Big Data Processing Technologies

Chentao Wu
Associate Professor
Dept. of Computer Science and Engineering
wuct@cs.sjtu.edu.cn

上海交通大學
SHANGHAI JIAO TONG UNIVERSITY

Schedule

- lec1: Introduction on big data and cloud computing
- lec2: Introduction on data storage
- lec3: Data reliability (Replication/Archive/EC)
- lec4: Data consistency problem
- lec5: Block storage and file storage
- lec6: Object-based storage
- lec7: Distributed file system
- lec8: Metadata management

Collaborators

Google 谷歌

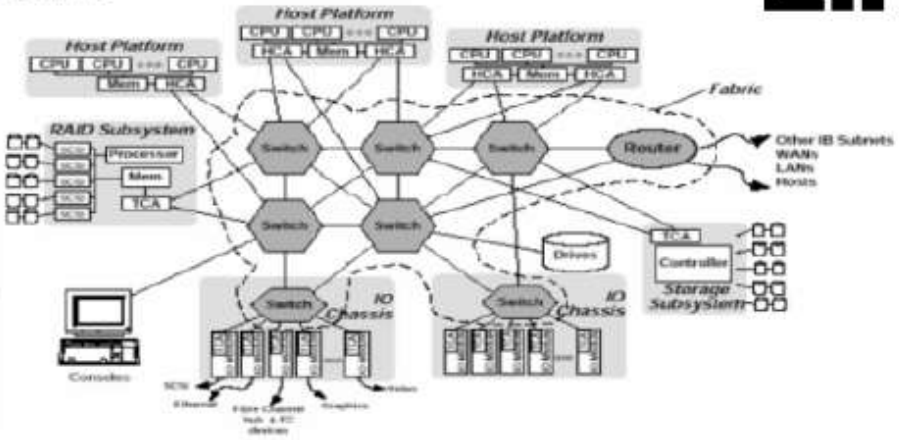
阿里云 aliyun.com

DELL EMC

Today's data share systems (1)

Microsoft

LinkedIn



Google

amazon.com

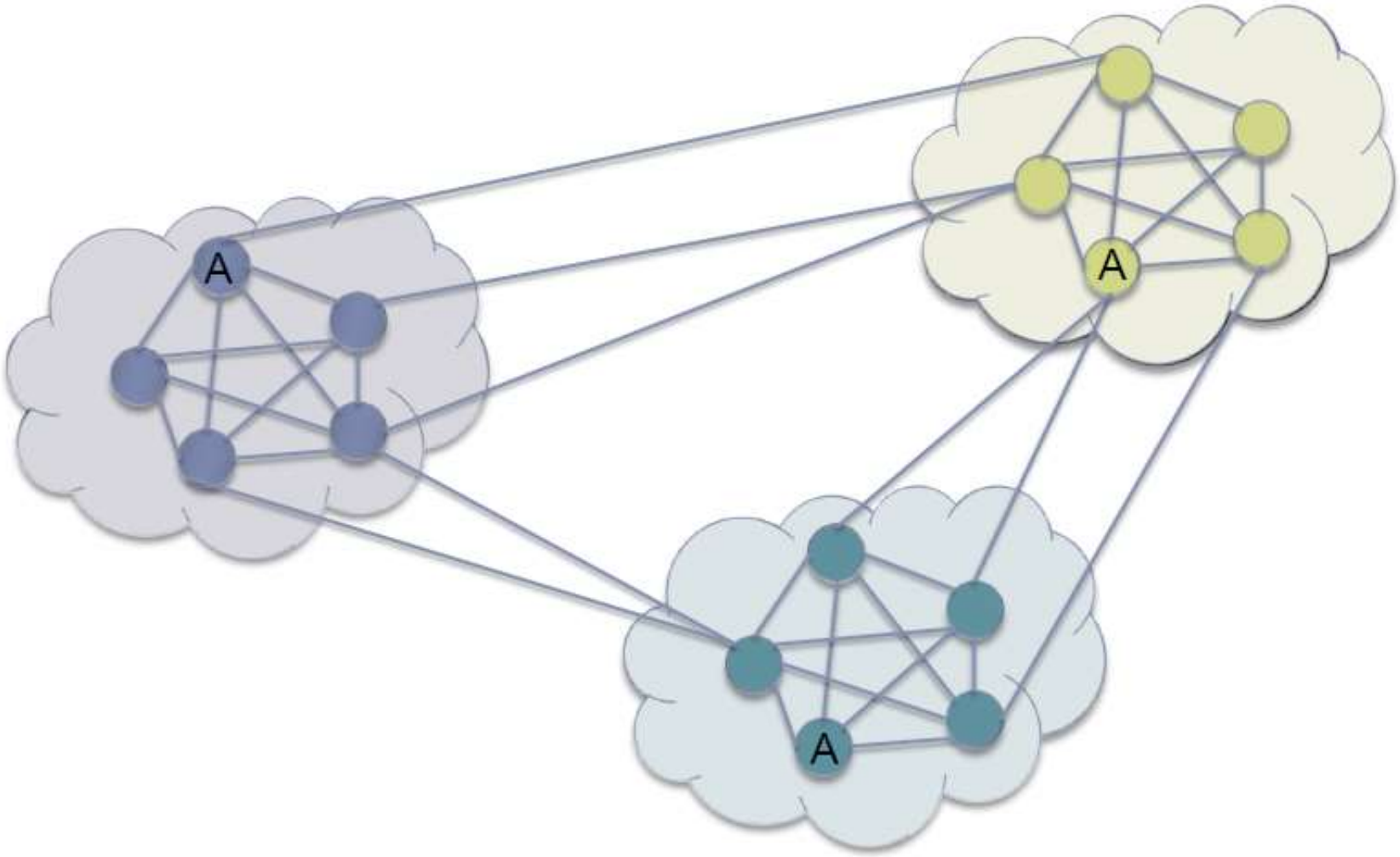
facebook

IBM

twitter

YAHOO!

Today's data share systems (2)



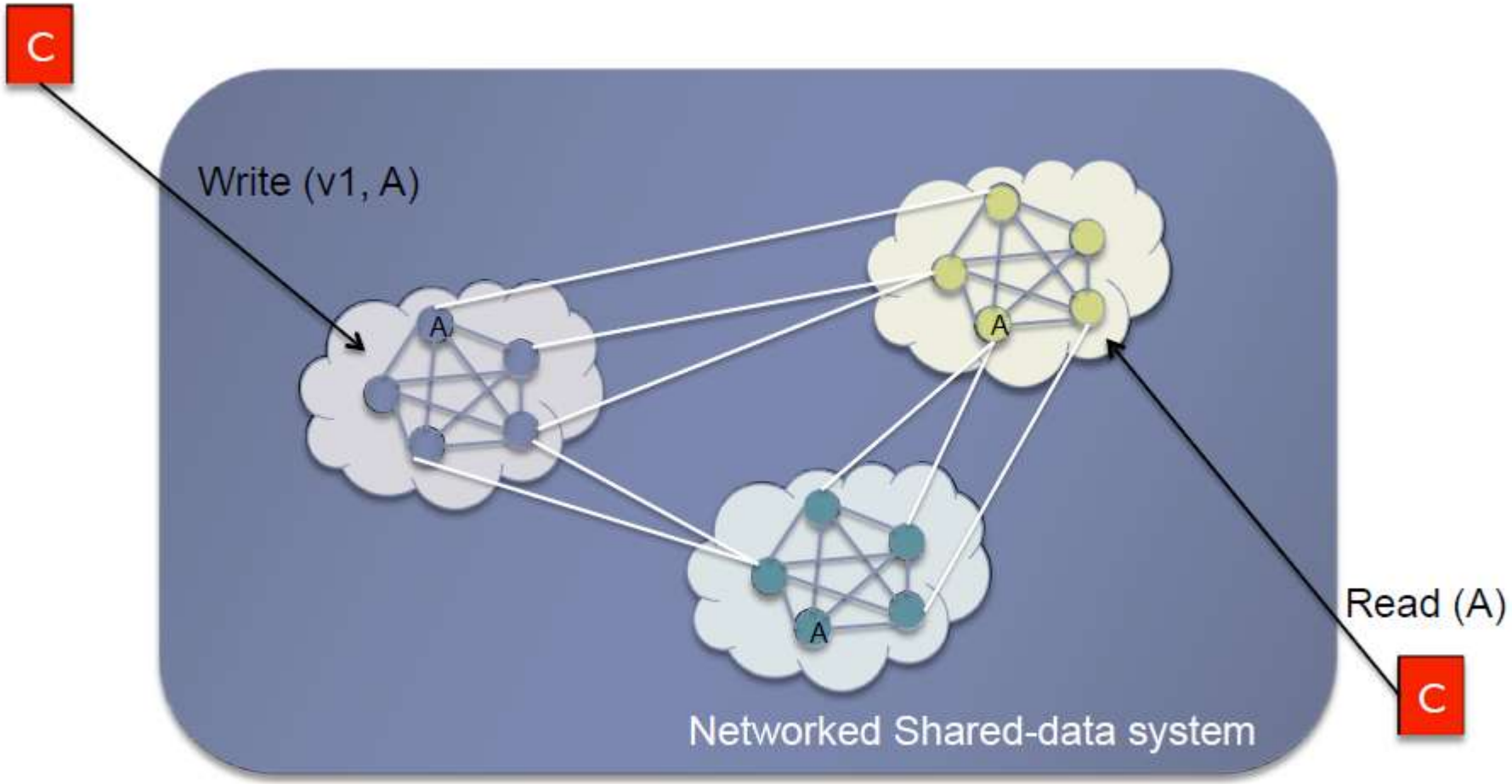
Fundamental Properties

- Consistency
  - (informally) “every request receives the right response”
  - E.g. If I get my shopping list on Amazon I expect it contains all the previously selected items
- Availability
  - (informally) “each request eventually receives a response”
  - E.g. eventually I access my shopping list
- tolerance to network Partitions
  - (informally) “servers can be partitioned in to multiple groups that cannot communicate with one other”

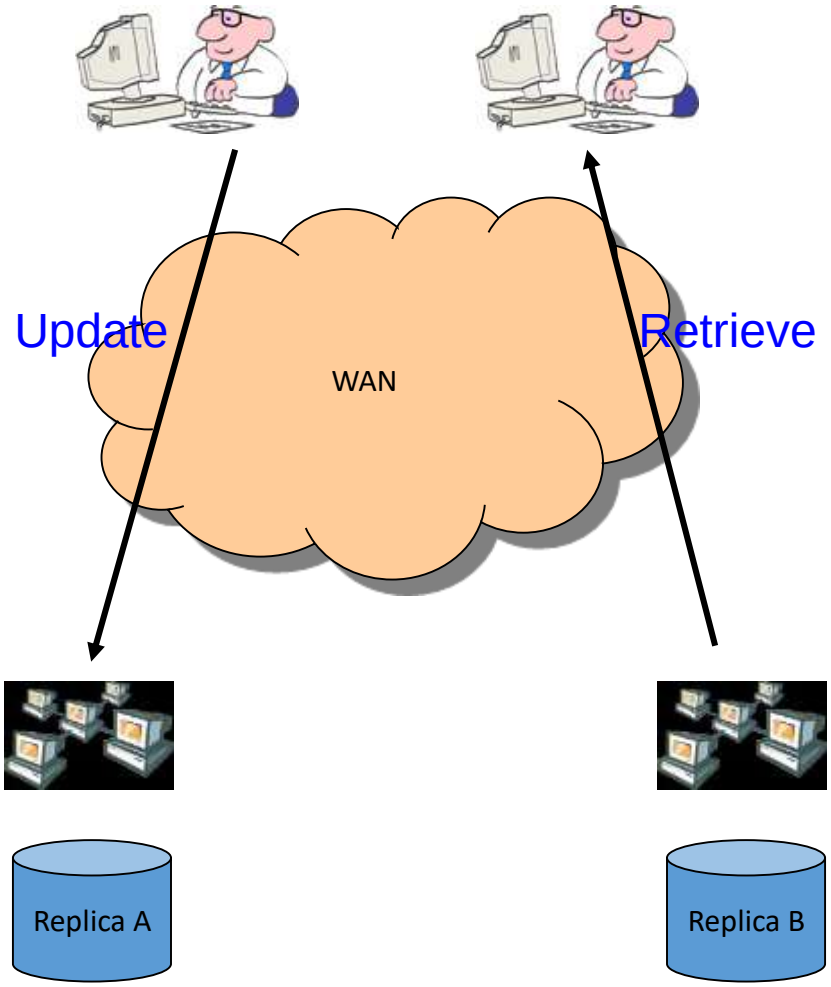
The CAP Theorem

- The CAP Theorem (Eric Brewer):
  - One can achieve at most two of the following:
    - Data Consistency
    - System Availability
    - Tolerance to network Partitions
  - Was first made as a conjecture At *PODC 2000 by* Eric Brewer
  - The Conjecture was formalized and confirmed by MIT researchers Seth Gilbert and Nancy Lynch in 2002

