-Algorithm.svg

Now what if...

- Cannot solve?

Now what if...

- Cannot solve?
  - Step back
  - Find a different path...

Now what if...

- Cannot solve?
  - Step back
  - Find a different path...
- What is REALLY needed?

Now what if...

- Cannot solve?
  - Step back
  - Find a different path...
- What is REALLY needed?
  - Ordering knowledge
  - "What happened first?"

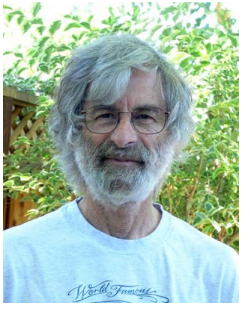
Now what if...

- Cannot solve?
  - Step back
  - Find a different path...
- What is REALLY needed?
  - Ordering knowledge
  - "What happened first?"
- Can be done without time...

Part 9

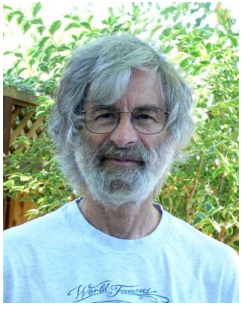
Logical ordering

- **L**amport's **T**ime **S**tamp
- Proposed by *Leslie Lamport*
- Establishes *happens-before* relation



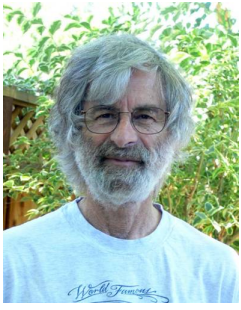
Logical ordering

- **L**amport's **T**ime **S**tamp
- Proposed by *Leslie Lamport*
- Establishes *happens-before* relation
- Each process has counter *lts*
- Initially $lts = 0$



Logical ordering

- **L**amport's **T**ime **S**tamp
- Proposed by *Leslie Lamport*
- Establishes *happens-before* relation
- Each process has counter *lts*
- Initially $lts = 0$
- Algorithm:
  - Before sending: $lts = lts + 1$
  - Include counter in the message
  - When receiving: $lts = max(lts, msg\_lts) + 1$
- Also can increment at will

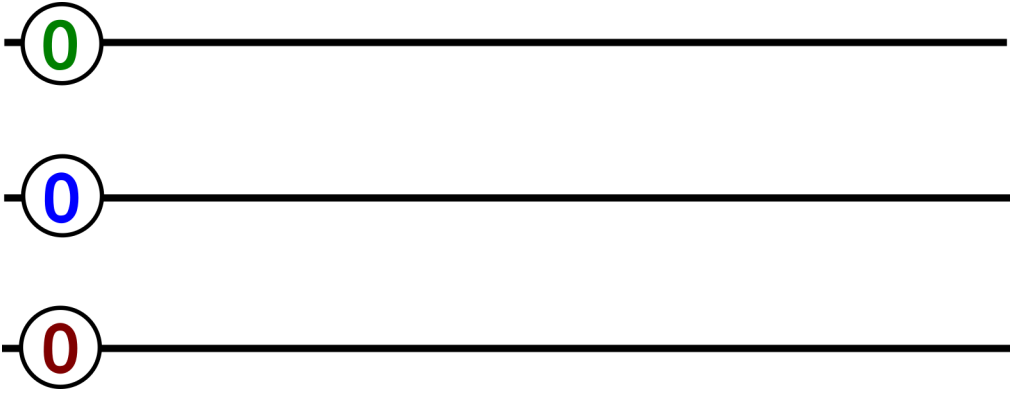


https://upload.wikimedia.org/wikipedia/commons/5/50/Leslie_Lamport.jpg

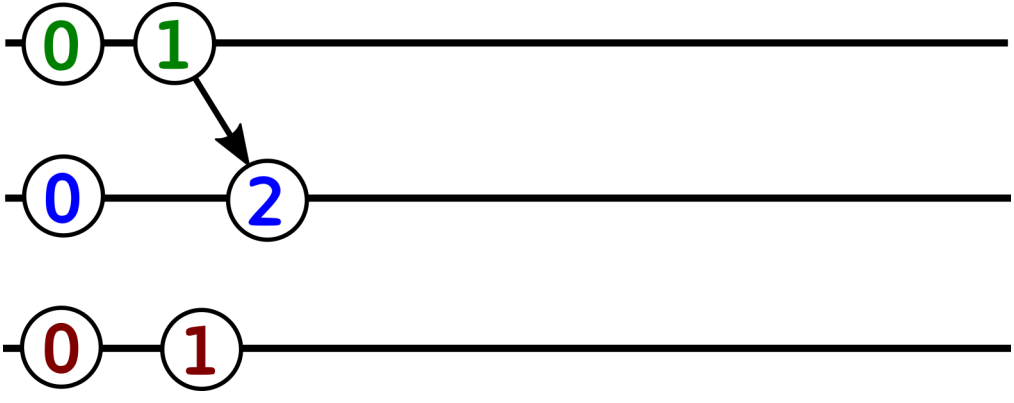
Come again?

http://lshss.pubs.asha.org/data/Journals/LSHSS/930529/LSHSS_45_3_220fig0.jpeg

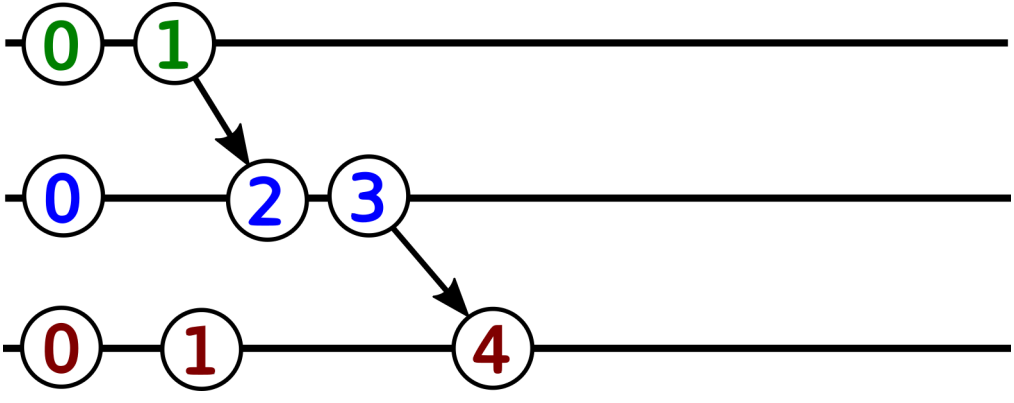
Example



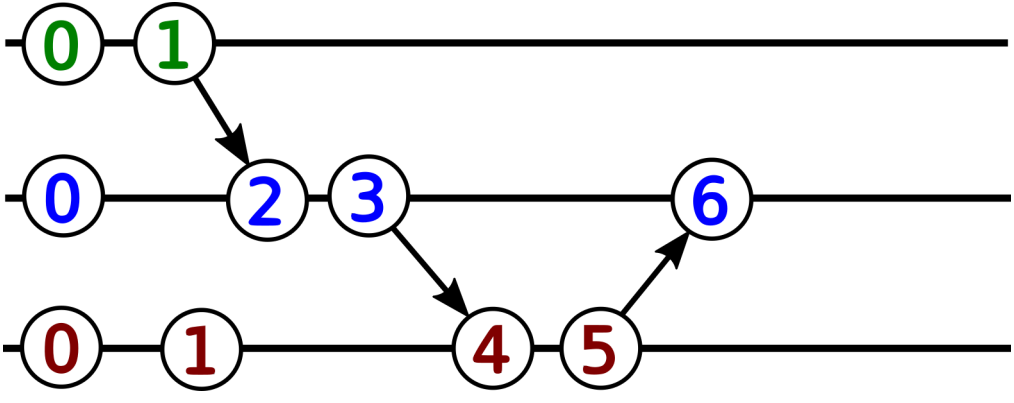
Example



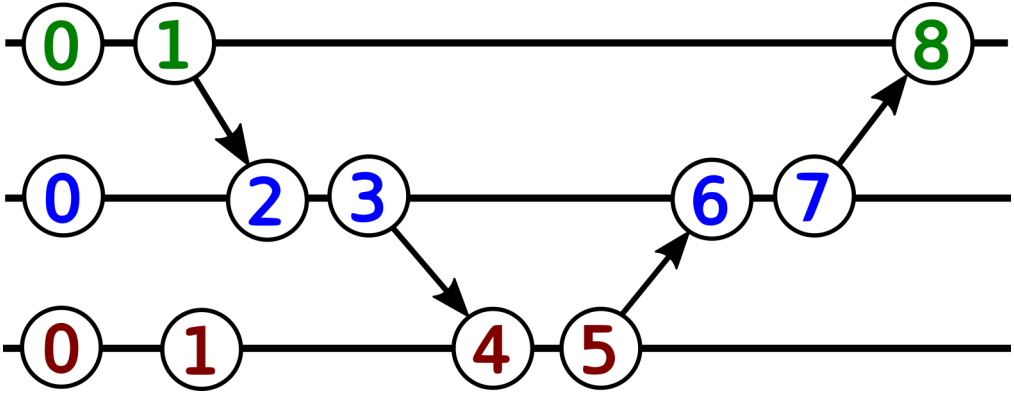
Example



Example



Example



Properties

- Events: A and B
- With timestamps: $C(A)$ and $C(B)$

Properties

- Events: A and B
- With timestamps: $C(A)$ and $C(B)$
- $A \rightarrow B \Rightarrow C(A) < C(B)$
- Note: implication!

Properties

- Events: A and B
- With timestamps: $C(A)$ and $C(B)$
- $A \rightarrow B \Rightarrow C(A) < C(B)$
- Note: implication!
- What does $C(A) < C(B)$ mean?

Properties

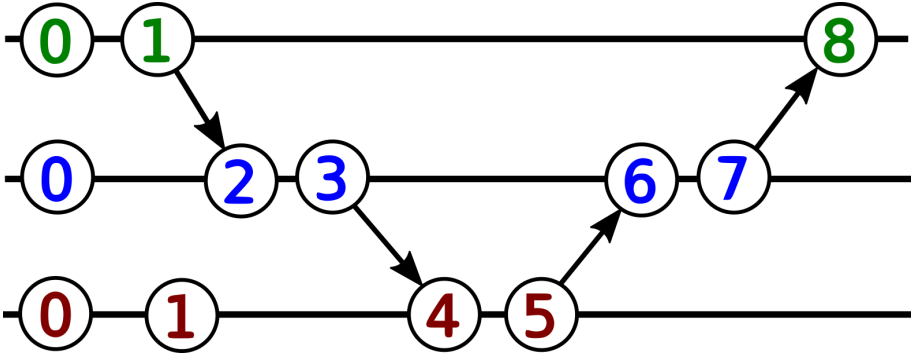
- Events: A and B
- With timestamps: $C(A)$ and $C(B)$
- $A \rightarrow B \Rightarrow C(A) < C(B)$
- Note: implication!
- What does $C(A) < C(B)$ mean?
  - One of:
    - $A \rightarrow B$
    - $A \parallel B$
  - But **not** $B \rightarrow A$

Properties

- Events: A and B
- With timestamps: $C(A)$ and $C(B)$
- $A \rightarrow B \Rightarrow C(A) < C(B)$
- Note: implication!
- What does $C(A) < C(B)$ mean?
  - One of:
    - $A \rightarrow B$
    - $A \parallel B$
  - But **not** $B \rightarrow A$
- Cause-effect ordering!

Let's check!

$$C(A) < C(B) \Leftrightarrow A \rightarrow B \vee A \parallel B$$



Lamport's timestamps at . . .

- Pretty much everywhere. . . ;)
- Including. . .

Lamport's timestamps at . . .

- Pretty much everywhere. . . ;)
- Including. . .

Lamport's timestamps at . . .

- Pretty much everywhere. . . ;)
- Including. . .

- Part of BTS software:
  - WCDMA OAM
  - SRAN OAM
  - Cloud OAM
- Distributed service registration
- Introduces logical ordering
- Registration updates' ordering

https://upload.wikimedia.org/wikipedia/commons/0/02/Nokia_wordmark.svg

Part 10

Enhanced logical ordering

- Similar to Lamport's timestamps
- Detects parallel events

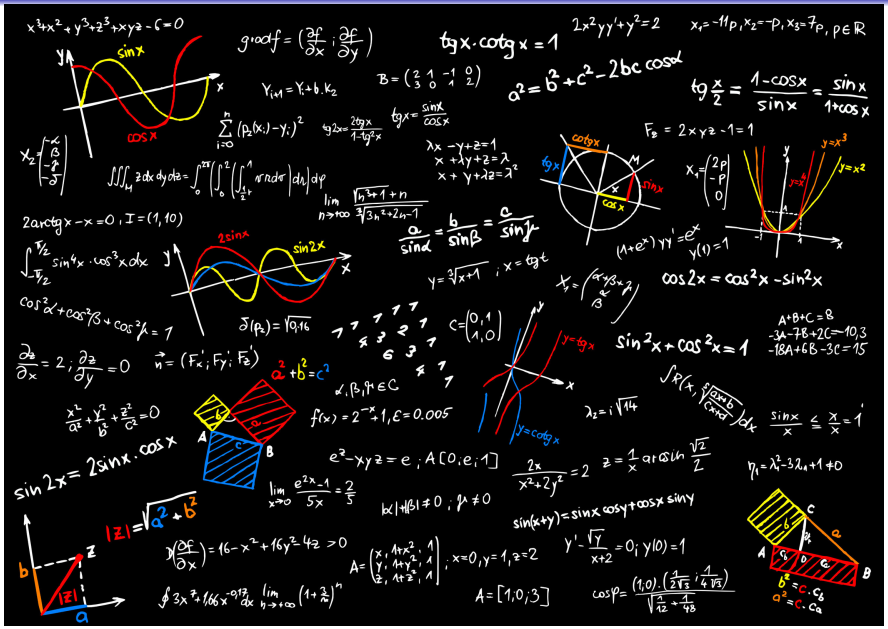
Enhanced logical ordering

- Similar to Lamport's timestamps
- Detects parallel events
- Each process has vector of counters $vc = [c1, c2, ..., cN]$
- Initially $vc = [0, 0, ..., 0]$