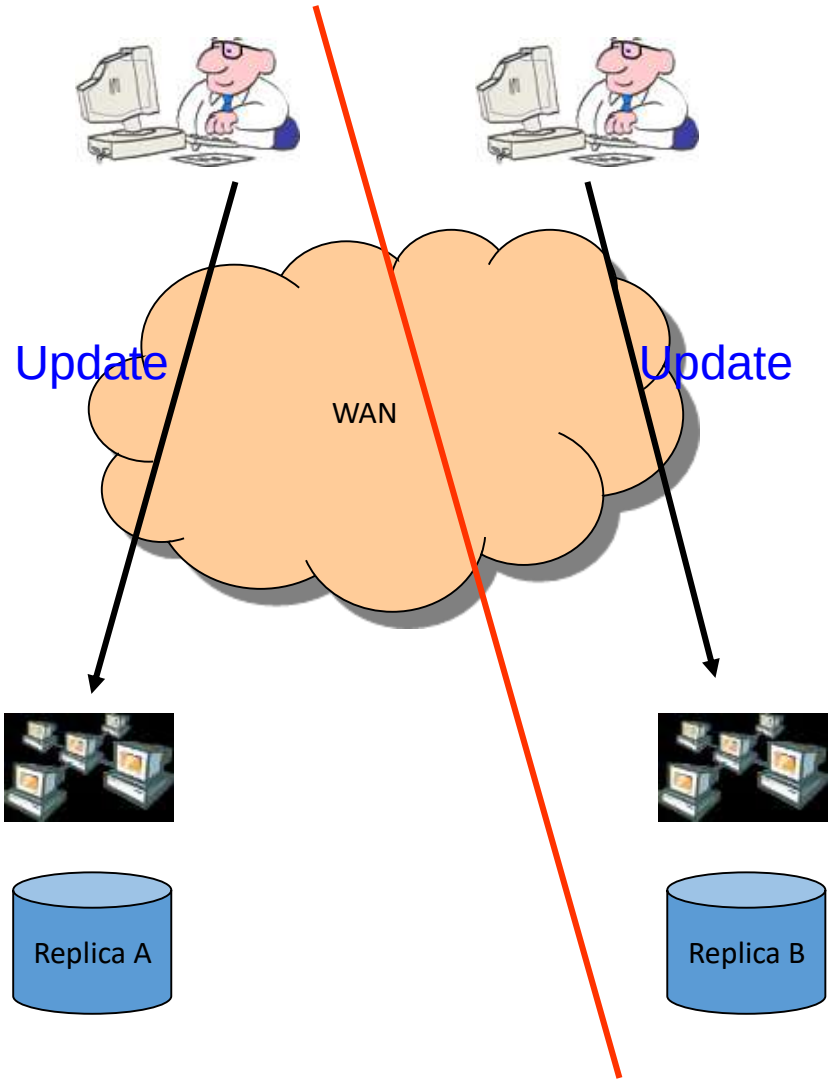
Proof



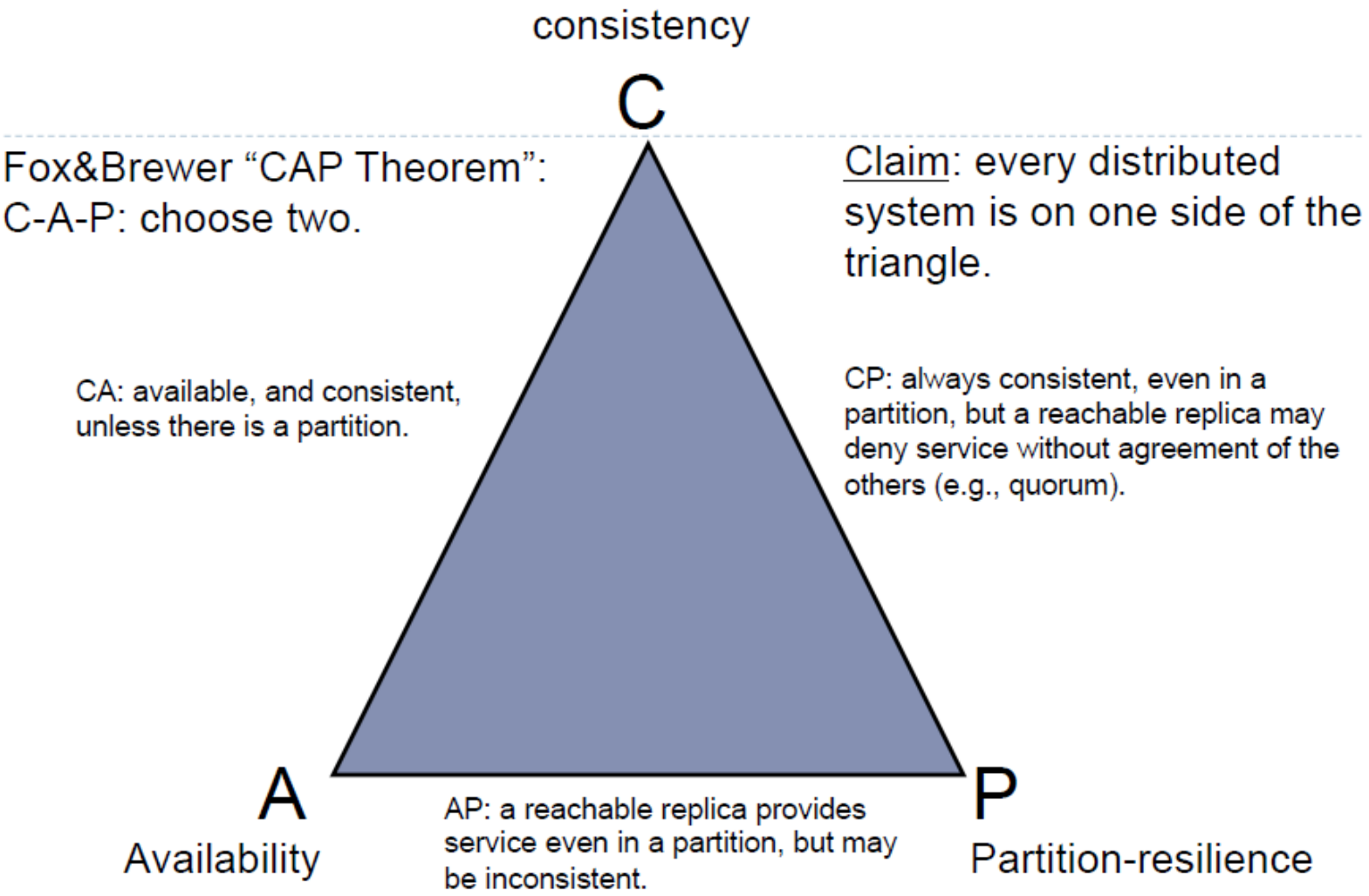
Consistency (Simplified)



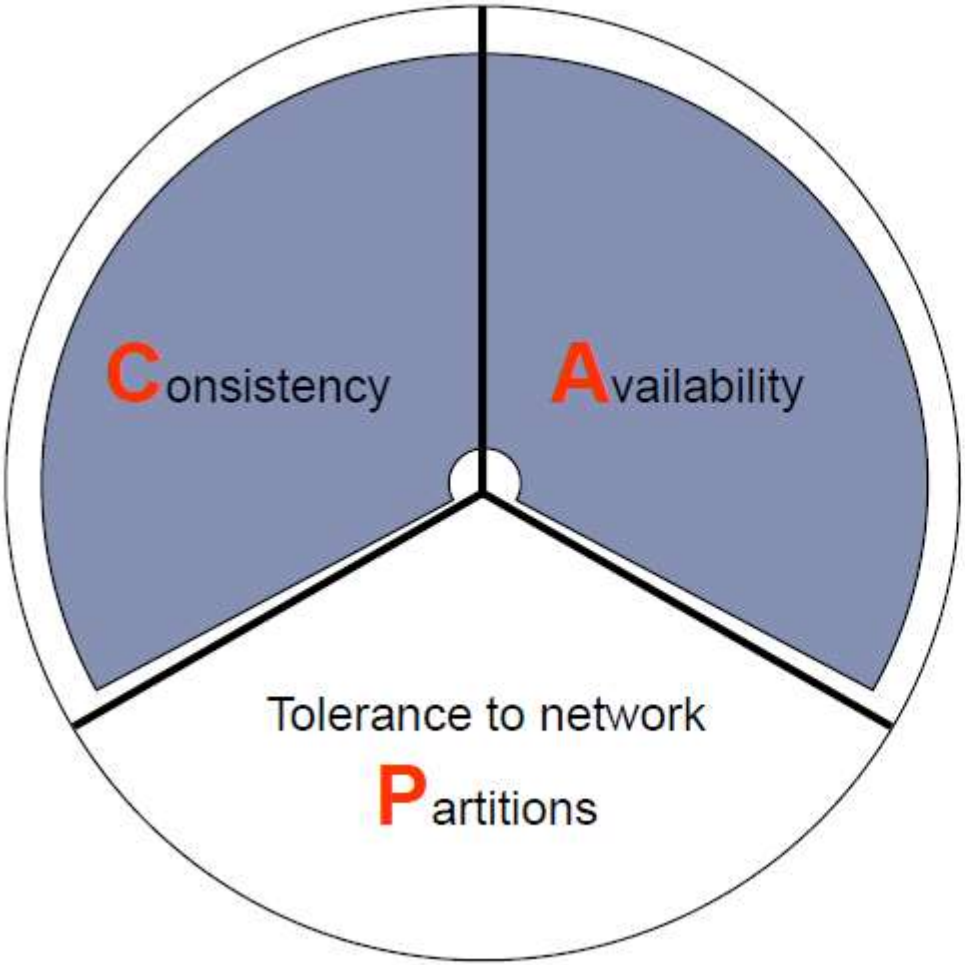
Tolerance to Network Partitions / Availability



CAP


consistency
C
Fox&Brewer "CAP Theorem":
C-A-P: choose two.
Claim: every distributed system is on one side of the triangle.
CA: available, and consistent, unless there is a partition.
CP: always consistent, even in a partition, but a reachable replica may deny service without agreement of the others (e.g., quorum).
A
Availability
AP: a reachable replica provides service even in a partition, but may be inconsistent.
P
Partition-resilience

Forfeit Partitions



Examples

- Single-site databases
- Cluster databases
- LDAP
- Fiefdoms

Traits

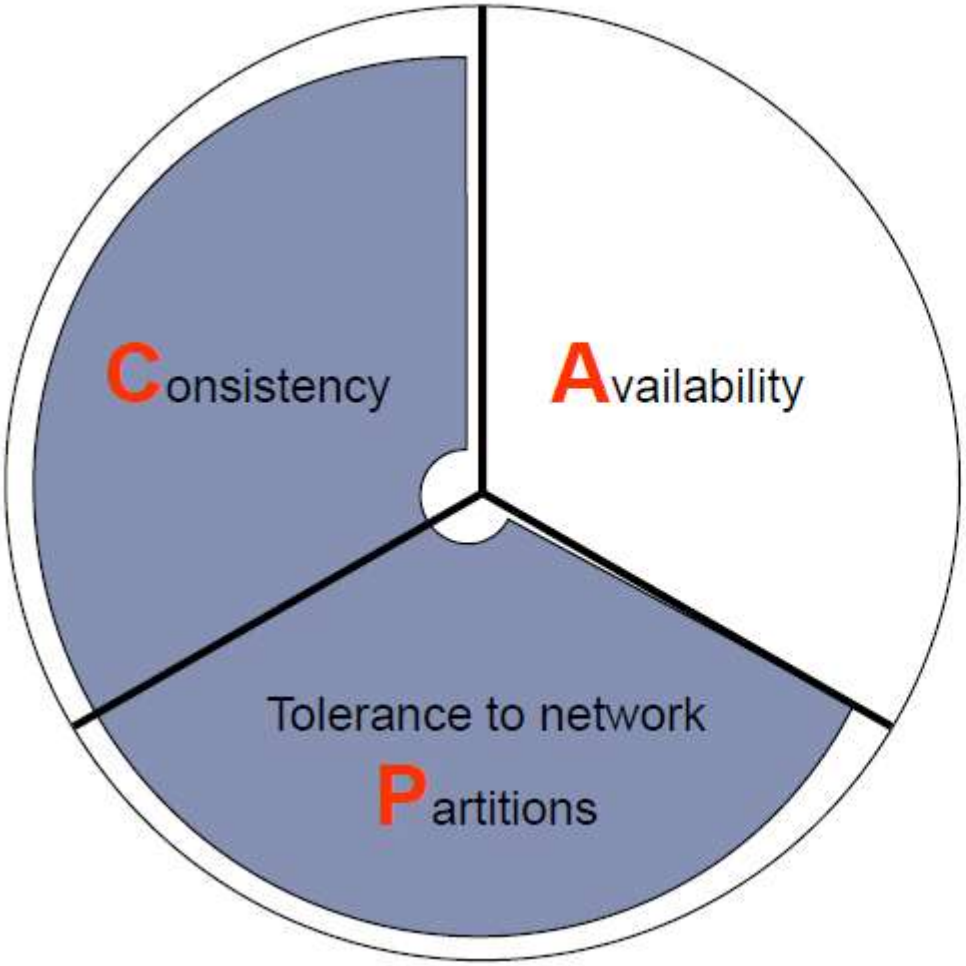
- 2-phase commit
- cache validation protocols
- The "inside"

Observations

- CAP states that in case of failures you can have at most two of these three properties for any shared-data system

- To scale out, you have to distribute resources.
  - P in not really an option but rather a need
  - The real selection is among consistency or availability
  - In almost all cases, you would choose availability over consistency

Forfeit Availability



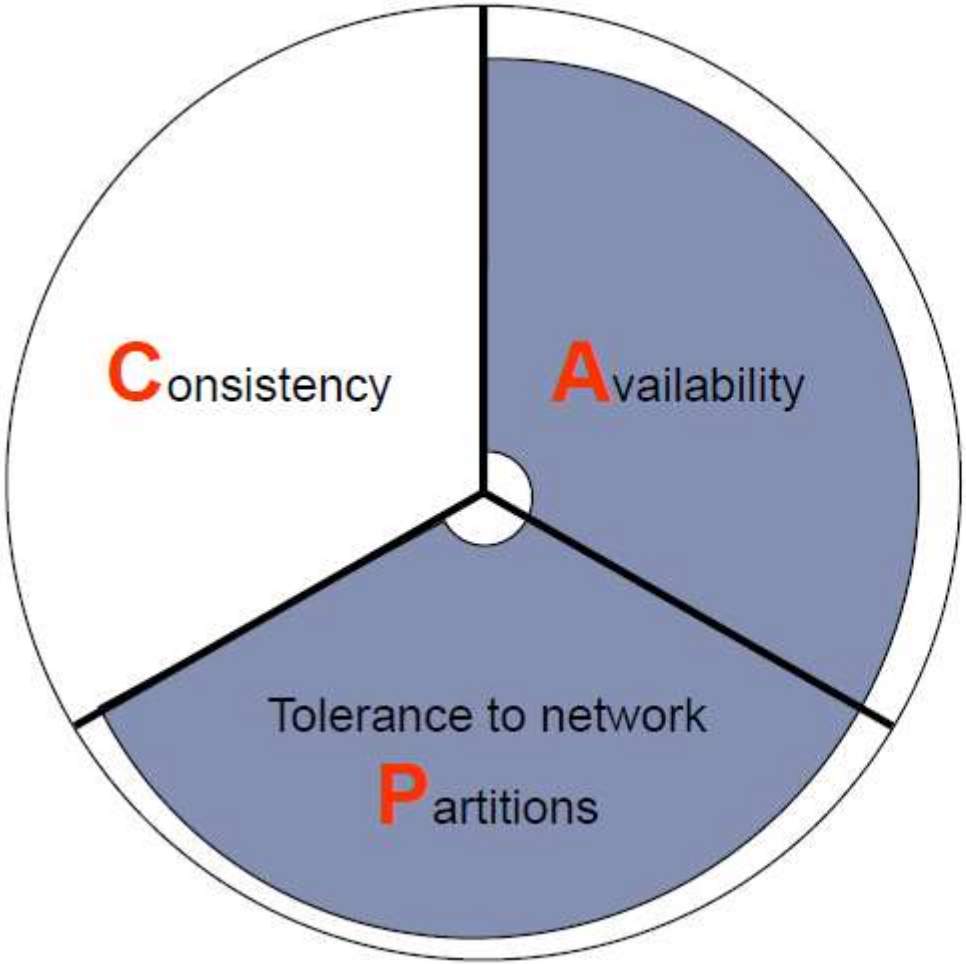
Examples

- Distributed databases
- Distributed locking
- Majority protocols

Traits

- Pessimistic locking
- Make minority partitions unavailable

Forfeit Consistency



Examples

- Coda
- Web caching
- DNS
- Emissaries

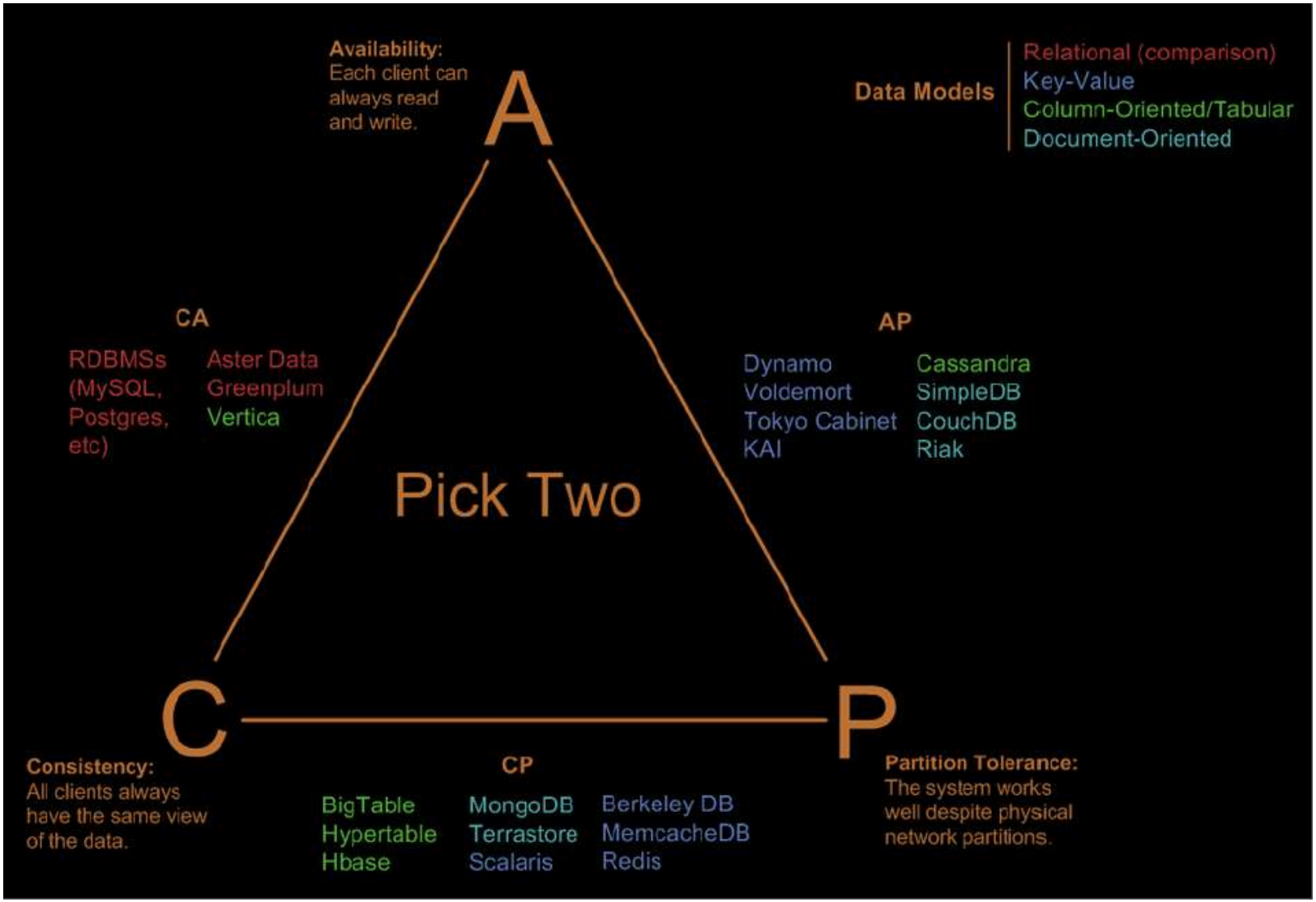
Traits

- expirations/leases
- conflict resolution
- Optimistic
- The "outside"