Enhanced logical ordering

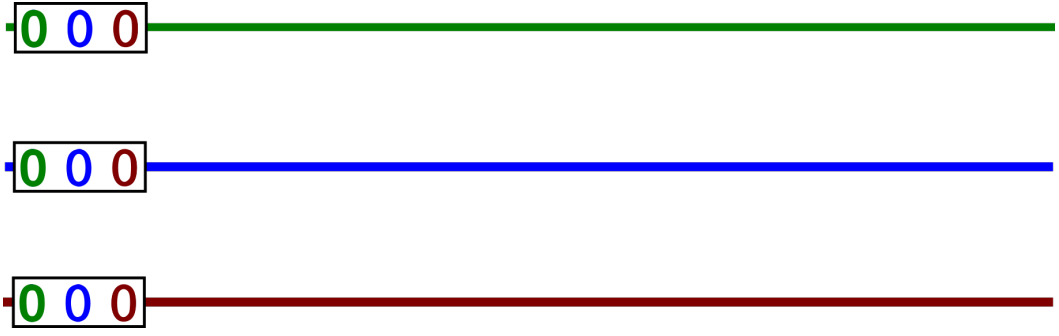
- Similar to Lamport's timestamps
- Detects parallel events
- Each process has vector of counters $vc = [c1, c2, ..., cN]$
- Initially $vc = [0, 0, ..., 0]$
- Algorithm:
  - On i-th node
  - Before sending: $vc[i] = vc[i] + 1$
  - Include vector in the message
  - When receiving:
    - $\forall j \neq i : vc[j] = max(vc[j], msg\_vc[j])$
    - $vc[i] = vc[i] + 1$
- Also can increment local counter at will

You don’t say...

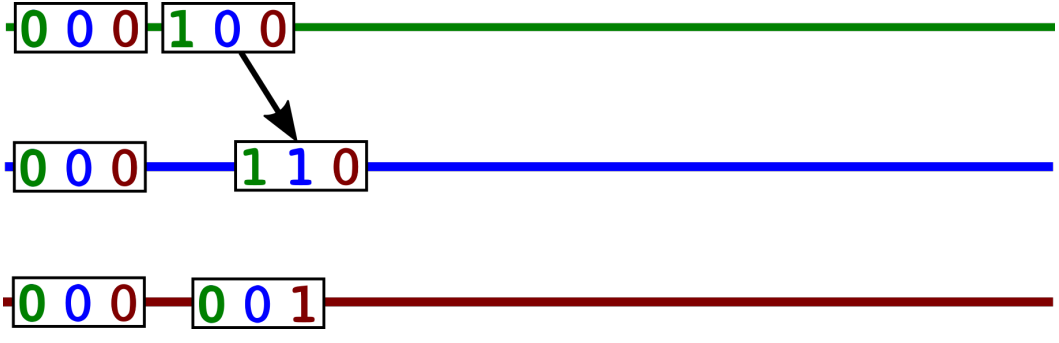


http://syosseths.com/z/acapics/mathslide.jpg

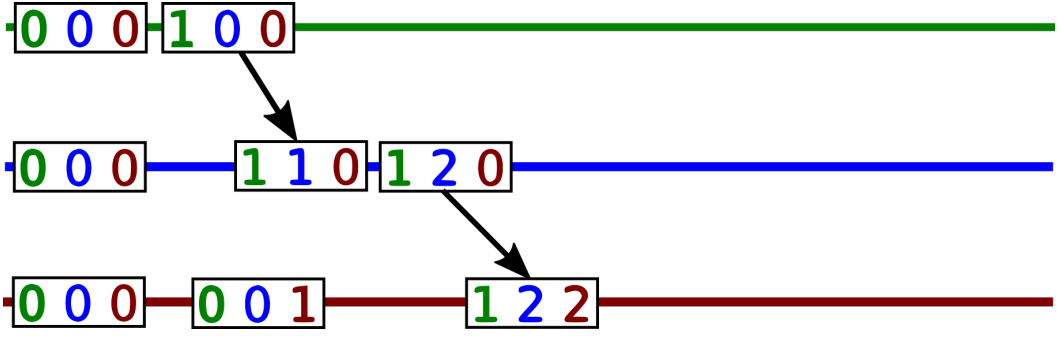
Example



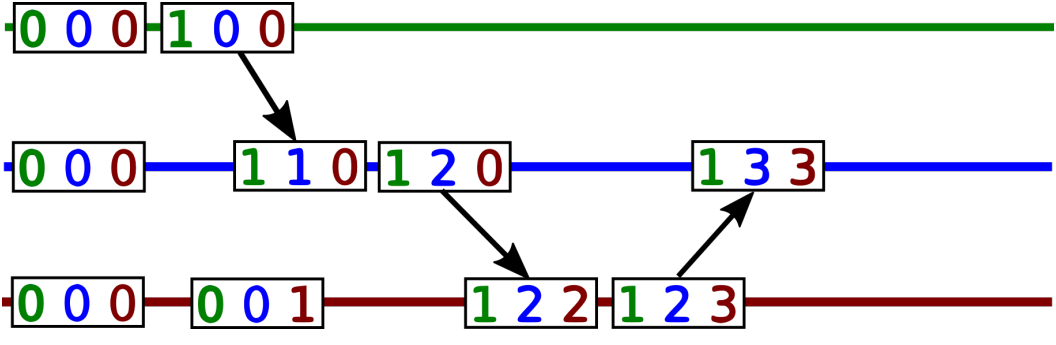
Example



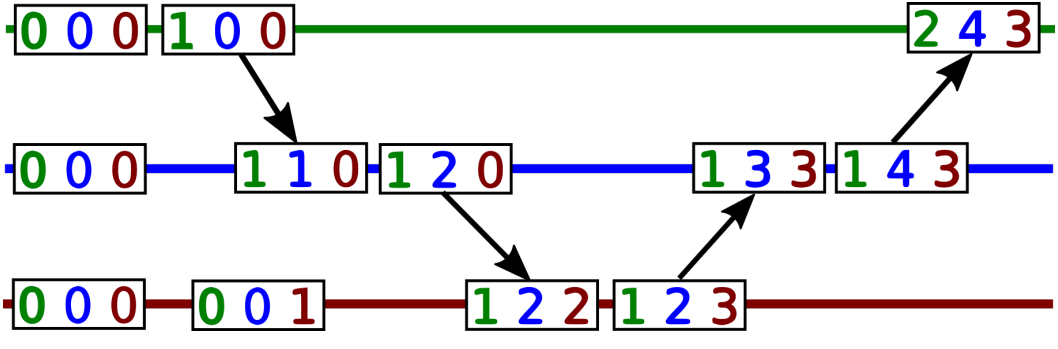
Example



Example



Example



Properties

- Events: A and B
- With vector clocks: $VC(A)$ and $VC(B)$

Properties

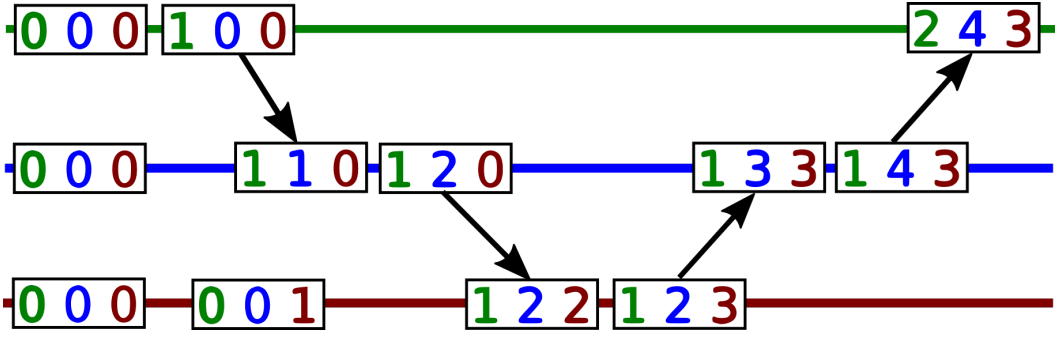
- Events: A and B
- With vector clocks: $VC(A)$ and $VC(B)$
- Operations:
  - $vc_1 = vc_2 \Leftrightarrow \forall i : vc_1[i] = vc_2[i]$
  - $vc_1 \leq vc_2 \Leftrightarrow \forall i : vc_1[i] \leq vc_2[i]$
  - $vc_1 < vc_2 \Leftrightarrow vc_1 \leq vc_2 \wedge \exists i : vc_1[i] < vc_2[i]$

Properties

- Events: A and B
- With vector clocks: $VC(A)$ and $VC(B)$
- Operations:
  - $vc_1 = vc_2 \Leftrightarrow \forall i : vc_1[i] = vc_2[i]$
  - $vc_1 \leq vc_2 \Leftrightarrow \forall i : vc_1[i] \leq vc_2[i]$
  - $vc_1 < vc_2 \Leftrightarrow vc_1 \leq vc_2 \wedge \exists i : vc_1[i] < vc_2[i]$
- Causality:
  - $A \rightarrow B \Leftrightarrow vc(A) < vc(B)$
  - $A \parallel B \Leftrightarrow \neg(vc(A) \leq vc(B)) \wedge \neg(vc(B) \leq vc(A))$

Let's check!

- $A \rightarrow B \Leftrightarrow vc(A) < vc(B)$
- $A \parallel B \Leftrightarrow \neg(vc(A) \leq vc(B)) \wedge \neg(vc(B) \leq vc(A))$



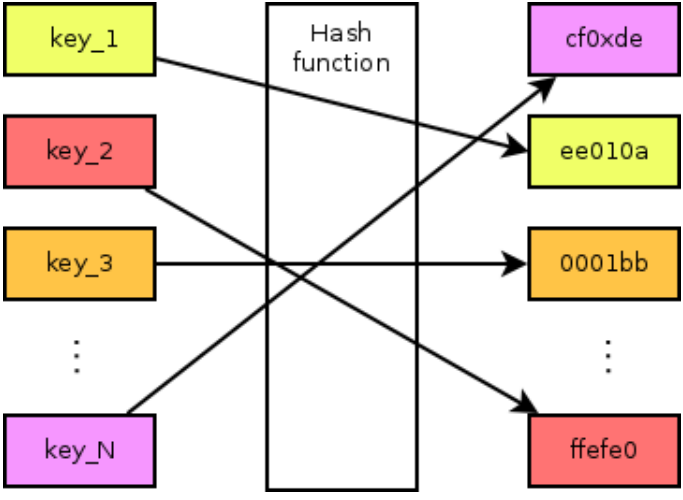
Vector clocks inside Riak

- Riak:
  - NoSQL database
  - Based on "Dynamo" paper
- VC for eventual consistency
- Help solve conflicts:
  - Resolve duplicates
  - Detect latest version
  - Detect simultaneous updates

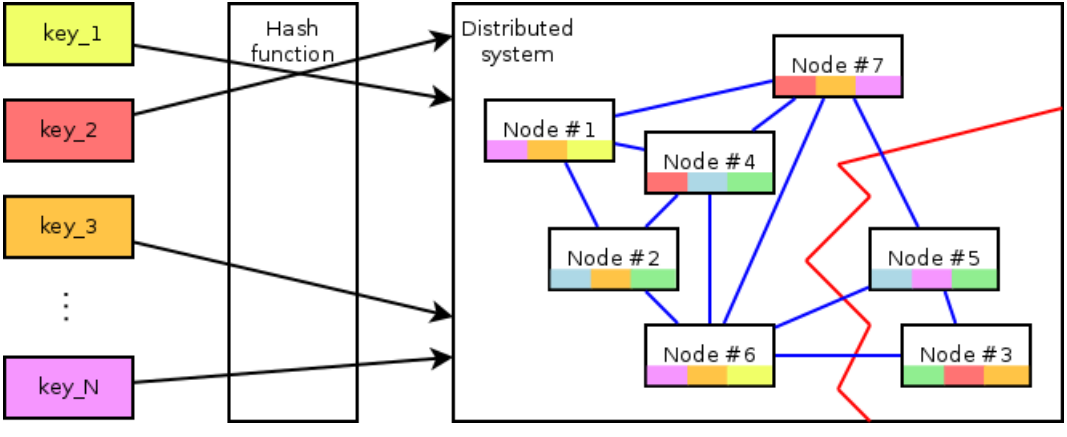
https://upload.wikimedia.org/wikipedia/en/8/8e/Riak_distributed_NoSQL_key-value_data_store_logo.png

Part 11

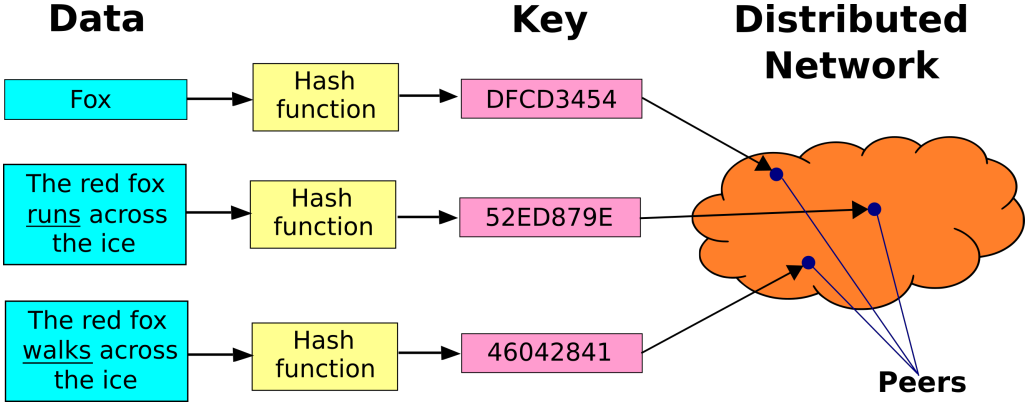
Hash table


key_1
key_2
key_3
key_N
Hash function
cf0xde
ee010a
0001bb
ffefe0

Distributed Hash Table (DHT)



Keys distribution



https://upload.wikimedia.org/wikipedia/commons/9/98/DHT_en.svg

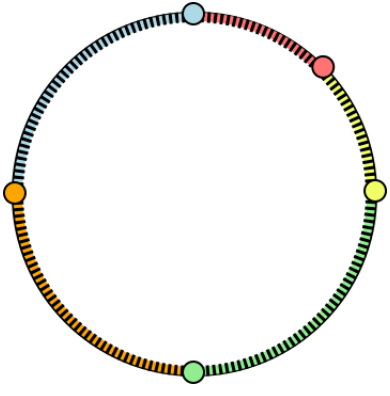
DHTs everywhere

- Common since the end of ’90
- Base of most NoSQL DBs
- Advantages:
  - Massive storage
  - Decentralized
  - Fault tolerant
  - Scalable