Consistency Boundary Summary

- We can have consistency & availability within a cluster.
  - No partitions within boundary!
- OS/Networking better at A than C
- Databases better at C than A
- Wide-area databases can't have both
- Disconnected clients can't have both

CAP in Database System



Another CAP -- BASE

- BASE stands for Basically Available Soft State Eventually Consistent system.
- Basically Available: the system available most of the time and there could exists a subsystems temporarily unavailable
- Soft State: data are “volatile” in the sense that their persistence is in the hand of the user that must take care of refresh them
- Eventually Consistent: the system eventually converge to a consistent state

Another CAP -- ACID

- Relation among ACID and CAP is core complex
- Atomicity: every operation is executed in “all-or-nothing” fashion
- Consistency: every transaction preserves the consistency constraints on data
- Integrity: transaction does not interfere. Every transaction is executed as it is the only one in the system
- Durability: after a commit, the updates made are permanent regardless possible failures

CAP vs. ACID

- CAP
- C here looks to single-copy consistency
- A here look to the service/data availability

- ACID
- C here looks to constraints on data and data model
- A looks to atomicity of operation and it is always ensured
- I is deeply related to CAP. I can be ensured in at most one partition
- D is independent from CAP

2 of 3 is misleading (1)

- In principle every system should be designed to ensure both C and A in normal situation
- When a partition occurs the decision among C and A can be taken
- When the partition is resolved the system takes corrective action coming back to work in normal situation

2 of 3 is misleading (2)

- Partitions are rare events
  - there are little reasons to forfeit by design C or A
- Systems evolve along time
  - Depending on the specific partition, service or data, the decision about the property to be sacrificed can change
- C, A and P are measured according to continuum
  - Several level of Consistency (e.g. ACID vs BASE)
  - Several level of Availability
  - Several degree of partition severity

Consistency/Latency Tradeoff (1)

- CAP does not force designers to give up A or C but why there exists a lot of systems trading C?

- CAP does not explicitly talk about latency…
- … however latency is crucial to get the essence of CAP

Consistency/Latency Tradeoff (2)

High Availability
- High Availability is a strong requirement of modern shared-data systems

Replication
- To achieve High Availability, data and services must be replicated

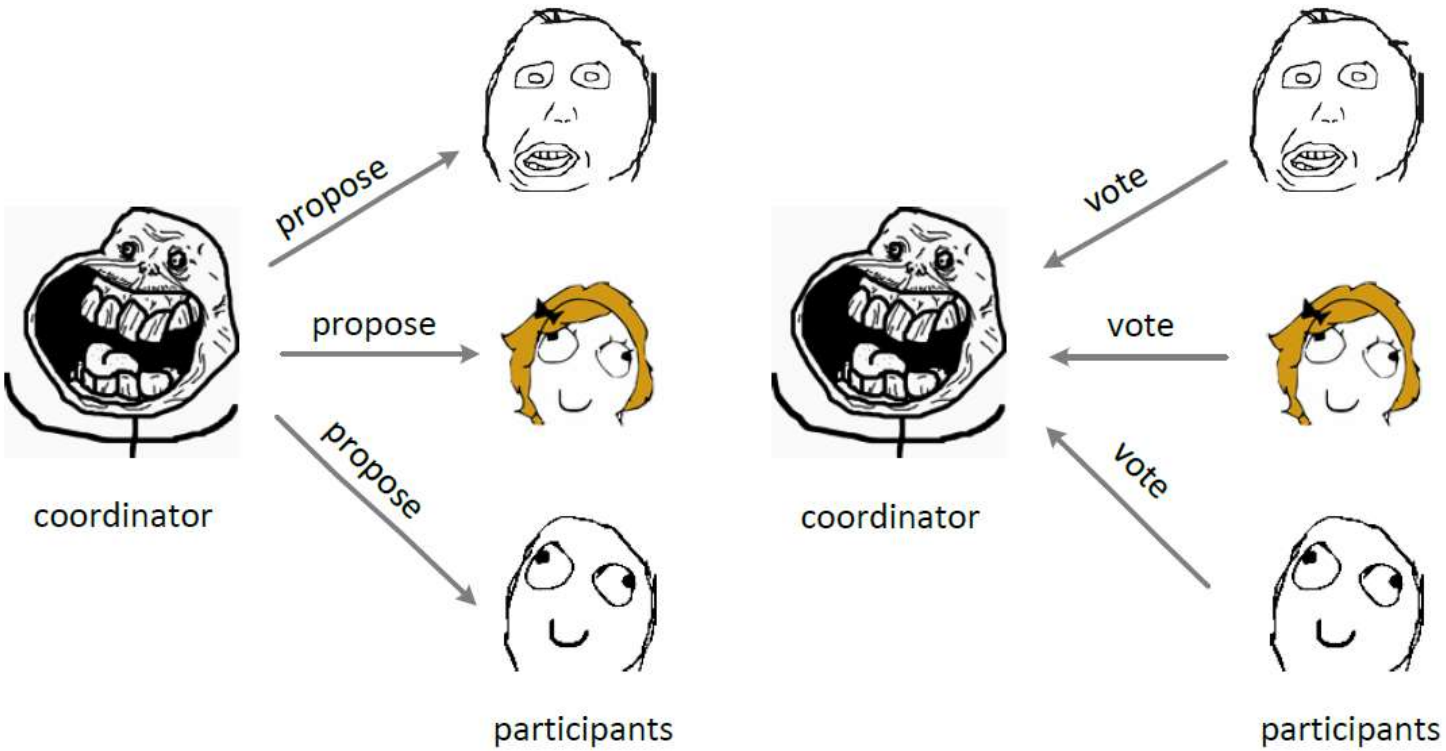
Consistency
- Replication impose consistency maintenance

Latency
- Every form of consistency requires communication and a stronger consistency requires higher latency

2 Consensus Protocol: 2PC and 3PC

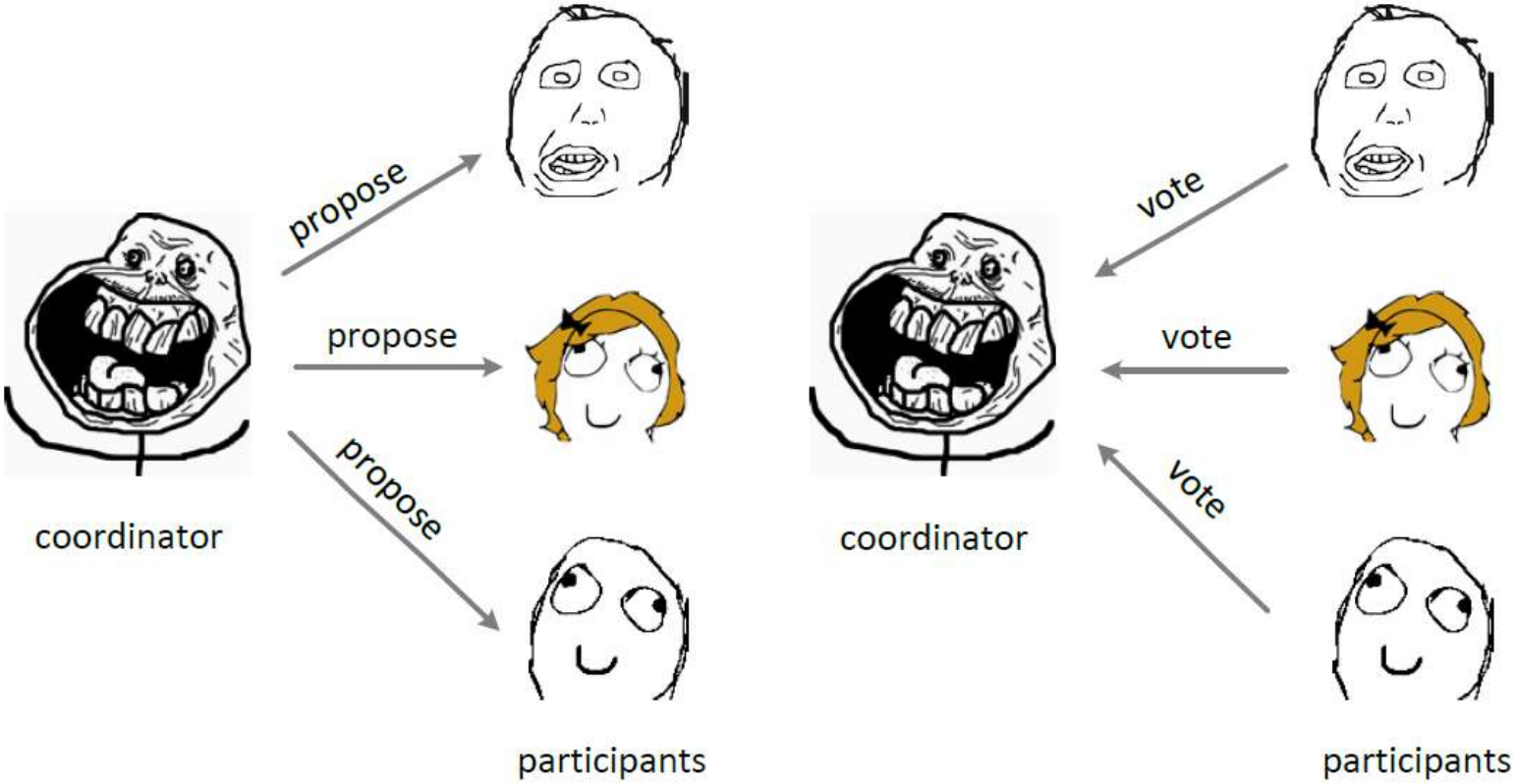
2PC: Two Phase Commit Protocol (1)

- Coordinator: propose a vote to other nodes
- Participants/Cohorts: send a vote to coordinator



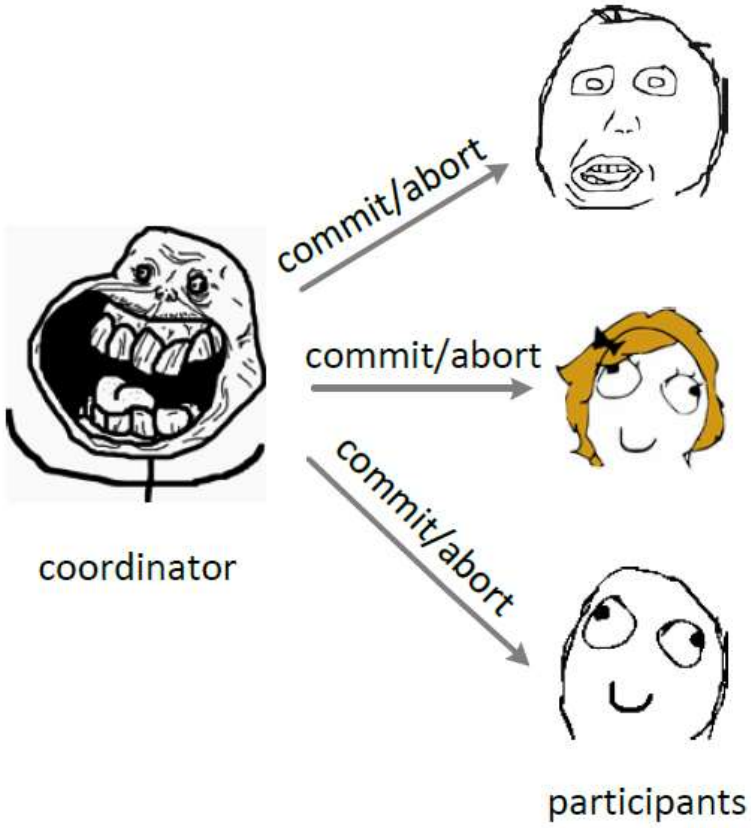
2PC: Phase one

- Coordinator propose a vote, and wait for the response of participants



2PC: Phase two

- Coordinator commits or aborts the transaction according to the participants' feedback
  - If all agree, commit
  - If any one disagree, abort



Problem of 2PC

- Scenario:

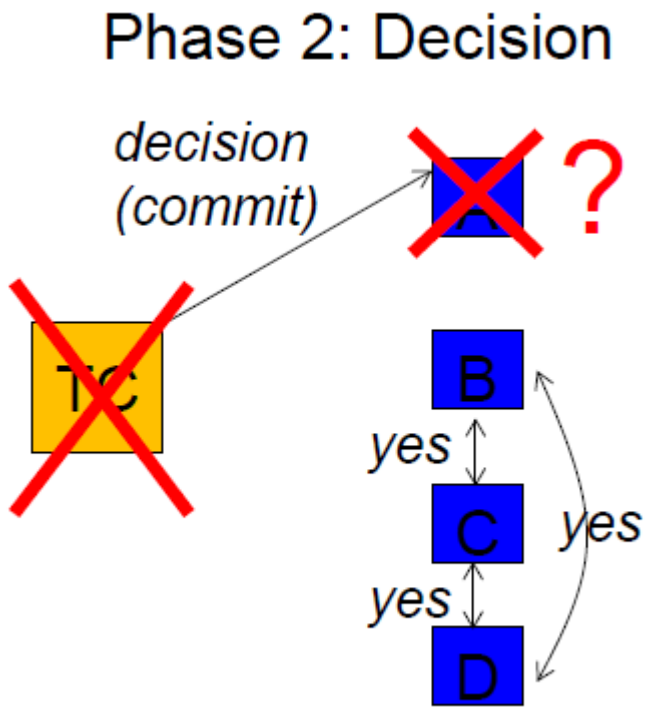
– TC sends commit decision to A, A gets it and commits, and then both TC and A crash

– B, C, D, who voted Yes, now need to wait for TC or A to reappear (w/ mutexes locked)

- They can't commit or abort, as they don't know what A responded

– If that takes a long time (e.g., a human must replace hardware), then availability suffers

– If TC is also participant, as it typically is, then this protocol is vulnerable to a single-node failure (the TC's failure)!