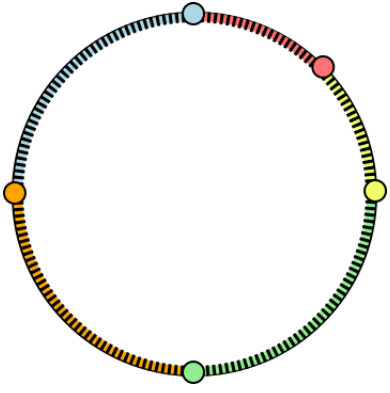
https://upload.wikimedia.org/wikipedia/commons/5/5e/Cassandra_logo.svg
http://www.gluster.org/images/antmascot.png?1432254431
https://upload.wikimedia.org/wikipedia/en/8/8e/Riak_distributed_NoSQL_key-value_data_store_logo.png
https://upload.wikimedia.org/wikipedia/commons/6/6d/Transmission_icon.png
https://upload.wikimedia.org/wikipedia/en/9/9f/UTorrent_%28logo%29.png
https://upload.wikimedia.org/wikipedia/commons/9/9e/Qbittorrent_logo.png
https://upload.wikimedia.org/wikipedia/commons/f/ff/Qbittorrent_mascot.png
https://upload.wikimedia.org/wikipedia/en/d/d5/BitComet_logo.svg
https://upload.wikimedia.org/wikipedia/commons/0/0a/Shareaza.png
https://upload.wikimedia.org/wikipedia/en/d/d6/Logo_of_BearShare_from_Website.jpg

Part 12

The ring

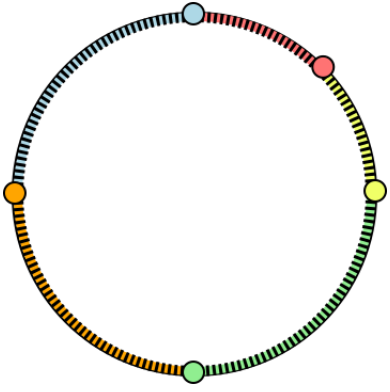


The ring



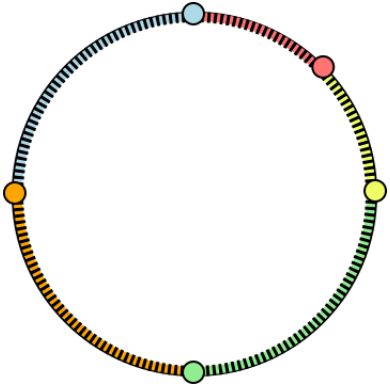
- Ring-organized
- m-bits (2^m possible entries)
- Ordered clock-wise
- Node keeps preceding keys

The ring



- Ring-organized
- m-bits (2^m possible entries)
- Ordered clock-wise
- Node keeps preceding keys
- Nodes hashed by:
  - Address (IP?)
  - Port

The ring



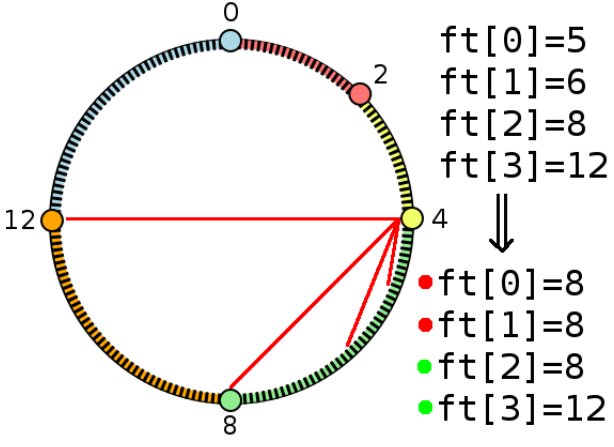
- Ring-organized
- m-bits (2^m possible entries)
- Ordered clock-wise
- Node keeps preceding keys
- Nodes hashed by:
  - Address (IP?)
  - Port
- Keys:
  - User's hash
  - Share key space with nodes
- *SHA-1* commonly used

Finger table

- Each node keeps own
- m-entries
- $(n + 2^i)\%2^m$
- Where:
  - i – entry number
  - n – node's number
  - m – hash bits

Finger table

- Each node keeps own
- m-entries
- $(n + 2^i)\%2^m$
- Where:
  - i – entry number
  - n – node's number
  - m – hash bits
- Here: $n = 4$, $m = 4$
- i.e. $2^4 = 16$ entries



One ring to hash them all!

https://upload.wikimedia.org/wikipedia/commons/b/b7/Unico_Anello.png

Query algorithm

1. Having query for k
2. If has k – return it

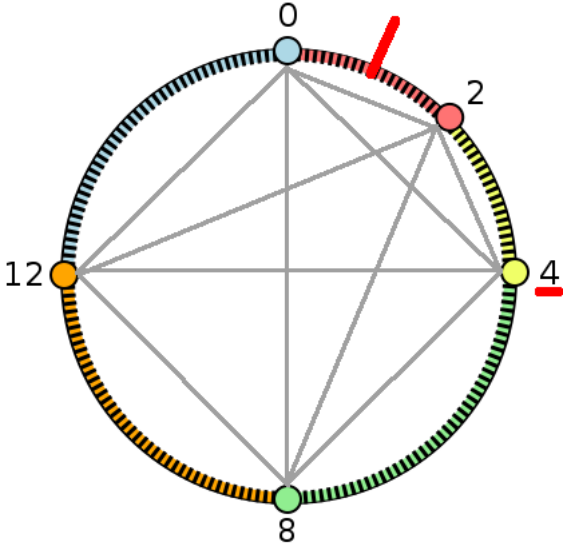
Query algorithm

1. Having query for k
2. If has k – return it
3. $next = \max(ft[i] : i \in \langle 0; m) \wedge ft[i] \leqslant k)$
4. Note: $ft[i] \leqslant k$ is done on the ring!

Query algorithm

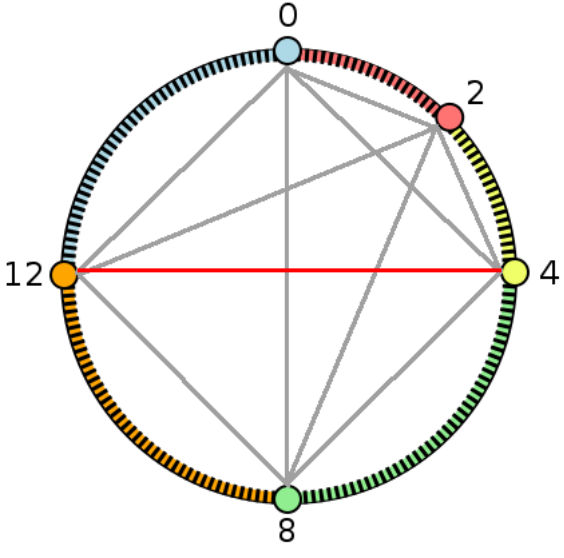
1. Having query for k
2. If has k – return it
3. $next = \max(ft[i] : i \in \langle 0; m) \wedge ft[i] \leqslant k)$
4. Note: $ft[i] \leqslant k$ is done on the ring!
5. If has $next$ – return it
6. Else return first successor

Query example



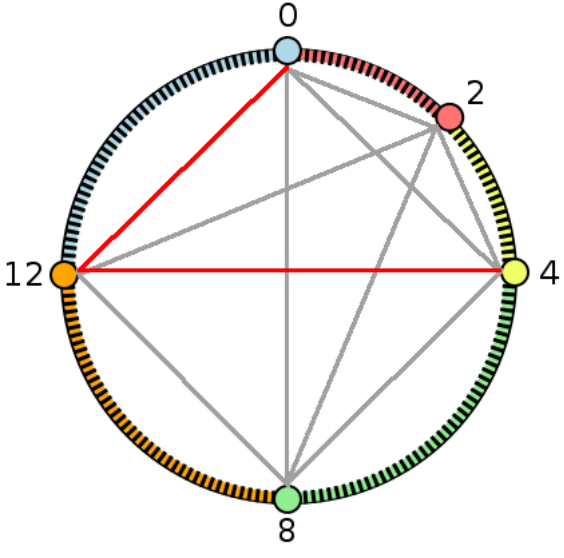
- Node 4 – query 1

Query example



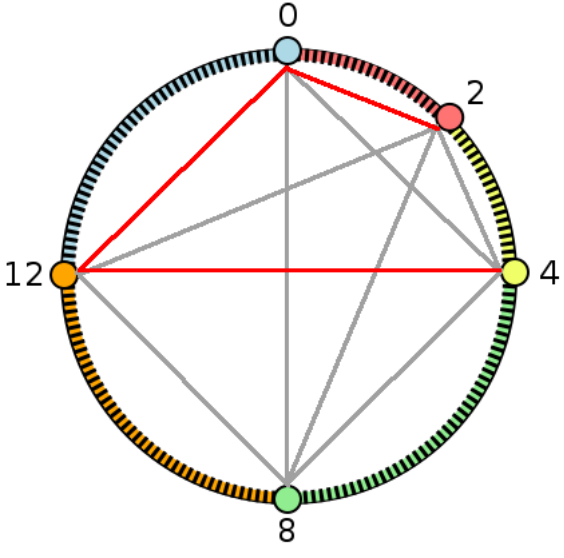
- Node 4 – query 1
- Finger tables:
  1. $n = 4$
     - $ft[0] = 8$
     - $ft[1] = 8$
     - $ft[2] = 8$
     - $ft[3] = 12$

Query example



- Node 4 – query 1
- Finger tables:
  1. $n = 4$
     - $ft[0] = 8$
     - $ft[1] = 8$
     - $ft[2] = 8$
     - $ft[3] = 12$
  2. $n = 12$
     - $ft[0] = 0$
     - $ft[1] = 0$
     - $ft[2] = 0$
     - $ft[3] = 4$

Query example



- Node 4 – query 1
- Finger tables:
  1. $n = 4$
     - $ft[0] = 8$
     - $ft[1] = 8$
     - $ft[2] = 8$
     - $ft[3] = 12$
  2. $n = 12$
     - $ft[0] = 0$
     - $ft[1] = 0$
     - $ft[2] = 0$
     - $ft[3] = 4$
  3. $n = 0$
     - $ft[0] = 2$
     - $ft[1] = 2$
     - $ft[2] = 4$
     - $ft[3] = 8$

Additional notes

- $O(\log N)$ memory
- (Amortized) $O(\log N)$ lookup
- Considered almost $O(1)$

Additional notes

- $O(\log N)$ memory
- (Amortized) $O(\log N)$ lookup
- Considered almost $O(1)$
- Other scenarios:
  - Initialization – query for self
  - Nodes joining – splitting ranges
  - Nodes leaving – merging ranges
  - Nodes crashing = nodes leaving

Additional notes

- $O(\log N)$ memory
- (Amortized) $O(\log N)$ lookup
- Considered almost $O(1)$
- Other scenarios:
  - Initialization – query for self
  - Nodes joining – splitting ranges
  - Nodes leaving – merging ranges
  - Nodes crashing = nodes leaving
- Failures handing:
  - Duplication of data
  - More entries per finger table
  - Keeping few predecessors
  - Monitoring neighborhood

Part 13

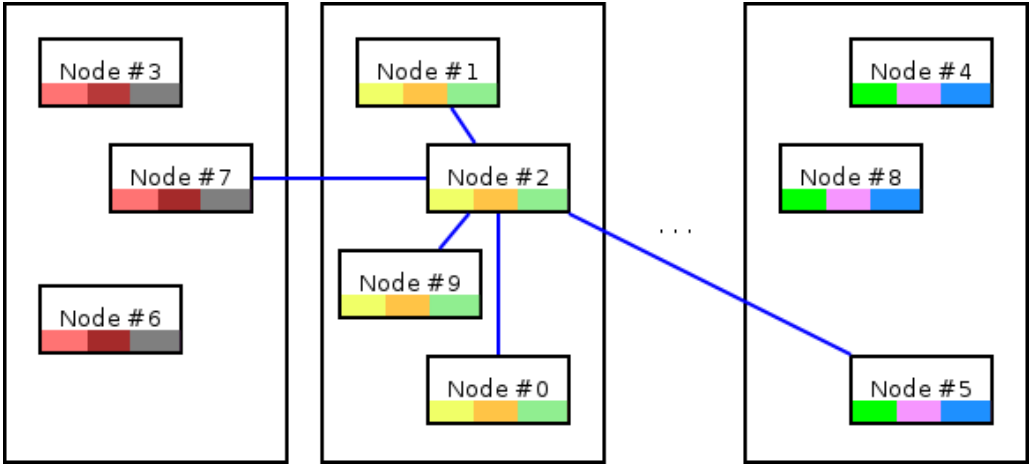
Idea

- Split space:
  - *k affinity groups*
  - $k \approx \sqrt{N}$

Idea

- Split space:
  - *k affinity groups*
  - $k \approx \sqrt{N}$
- Group based on node's hash
- Node in a group knows:
  - All members of its group
  - One node per other groups
  - All group's hashes