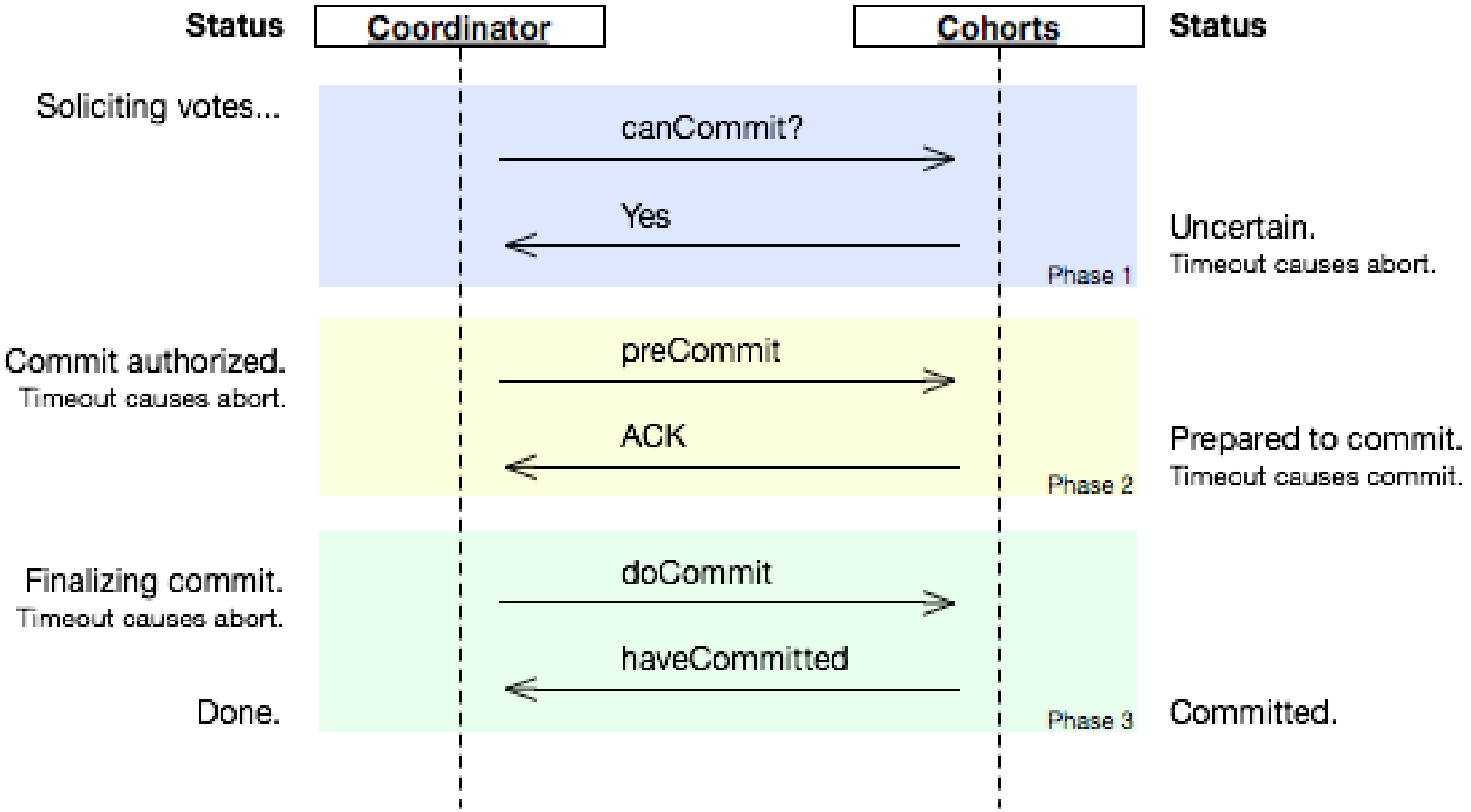


- This is why 2 phase commit is called a blocking protocol
- In context of consensus requirements: 2PC is safe, but not live

3PC: Three Phase Commit Protocol (1)

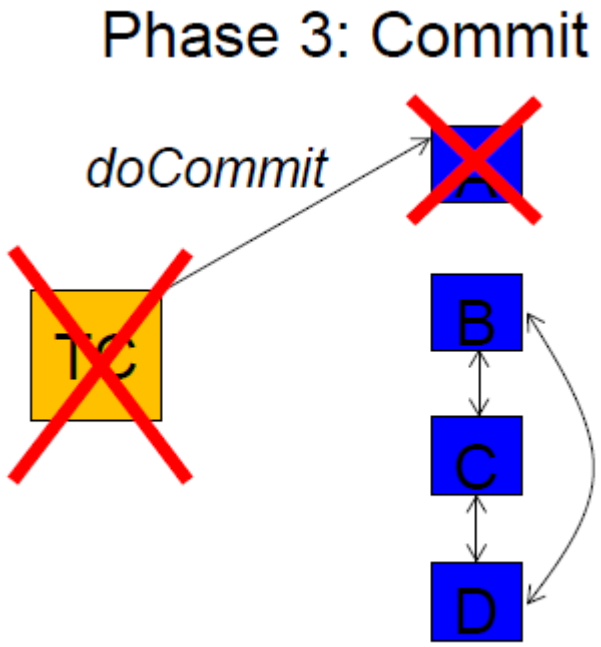
- Goal: Turn 2PC into a live (non-blocking) protocol
  - 3PC should never block on node failures as 2PC did
- Insight: 2PC suffers from allowing nodes to irreversibly commit an outcome before ensuring that the others know the outcome, too
- Idea in 3PC: split "commit/abort" phase into two phases
  - First communicate the outcome to everyone
  - Let them commit only after everyone knows the outcome

3PC: Three Phase Commit Protocol (2)



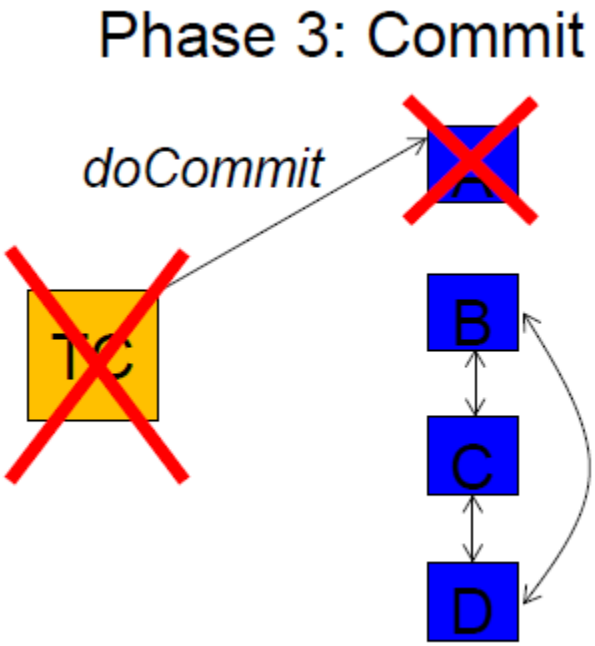
Can 3PC Solving the Blocking Problem? (1)

- Assuming same scenario as before (TC, A crash), can B/C/D reach a safe decision when they time out?
- 1. If one of them has received preCommit, …
- 2. If none of them has received preCommit, …



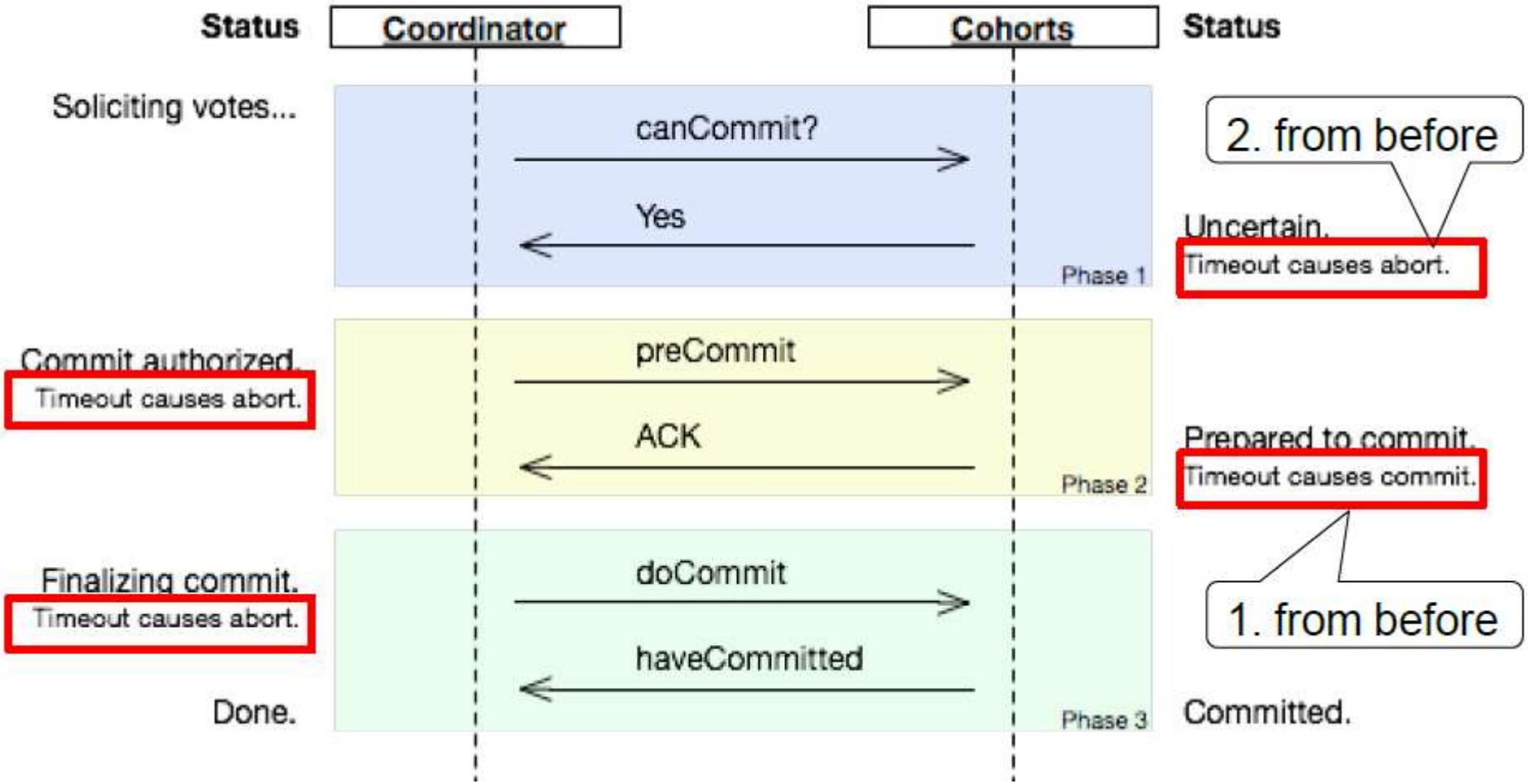
Can 3PC Solving the Blocking Problem? (2)

- Assuming same scenario as before (TC, A crash), can B/C/D reach a safe decision when they time out?
  - 1. If one of them has received preCommit, they can all commit
  - This is safe if we assume that A is DEAD and after coming back it runs a recovery protocol in which it requires input from B/C/D to complete an uncommitted transaction
  - This conclusion was impossible to reach for 2PC b/c A might have already committed and exposed outcome of transaction to world
  - 2. If none of them has received preCommit, they can all abort
  - This is safe, b/c we know A couldn't have received a doCommit, so it couldn't have committed



3PC is safe for node crashes (including TC+participant)

3PC: Timeout Handling Specs (trouble begins)

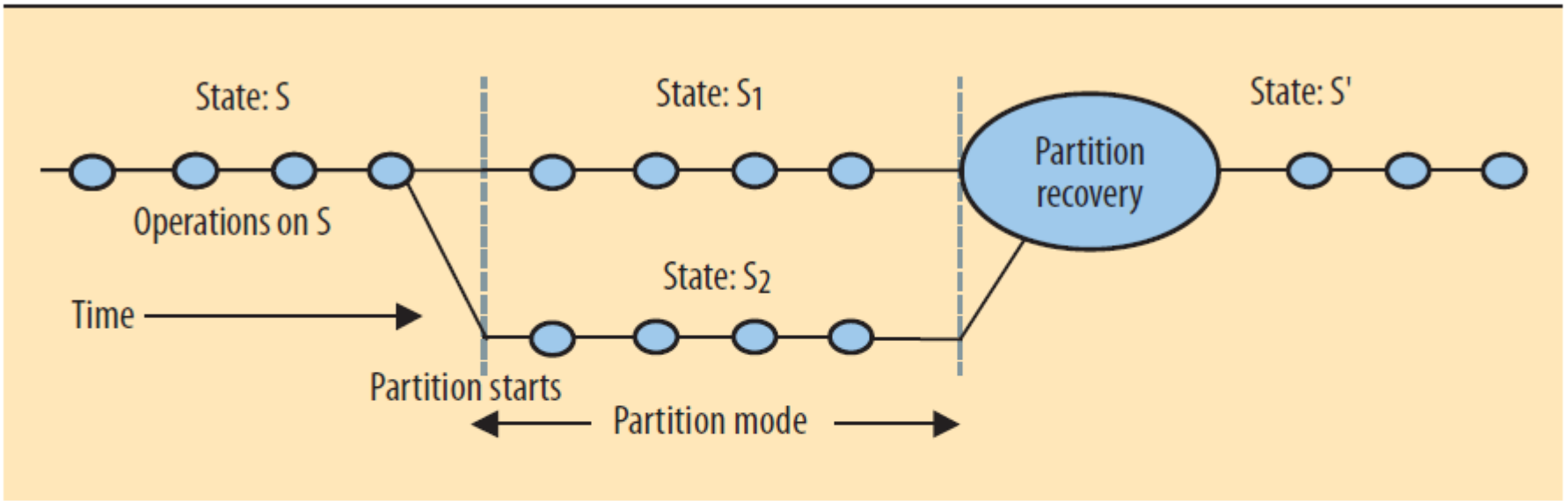


But Does 3PC Achieve Consensus?

- Liveness (availability): Yes
  - Doesn't block, it always makes progress by timing out
- Safety (correctness): No
  - Can you think of scenarios in which original 3PC would result in inconsistent states between the replicas?
- Two examples of unsafety in 3PC:
  - A hasn't crashed, it's just offline
  - TC hasn't crashed, it's just offline

} Network Partitions

Partition Management



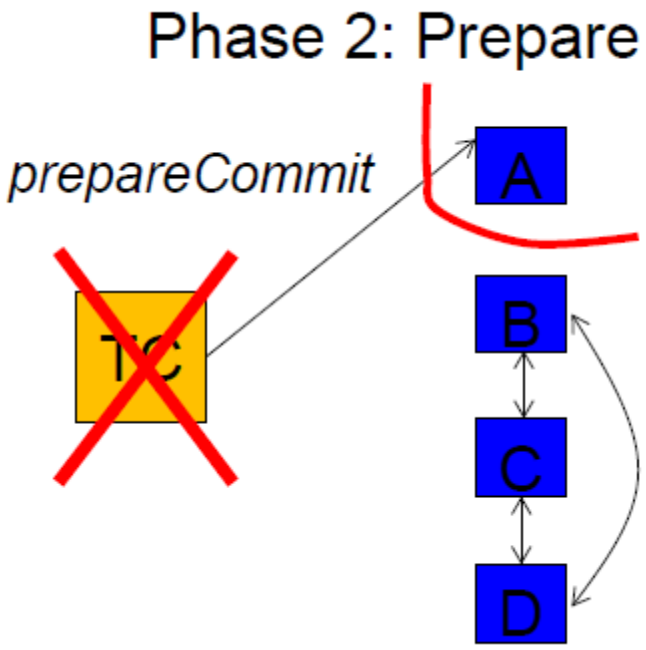
Partition Detection

Activating Partition Mode

Partition Recovery

3PC with Network Partitions

- One example scenario:
  - A receives prepareCommit from TC
  - Then, A gets partitioned from B/C/D and TC crashes
  - None of B/C/D have received prepareCommit, hence they all abort upon timeout
  - A is prepared to commit, hence, according to protocol, after it times out, it unilaterally decides to commit



- Similar scenario with partitioned, not crashed, TC

Safety vs. liveness

- So, 3PC is doomed for network partitions
  - The way to think about it is that this protocol's design trades safety for liveness
- Remember that 2PC traded liveness for safety
- Can we design a protocol that's both safe and live?