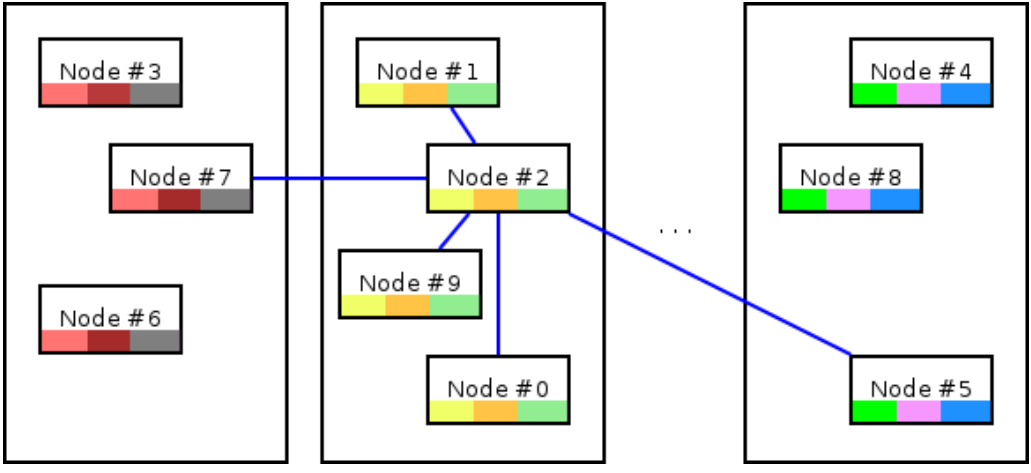
Example - node 2



Local query

Querying for key from the same affinity group

Node #2 querying "yellow" key



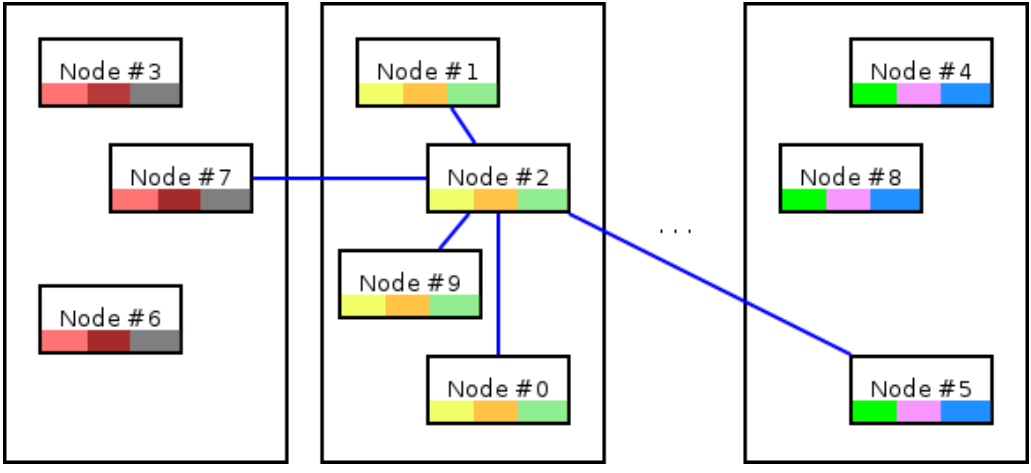
It’s here!

http://asalesguyrecruiting.com/wp-content/uploads/2014/03/Oh-Hell-Yeah-566x372.png

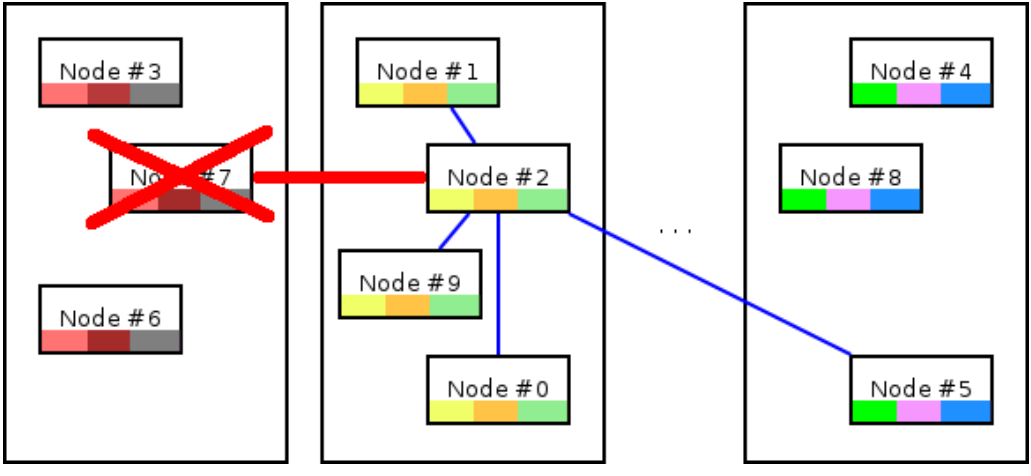
Remote query

Querying for key from different affinity group

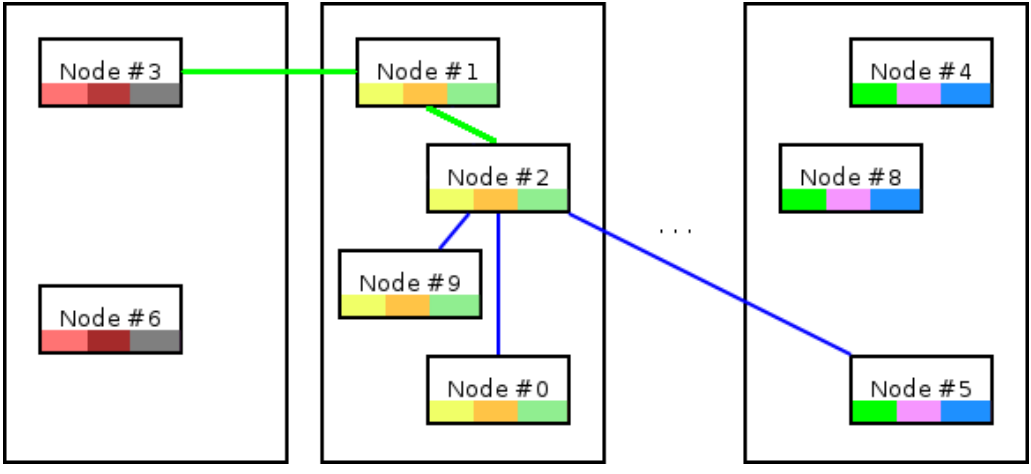
Node #2 querying "red" key



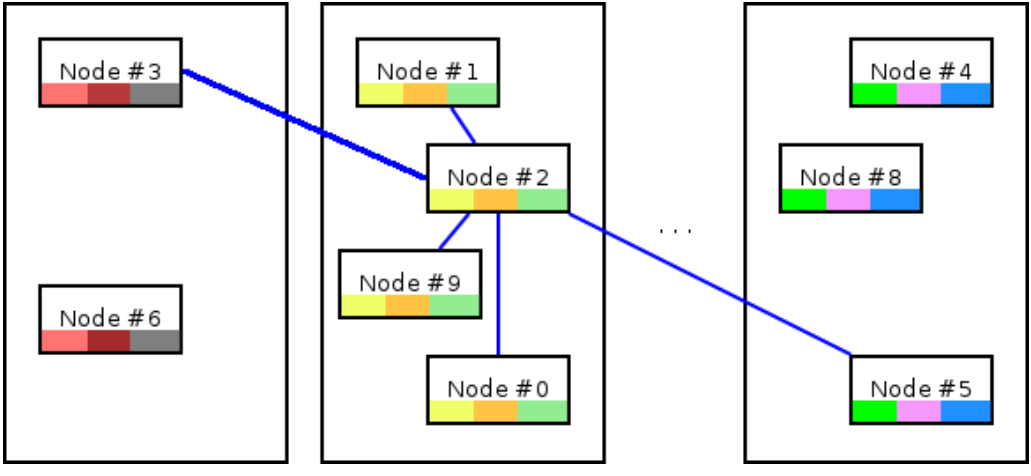
Node #2 querying "red" key



Node #2 querying "red" key



Node #2 querying "red" key



Additional notes

- (Amortized) $O(1)$ lookup

Additional notes

- (Amortized) $O(1)$ lookup
- $O(\sqrt{N})$ memory
  - Scary?
  - Not really...
  - $N \approx$ *millions* $\rightarrow$ few MBs...