3

Paxos

Paxos (1)

- The only known completely-safe and largely-live agreement protocol
- Lets all nodes agree on the same value despite node failures, network failures, and delays

  – Only blocks in exceptional circumstances that are vanishingly rare in practice

- Extremely useful, e.g.:

  – nodes agree that client X gets a lock

  – nodes agree that Y is the primary

  – nodes agree that Z should be the next operation to be executed

Paxos (2)

- Widely used in both industry and academia
- Examples:
  - **Google**: Chubby (Paxos-based distributed lock service)

    Most Google services use Chubby directly or indirectly
  - **Yahoo**: Zookeeper (Paxos-based distributed lock service)

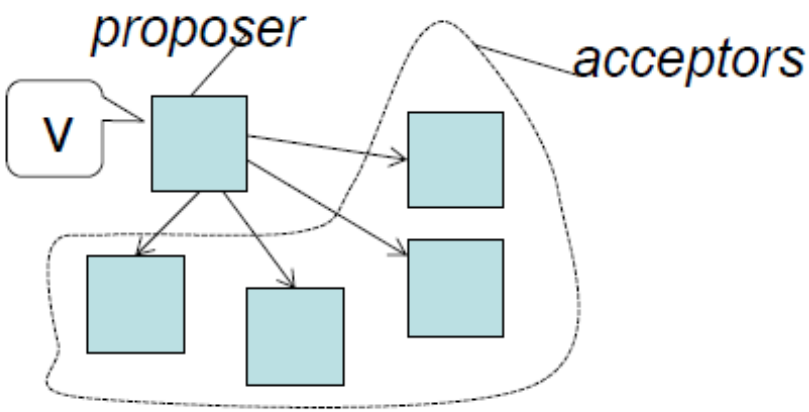
    In Hadoop rightnow
  - **MSR**: Frangipani (Paxos-based distributed lock service)
  - **UW**: Scatter (Paxos-based consistent DHT)
  - Open source:
    - libpaxos (Paxos-based atomic broadcast)
    - Zookeeper is open-source and integrates with Hadoop

Paxos Properties

- Safety
  - If agreement is reached, everyone agrees on the same value
  - The value agreed upon was proposed by some node
- Fault tolerance (i.e., as-good-as-it-gets liveness)
  - If less than half the nodes fail, the rest nodes reach agreement *eventually*
- No guaranteed termination (i.e., imperfect liveness)
  - Paxos may not always converge on a value, but only in very degenerate cases that are improbable in the real world
- Lots of awesomeness
  - Basic idea seems natural in retrospect, but why it works in any detail is incredibly complex!

Basic Idea (1)

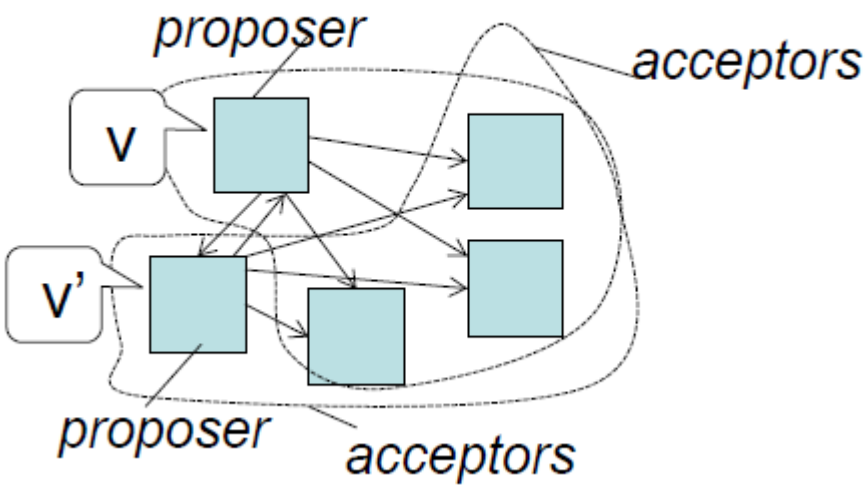
- Paxos is similar to 2PC, but with some twists
- One (or more) node decides to be coordinator (*proposer*)
- Proposer proposes a value and solicits acceptance from others (*acceptors*)
- Proposer announces the chosen value or tries again if it's failed to converge on a value



- Values to agree on:
- Whether to commit/abort a transaction
- Which client should get the next lock
- Which write we perform next
- What time to meet (party example)

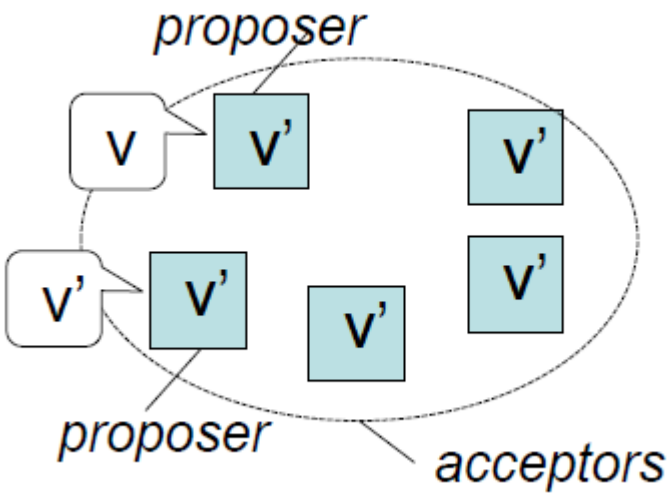
Basic Idea (2)

- Paxos is similar to 2PC, but with some twists
- One (or more) node decides to be coordinator (*proposer*)
- Proposer proposes a value and solicits acceptance from others (*acceptors*)
- Proposer announces the chosen value or tries again if it's failed to converge on a value



Basic Idea (3)

- Paxos is similar to 2PC, but with some twists
- One (or more) node decides to be coordinator (*proposer*)
- Proposer proposes a value and solicits acceptance from others (*acceptors*)
- Proposer announces the chosen value or tries again if it's failed to converge on a value



- Hence, Paxos is egalitarian: any node can propose/accept, no one has special powers
- Just like real world, e.g., group of friends organize a party – anyone can take the lead

Challenges

- What if multiple nodes become proposers simultaneously?
- What if the new proposer proposes different values than an already decided value?
- What if there is a network partition?
- What if a proposer crashes in the middle of solicitation?
- What if a proposer crashes after deciding but before announcing results?

Core Differentiating Mechanisms

1. Proposal ordering

– Lets nodes decide which of several concurrent proposals to accept and which to reject

2. Majority voting

– 2PC needs all nodes to vote Yes before committing

- As a result, 2PC may block when a single node fails

– Paxos requires only a majority of the acceptors (half+1) to accept a proposal

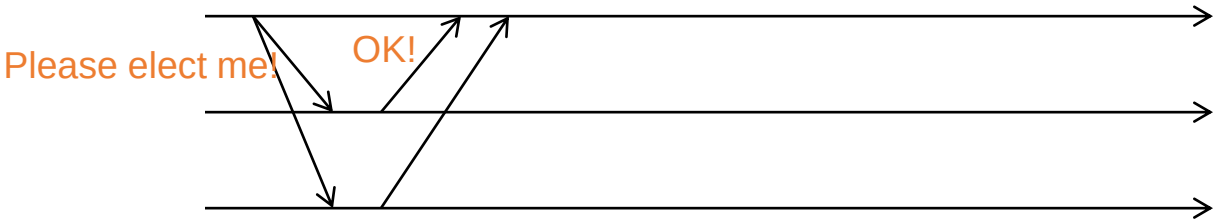
- As a result, in Paxos nearly half the nodes can fail to reply and the protocol continues to work correctly
- Moreover, since no two majorities can exist simultaneously, network partitions do not cause problems (as they did for 3PC)

Implementation of Paxos

- Paxos has rounds; each round has a unique ballot id
- Rounds are asynchronous
  - Time synchronization not required
  - If you're in round *j* and hear a message from round *j+1*, abort everything and move over to round *j+1*
  - Use timeouts; may be pessimistic
- Each round itself broken into phases (which are also asynchronous)
  - Phase 1: A leader is elected (Election)
  - Phase 2: Leader proposes a value, processes ack (Bill)
  - Phase 3: Leader multicasts final value (Law)

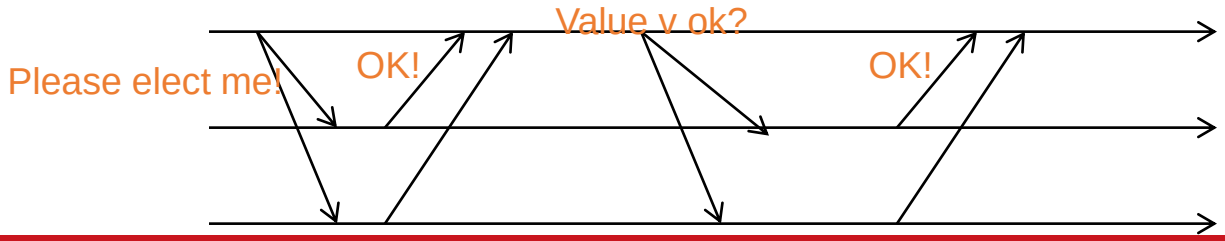
Phase 1 – Election

- Potential leader chooses a unique ballot id, higher than seen anything so far
- Sends to all processes
- Processes wait, respond once to highest ballot id
  - If potential leader sees a higher ballot id, it can't be a leader
  - Paxos tolerant to multiple leaders, but we'll only discuss 1 leader case
  - Processes also log received ballot ID on disk
- If a process has in a previous round decided on a value v', it includes value v' in its response
- If majority (i.e., quorum) respond OK then you are the leader
  - If no one has majority, start new round
- A round cannot have two leaders (why?)



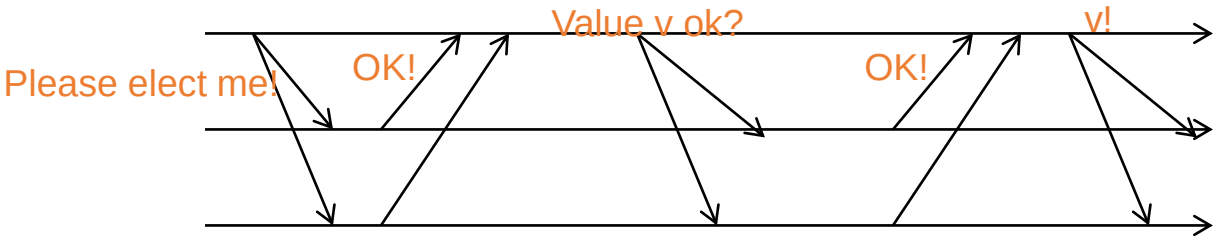
Phase 2 – Proposal (Bill)

- Leader sends proposed value v to all
  - use v=v’ if some process already decided in a previous round and sent you its decided value v’
- Recipient logs on disk; responds OK



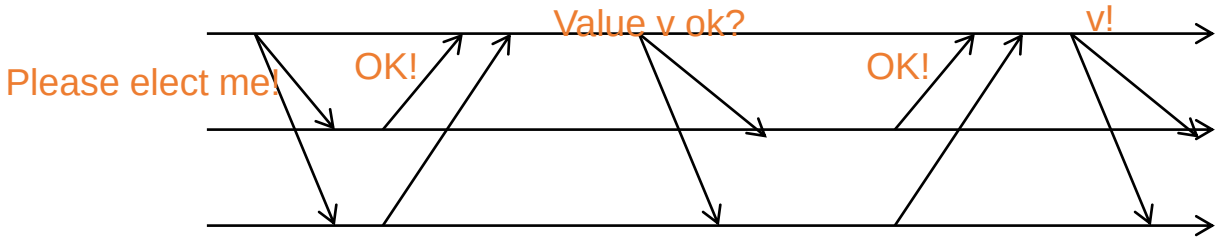
Phase 3 – Decision (Law)

- If leader hears a majority of OKs, it lets everyone know of the decision
- Recipients receive decision, log it on disk



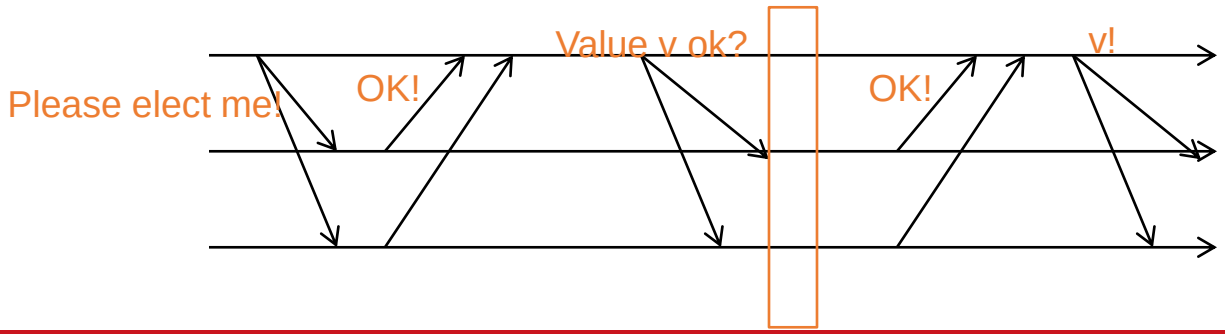
Which is the point of no-return? (1)

- That is, when is consensus reached in the system



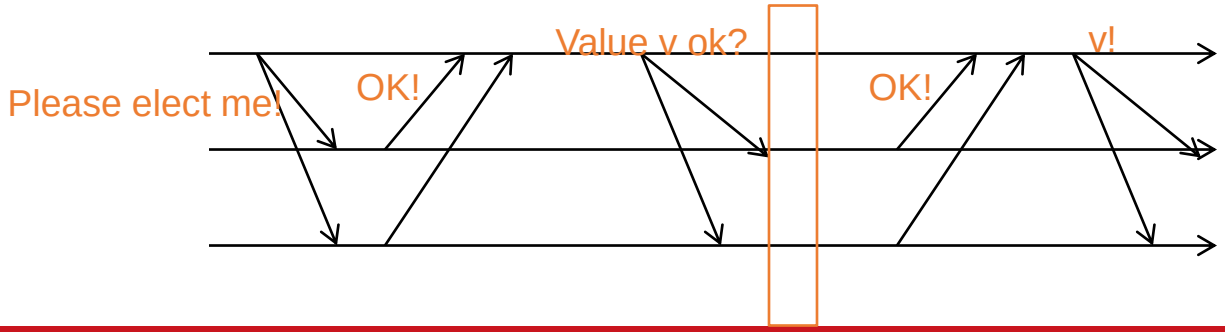
Which is the point of no-return? (2)

- If/when a majority of processes hear proposed value and accept it (i.e., are about to/have respond(ed) with an OK!)
- Processes *may not know it yet*, but a decision has been made for the group
  - Even leader does not know it yet
- What if leader fails after that?
  - Keep having rounds until some round completes



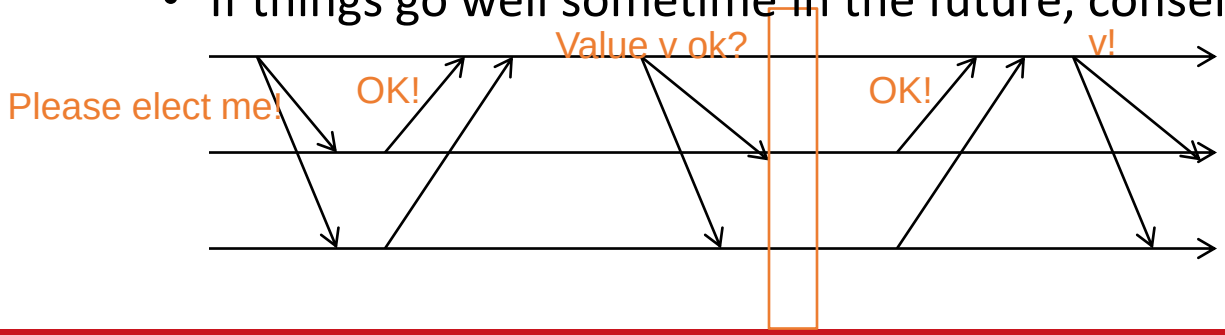
Safety

- If some round has a majority (i.e., quorum) hearing proposed value v’ and accepting it (middle of Phase 2), then subsequently at each round either: 1) the round chooses v’ as decision or 2) the round fails
- Proof:
  - Potential leader waits for majority of OKs in Phase 1
  - At least one will contain v’ (because two majorities or quorums always intersect)
  - It will choose to send out v’ in Phase 2
- Success requires a majority, and any two majority sets intersect



What could go wrong?

- Process fails
  - Majority does not include it
  - When process restarts, it uses log to retrieve a past decision (if any) and past-seen ballot ids. Tries to know of past decisions.
- Leader fails
  - Start another round
- Messages dropped
  - If too flaky, just start another round
- Note that anyone can start a round any time
- Protocol may never end – tough luck, buddy!
  - Impossibility result not violated
  - If things go well sometime in the future, consensus reached



4 Chubby and Zookeeper

Google Chubby

- Research Paper
  - The Chubby Lock Service for Loosely-coupled Distributed Systems. Proc. of OSDI'06.
- What is Chubby?
  - Lock service in a loosely-coupled distributed system (e.g., 10K 4-processor machines connected by 1Gbps Ethernet)
  - Client interface similar to whole-file advisory locks with notification of various events (e.g., file modifications)
  - Primary goals: reliability, availability, easy-to-understand semantics
- How is it used?
  - Used in Google: GFS, Bigtable, etc.
  - Elect leaders, store small amount of meta-data, as the root of the distributed data structures

System Structure (1)

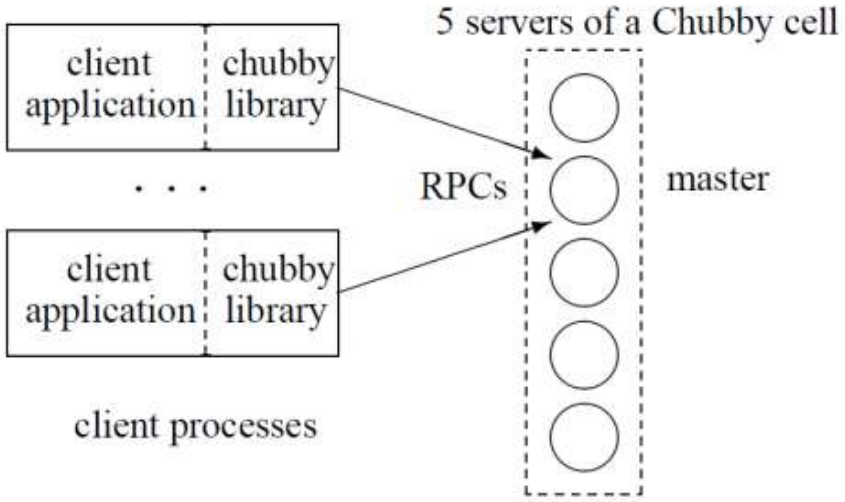


Figure 1: System structure

- A chubby cell consists of a small set of servers (replicas)
- A master is elected from the replicas via a consensus protocol
  - Master lease: several seconds
  - If a master fails, a new one will be elected when the master leases expire
- Client talks to the master via chubby library
  - All replicas are listed in DNS; clients discover the master by talking to any replica

System Structure (2)

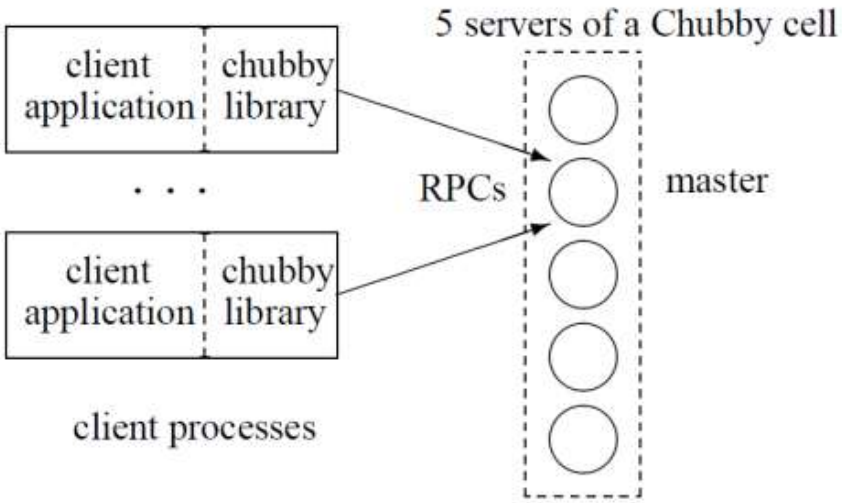


Figure 1: System structure

- Replicas maintain copies of a simple database
- Clients send read/write requests only to the master
- For a write:
  - The master propagates it to replicas via the consensus protocol
  - Replies after the write reaches a majority of replicas
- For a read:
  - The master satisfies the read alone

System Structure (3)

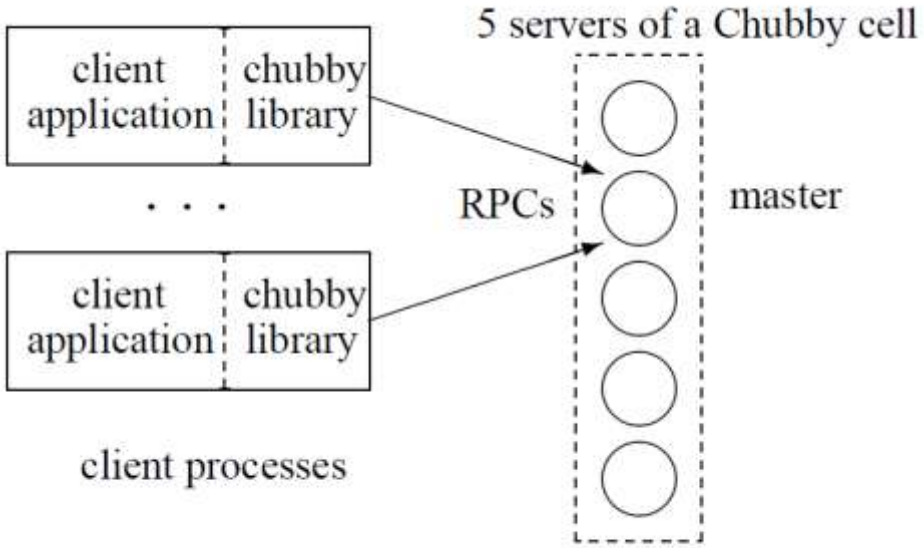


Figure 1: System structure

- If a replica fails and does not recover for a long time (a few hours)
  - A fresh machine is selected to be a new replica, replacing the failed one
  - It updates the DNS
  - Obtains a recent copy of the database
  - The current master polls DNS periodically to discover new replicas

Simple UNIX-like File System Interface

- Chubby supports a strict tree of files and directories
  - No symbolic links, no hard links
  - /ls/foo/wombat/pouch
    - 1st component (ls): lock service (common to all names)
    - 2nd component (foo): the chubby cell (used in DNS lookup to find the cell master)
    - The rest: name inside the cell
  - Can be accessed via Chubby's specialized API / other file system interface (e.g., GFS)
- Support most normal operations (create, delete, open, write, …)
- Support advisory reader/writer lock on a node

ACLs and File Handles

- Access Control List (ACL)
  - A node has three ACL names (read/write/change ACL names)
  - An ACL name is a name to a file in the ACL directory
  - The file lists the authorized users
- File handle:
  - Has check digits encoded in it; cannot be forged
  - Sequence number:
    - a master can tell if this handle is created by a previous master
  - Mode information at open time:
    - If previous master created the handle, a newly restarted master can learn the mode information

Locks and Sequences

- Locks: advisory rather than mandatory
- Potential lock problems in distributed systems
  - A holds a lock L, issues request W, then fails
  - B acquires L (because A fails), performs actions
  - W arrives (out-of-order) after B's actions
- Solution #1: backward compatible
  - Lock server will prevent other clients from getting the lock if a lock become inaccessible or the holder has failed
  - Lock-delay period can be specified by clients
- Solution #2: sequencer
  - A lock holder can obtain a sequencer from Chubby
  - It attaches the sequencer to any requests that it sends to other servers (e.g., Bigtable)
  - The other servers can verify the sequencer information

Chubby Events

- Clients can subscribe to events (up-calls from Chubby library)
  - File contents modified: if the file contains the location of a service, this event can be used to monitor the service location
  - Master failed over
  - Child node added, removed, modified
  - Handle becomes invalid: probably communication problem
  - Lock acquired (rarely used)
  - Locks are conflicting (rarely used)

APIs

- Open()
  - Mode: read/write/change ACL; Events; Lock-delay
  - Create new file or directory?
- Close()
- GetContentsAndStat(), GetStat(), ReadDir()
- SetContents(): set all contents; SetACL()
- Delete()
- Locks: Acquire(), TryAcquire(), Release()
- Sequencers: GetSequencer(), SetSequencer(), CheckSequencer()

Example – Primary Election

```
Open("write mode");
If (successful) {
   // primary
   SetContents("identity");
}
Else {
   // replica
   open ("read mode", "file-modification event");
   when notified of file modification:
         primary= GetContentsAndStat();
}
```

Caching

- Strict consistency: easy to understand
  - Lease based
  - master will invalidate cached copies upon a write request
- Write-through caches

Sessions, Keep-Alives, Master Fail-overs (1)

- Session:
  - A client sends keep-alive requests to a master
  - A master responds by a keep-alive response
  - Immediately after getting the keep-alive response, the client sends another request for extension
  - The master will block keep-alives until close the expiration of a session
  - Extension is default to 12s
- Clients maintain a local timer for estimating the session timeouts (time is not perfectly synchronized)
- If local timer runs out, wait for a 45s grace period before ending the session
  - Happens when a master fails over

Sessions, Keep-Alives, Master Fail-overs (2)

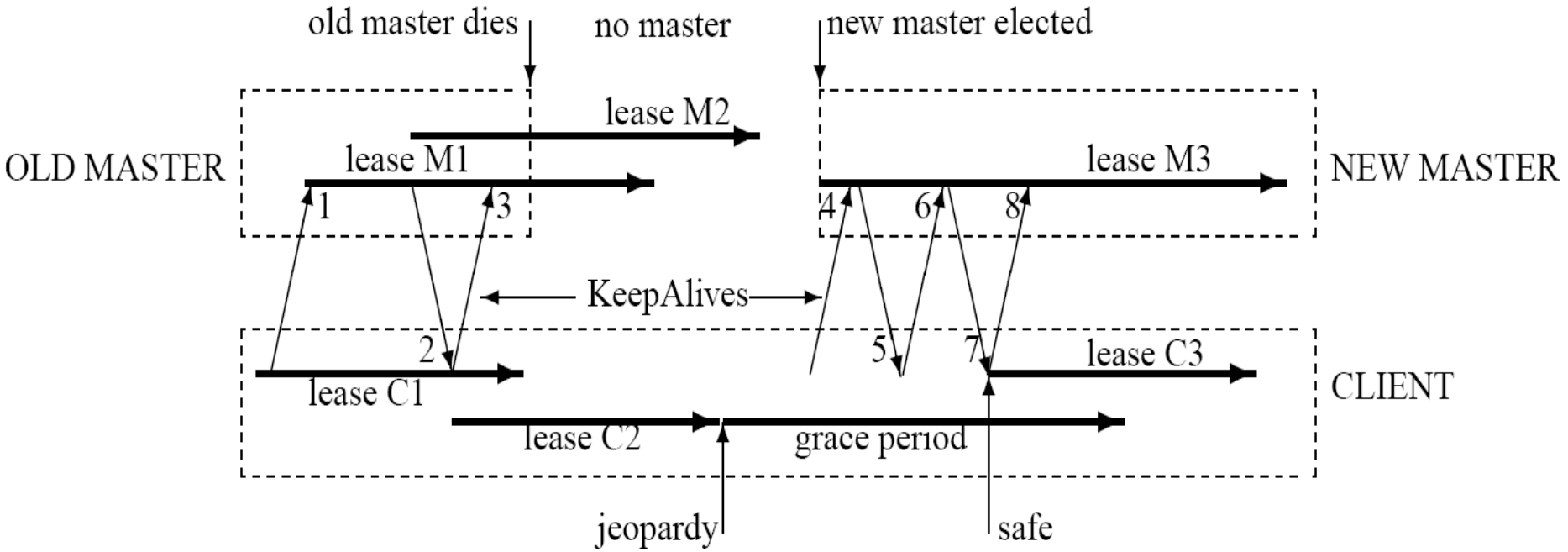


Figure 2: The role of the grace period in master fail-over

Other details

- Database implementation
  - a simple database with write ahead logging and snapshotting
- Backup:
  - Write a snapshot to a GFS server in a different building
- Mirroring files across multiple cells
  - Configuration files (e.g., locations of other services, access control lists, etc.)

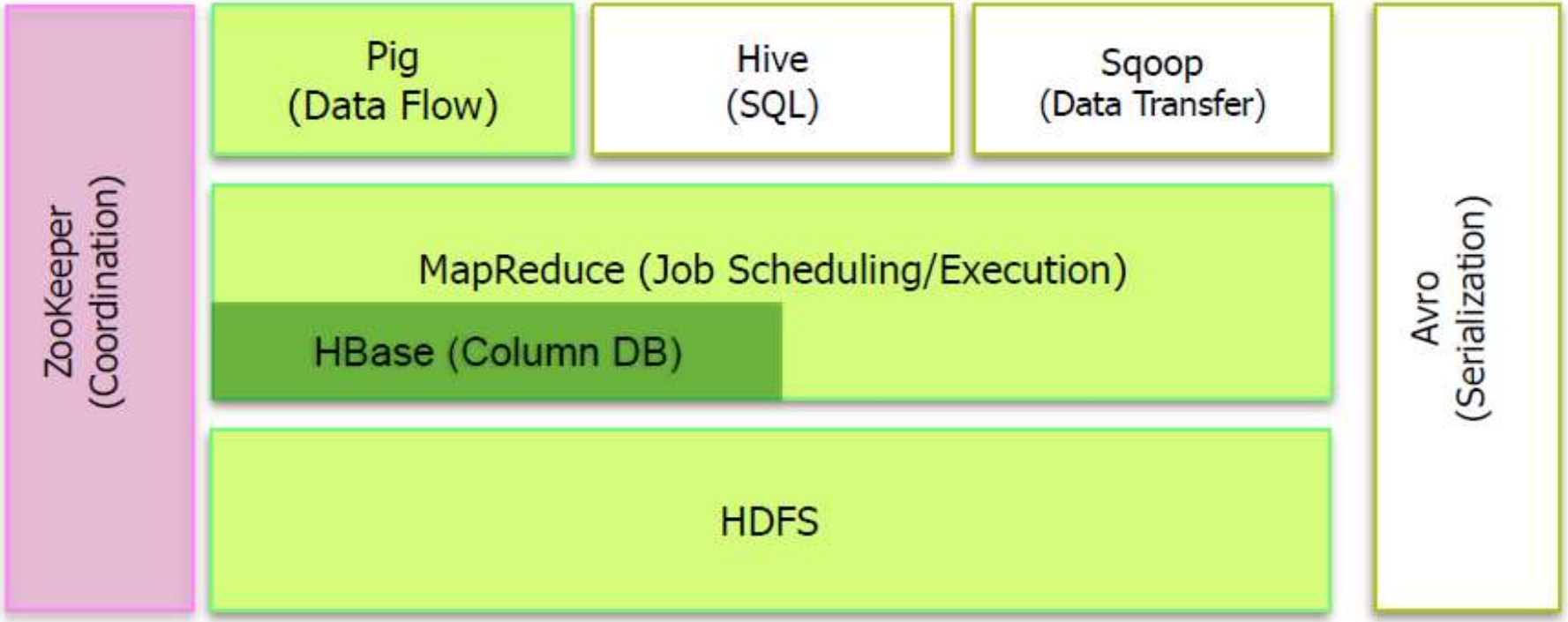
ZooKeeper

A highly-available service for coordinating processes of distributed applications.