Additional notes

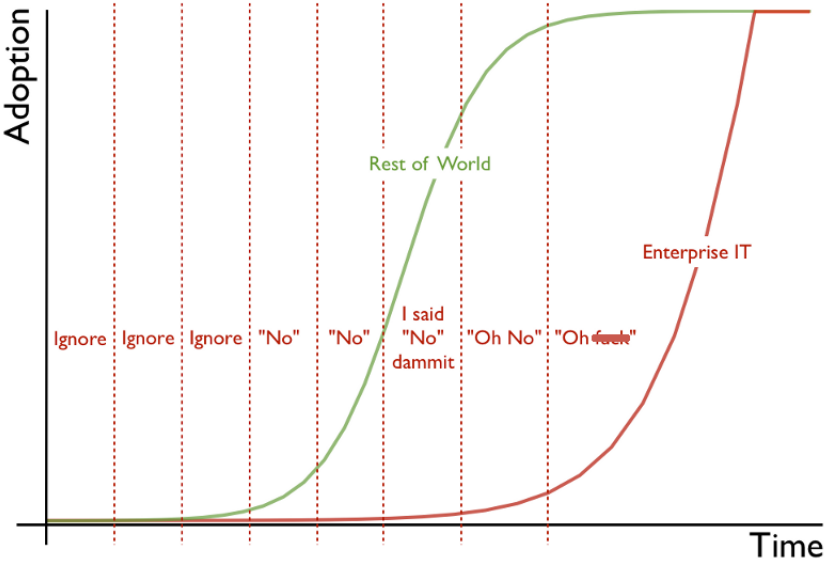
- (Amortized) $O(1)$ lookup
- $O(\sqrt{N})$ memory
  - Scary?
  - Not really...
  - $N \approx$ *millions* $\rightarrow$ few MBs...
- Other scenarios:
  - Initialization – query own affinity group
  - Nodes joining – more data replicated
  - Nodes leaving – less replication
  - Nodes crashing = nodes leaving

Additional notes

- (Amortized) $O(1)$ lookup
- $O(\sqrt{N})$ memory
  - Scary?
  - Not really...
  - $N \approx$ *millions* $\rightarrow$ few MBs...
- Other scenarios:
  - Initialization – query own affinity group
  - Nodes joining – more data replicated
  - Nodes leaving – less replication
  - Nodes crashing = nodes leaving
- Failures handing:
  - Monitoring own affinity group
  - Querying others for affinity neighbors

Part 14

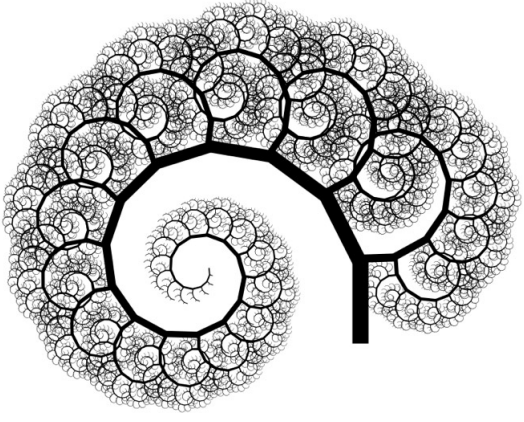
Cloud @ enterprises



http://enterpriseitadoption.com/images/adoption_cycle.png

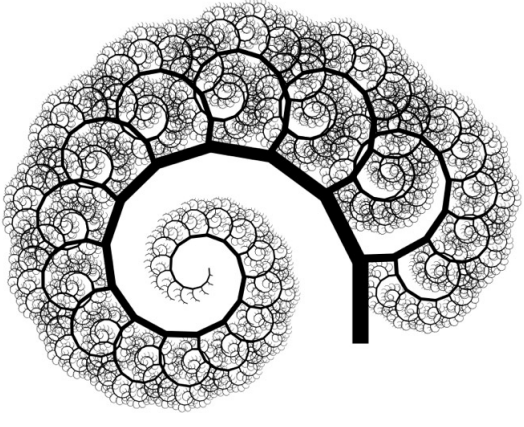
More to learn...

- Coursera:
  - Cloud computing concepts – part 1 & 2
  - Cloud computing applications
  - Cloud networking



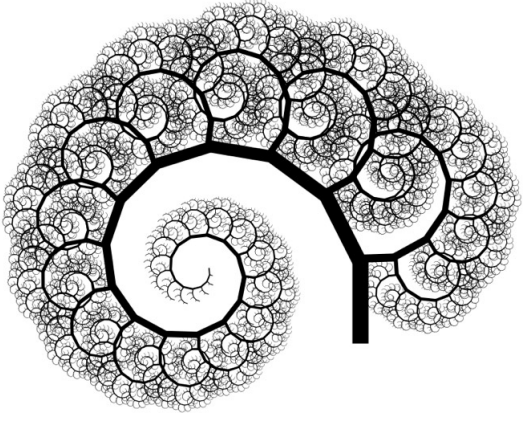
More to learn...

- Coursera:
  - Cloud computing concepts – part 1 & 2
  - Cloud computing applications
  - Cloud networking
- Other problems/algorithms:
  - Mutual exclusion
  - Snapshoting
  - Multicast problem
  - Transactions



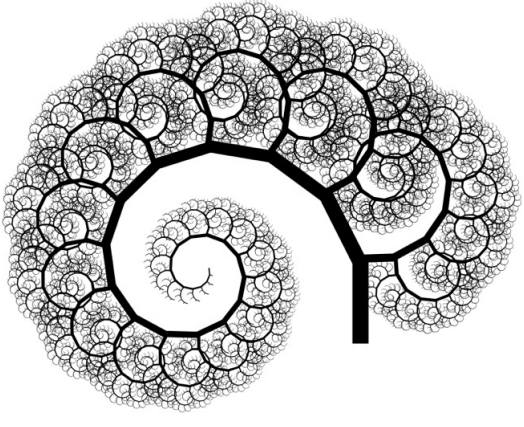
More to learn...

- Coursera:
  - Cloud computing concepts – part 1 & 2
  - Cloud computing applications
  - Cloud networking
- Other problems/algorithms:
  - Mutual exclusion
  - Snapshoting
  - Multicast problem
  - Transactions
- General concepts:
  - Microservices
  - Continuous deployment
  - NoSQL



More to learn...

- Coursera:
  - Cloud computing concepts – part 1 & 2
  - Cloud computing applications
  - Cloud networking
- Other problems/algorithms:
  - Mutual exclusion
  - Snapshoting
  - Multicast problem
  - Transactions
- General concepts:
  - Microservices
  - Continuous deployment
  - NoSQL
- Hot topics:
  - Docker
  - NewSQL
- . . .



http://matthewjamestaylor.com/img/recursive-drawing/pubic-hair.jpg

Keep learning

Johann Wolfgang von Goethe

What we do not understand we do not possess.

Keep learning

Johann Wolfgang von Goethe

What we do not understand we do not possess.

- Knowledge is power
- Better understanding
- Control over environment

Questions?