- Developed at Yahoo! Research
- Started as sub-project of Hadoop, now a top-level Apache project
- Development is driven by application needs

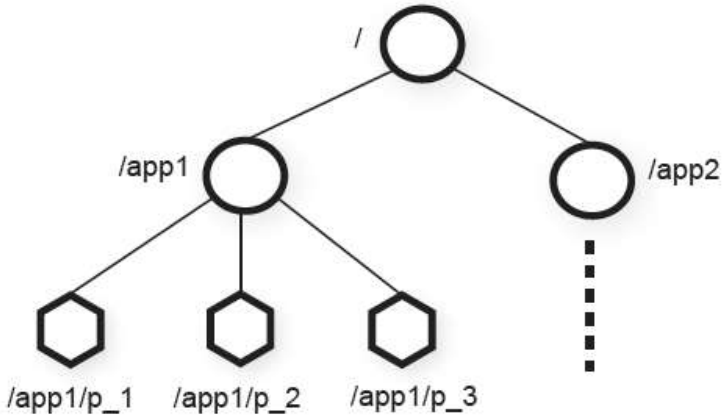
- [book] *ZooKeeper* by Junqueira & Reed, 2013

ZooKeeper in the Hadoop Ecosystem



ZooKeeper Service (1)

- Znode
  - In-memory data node in the Zookeeper data
  - Have a hierarchical namespace
  - UNIX like notation for path
- Types of Znode
  - Regular
  - Ephemeral
- Flags of Znode
  - Sequential flag



ZooKeeper Service (2)

- Watch Mechanism
  - Get notification
  - One time triggers
- Other properties of Znode
  - Znode doesn't not design for data storage, instead it store meta-data or configuration
  - Can store information like timestamp version
- Session
  - A connection to server from client is a session
  - Timeout mechanism

Client API

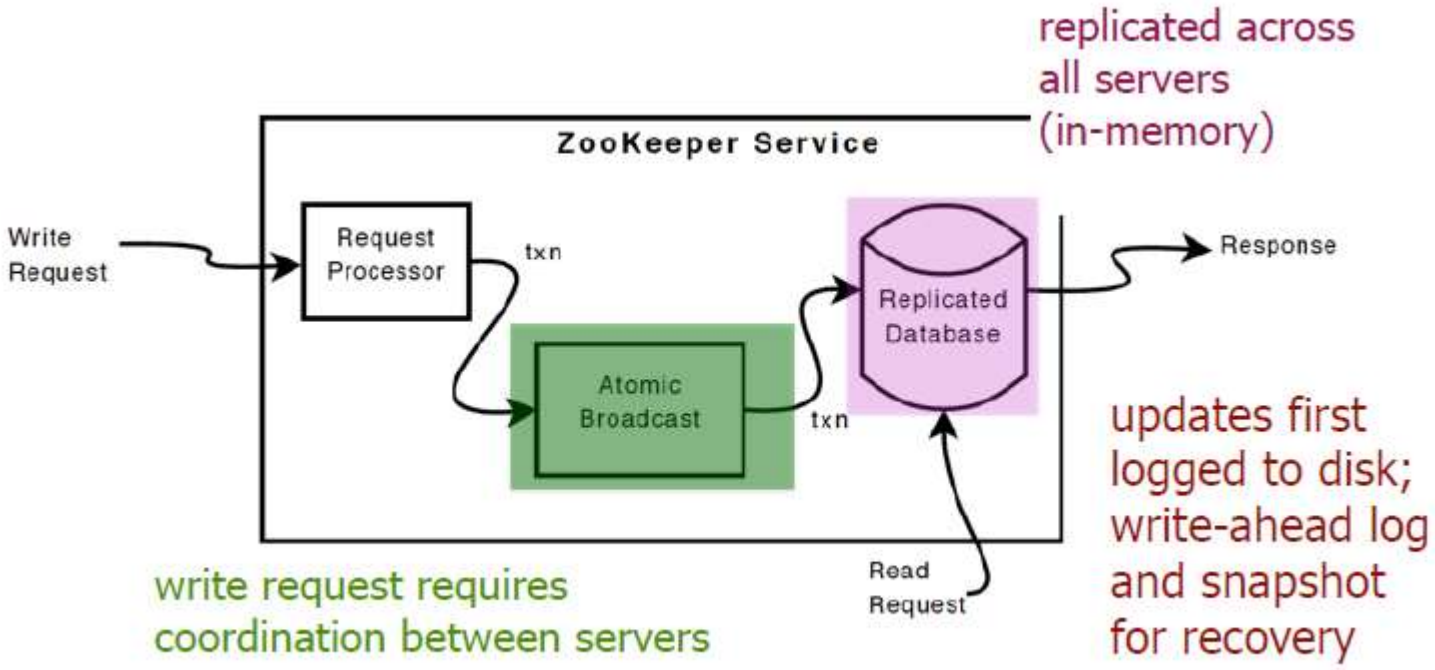
- Create(path, data, flags)
- Delete(path, version)
- Exist(path, watch)
- getData(path, watch)
- setData(path, data, version)
- getChildren(path, watch)
- Sync(path)
- Two version synchronous and asynchronous

Guarantees

- Linearizable writes
  - All requests that update the state of ZooKeeper are serializable and respect precedence
- FIFO client order
  - All requests are in order that they were sent by client.

Implementation (1)

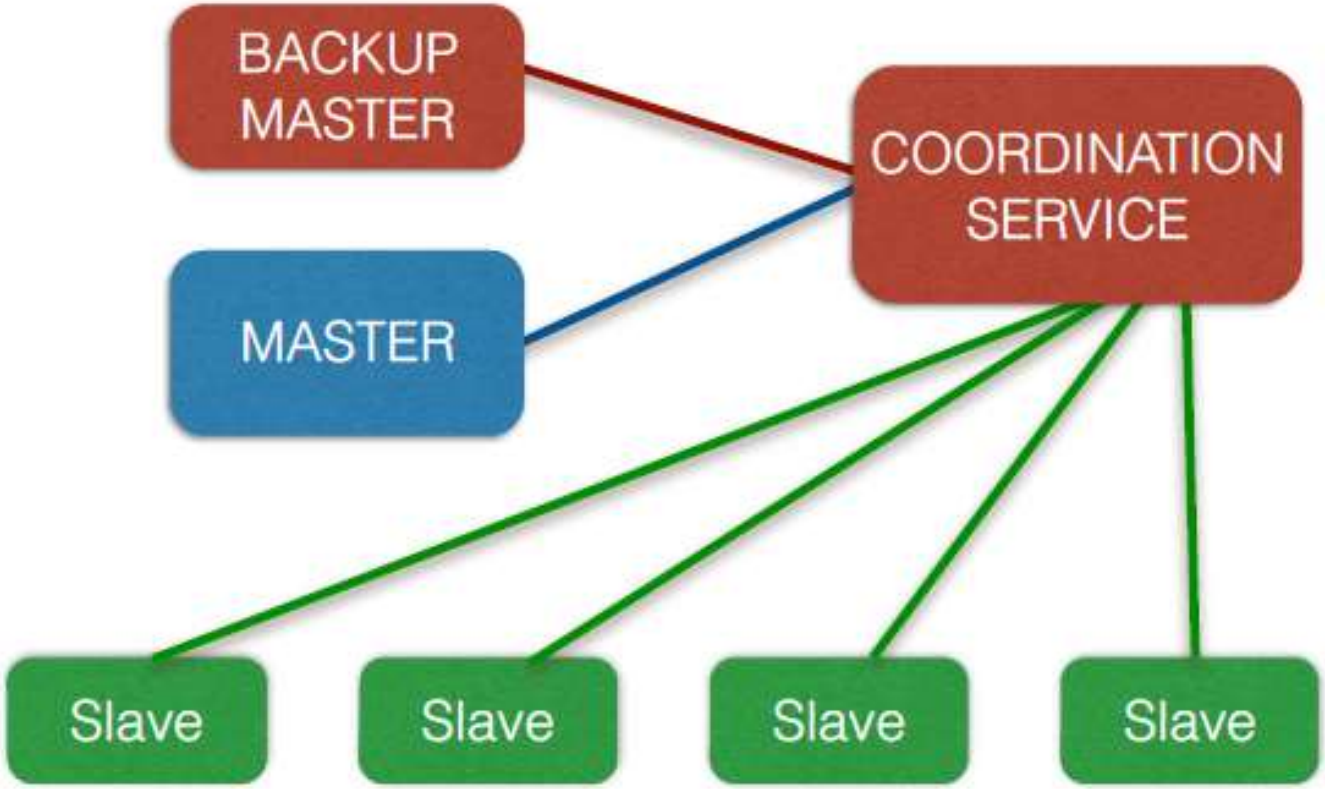
- ZooKeeper data is replicated on each server that composes the service



Implementation (2)

- ZooKeeper server services clients
- Clients connect to exactly one server to submit requests
  - read requests served from the local replica
  - write requests are processed by an agreement protocol (an elected server leader initiates processing of the write request)

Hadoop Environment



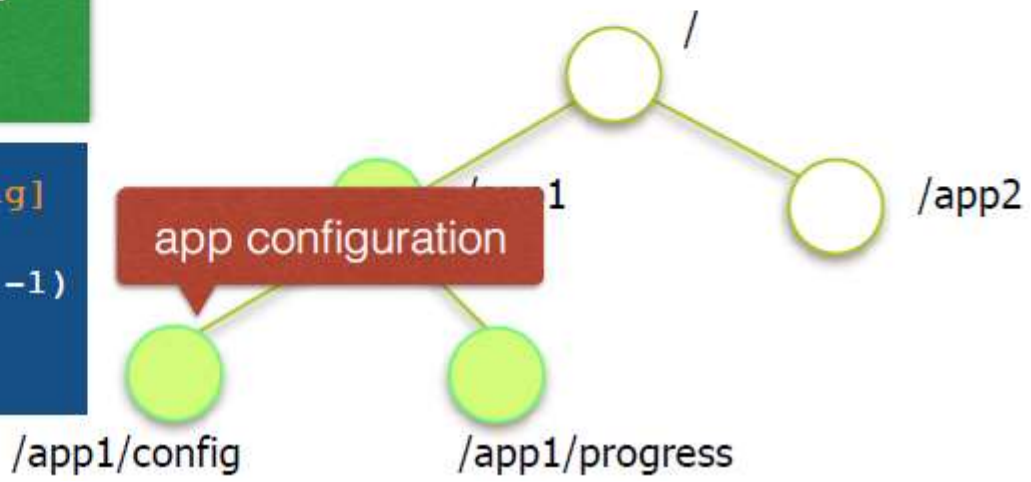
Example: Configuration

- `String create(path, data, flags)`
- `void delete(path, version)`
- `Stat exists(path, watch)`
- `(data, Stat) getData(path, watch)`
- `Stat setData(path, data, version)`
- `String[] getChildren(path, watch)`

Questions:

1. How does a **new** worker query ZK for a configuration?
2. How does an administrator **change** the configuration **on the fly**?
3. How do the workers read the **new** configuration?

```
[configuration stored in /app1/config]
1. getData(/app1/config,true)
2. setData(/app1/config/config_data,-1)
[notify watching clients]
3. getData(/app1/config,true)
```

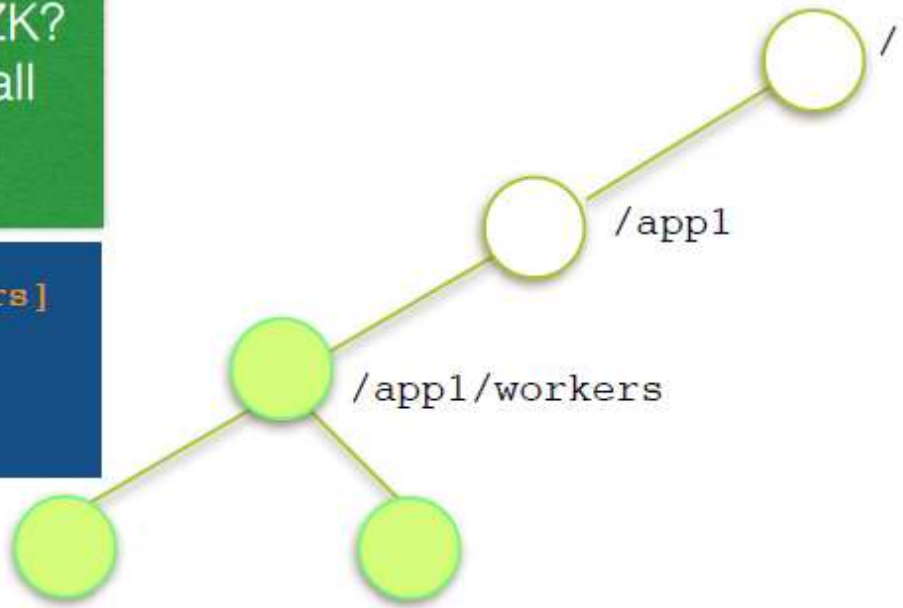


Example: group membership

- String create(path, data, flags)
- void delete(path, version)
- Stat exists(path, watch)
- (data, Stat) getData(path, watch)
- Stat setData(path, data, version)
- String[] getChildren(path, watch)

Questions:
1. How can all workers (slaves) of an application **register themselves** on ZK?
2. How can a process find out about all **active** workers of an application?

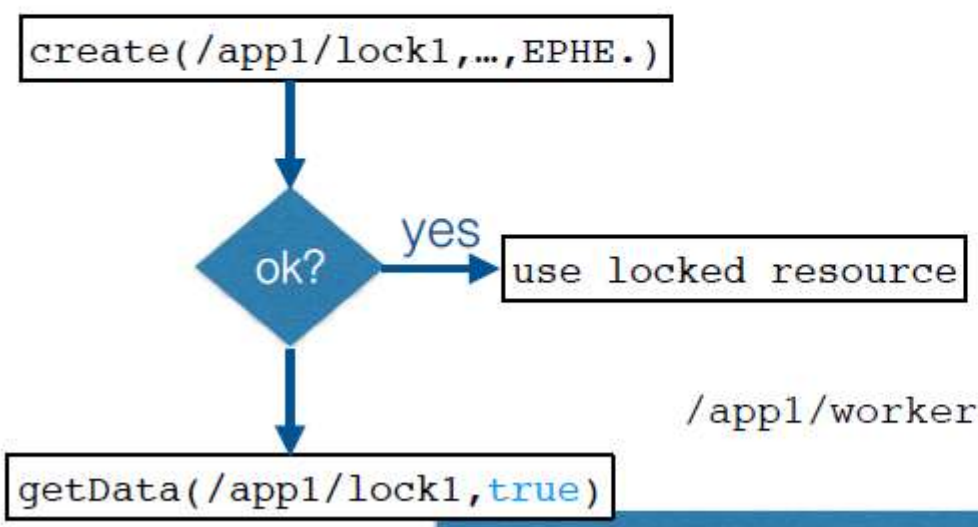
```
[a znode is designated to store workers]
1. create(/app1/workers/
worker,data,EPHEMERAL)
2. getChildren(/app1/workers,true)
```

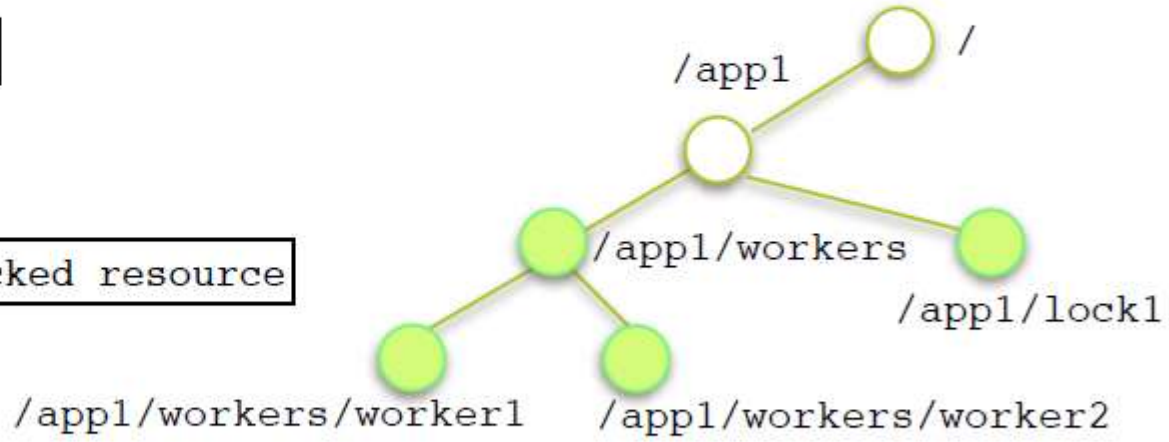


Example: simple locks

- `String create(path, data, flags)`
- `void delete(path, version)`
- `Stat exists(path, watch)`
- `(data, Stat) getData(path, watch)`
- `Stat setData(path, data, version)`
- `String[] getChildren(path, watch)`

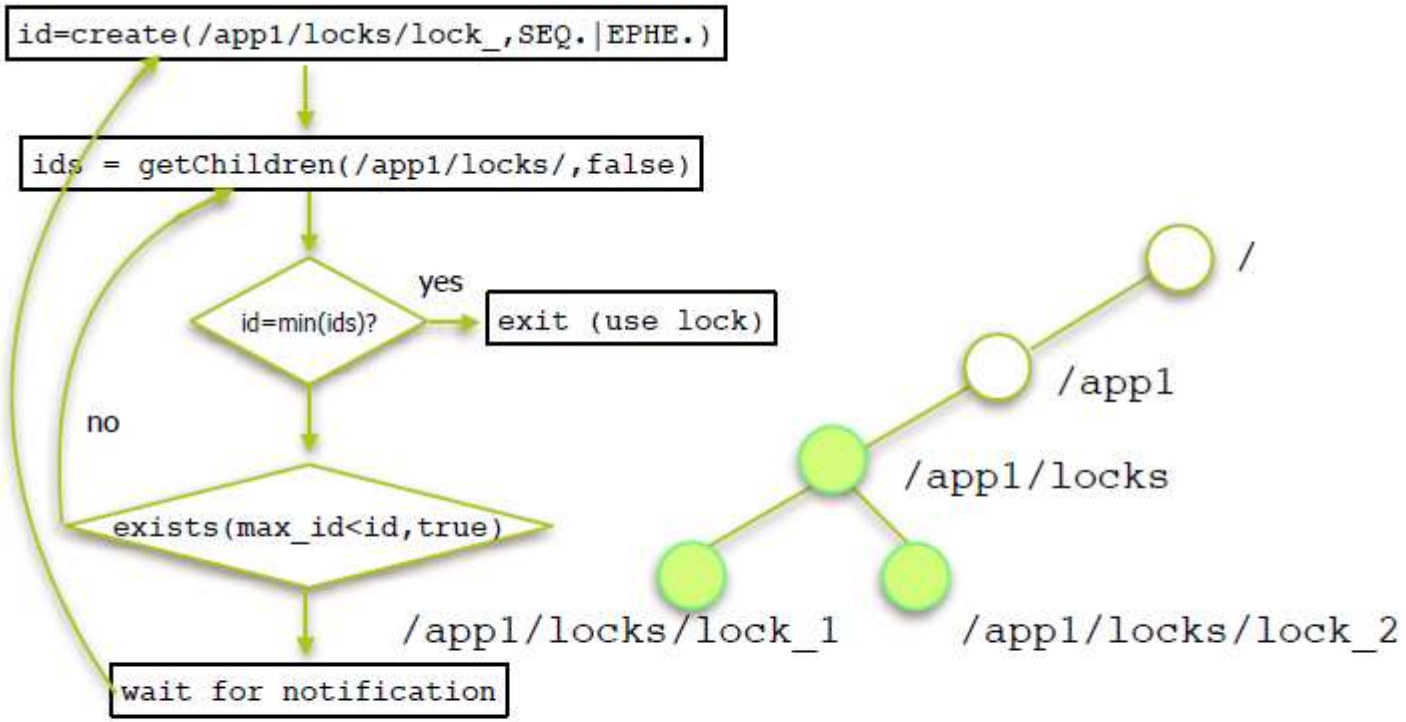
Question:
1. How can all workers of an application use a single resource through a lock?





all processes compete at all times for the lock

Example: locking without herd effect



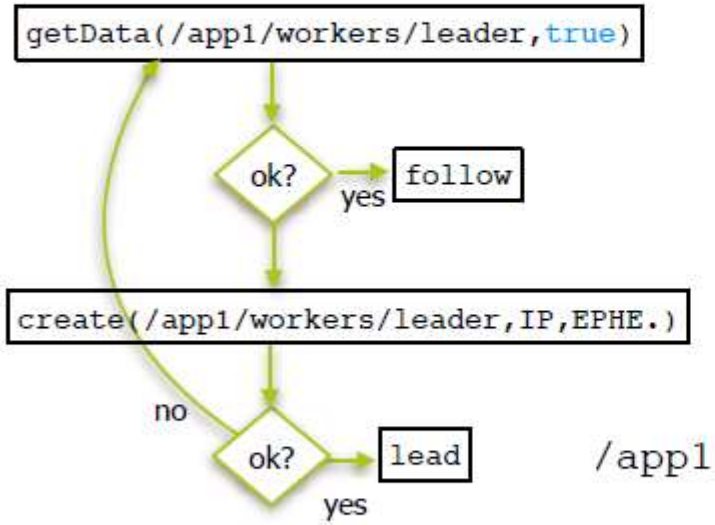
Question:

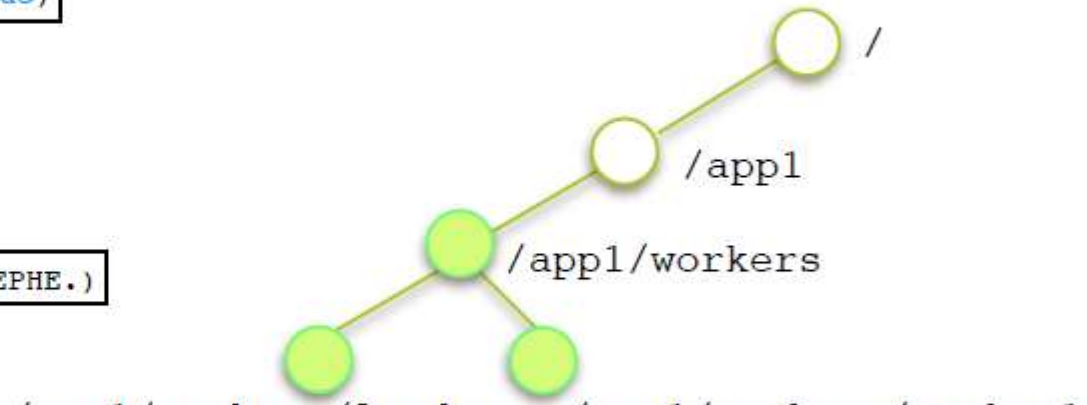
1. How can all workers of an application use a single resource through a lock?

Example: leader election

- `String create(path, data, flags)`
- `void delete(path, version)`
- `Stat exists(path, watch)`
- `(data, Stat) getData(path, watch)`
- `Stat setData(path, data, version)`
- `String[] getChildren(path, watch)`

Question:
1. How can all workers of an application elect a leader among themselves?

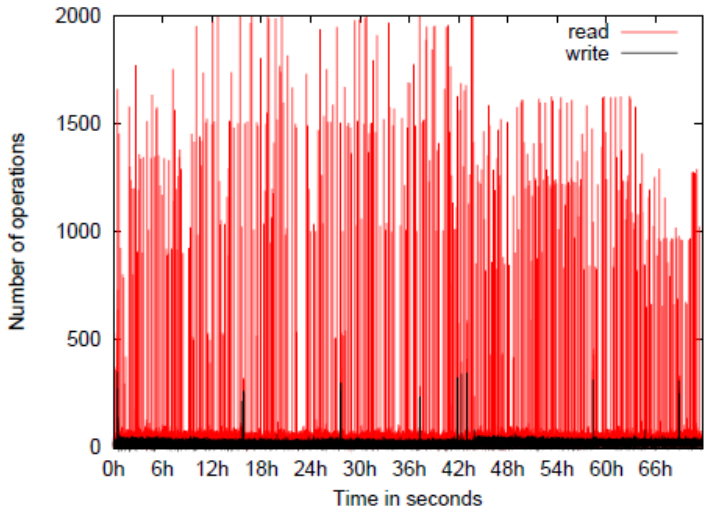




if the leader dies, elect again ("herd effect")

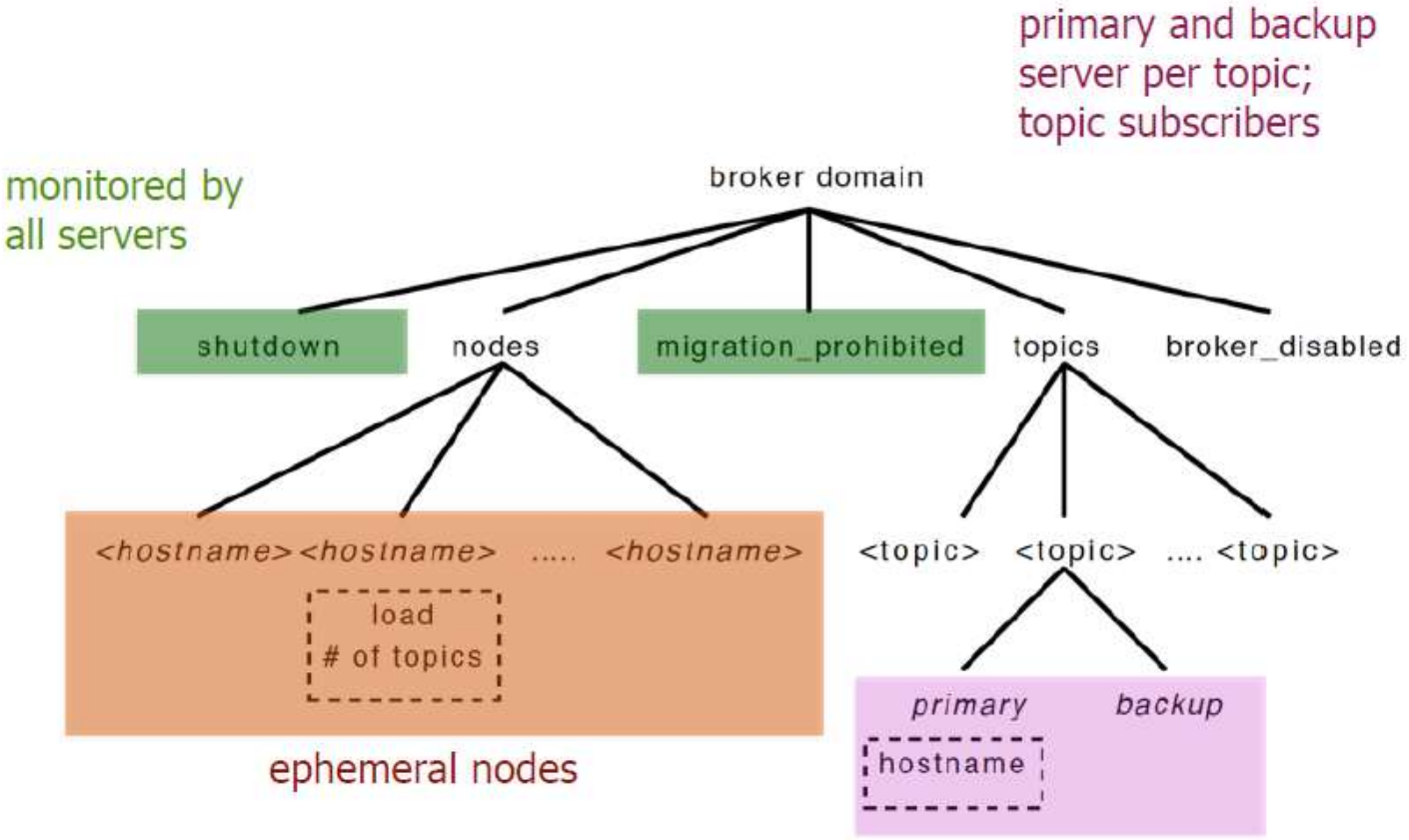
Zookeeper Application (1)

- Fetching Service
  - Using ZooKeeper for recovering from failure of masters
  - Configuration metadata and leader election



Zookeeper Application (2)

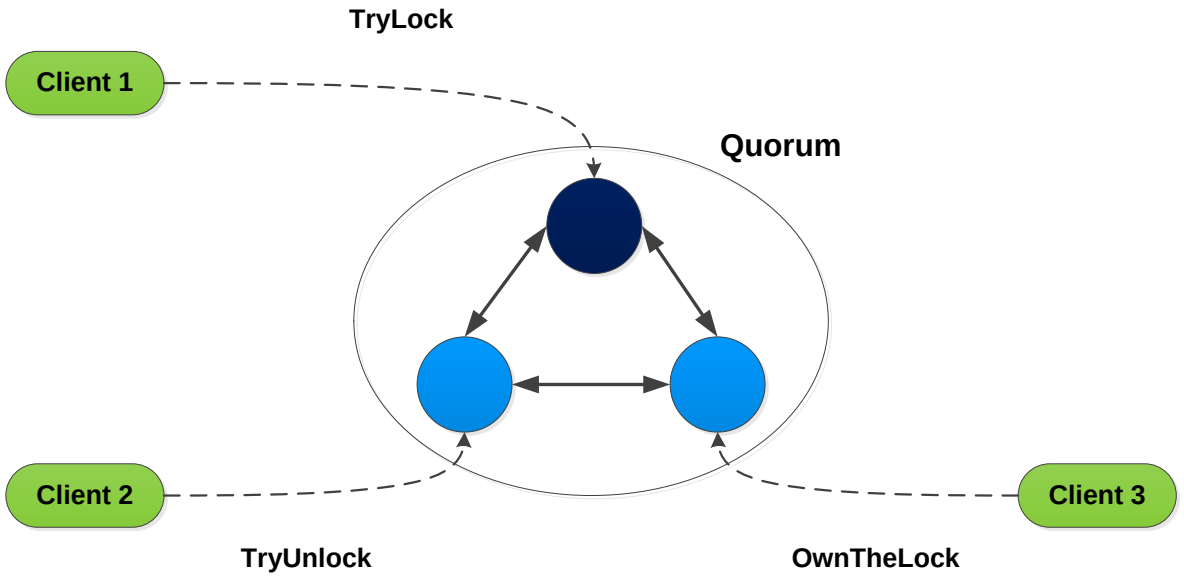
- Yahoo! Message Broker
  - A distributed publish-subscribe system



5 Project 2

Distributed Lock Design (1)

- Design a simple consensus system, which satisfy the following requirements,
  - Contain one leader server and multiple follower server
  - Each follower server has a replicated map, the map is consisted with the leader server. The key of map is the name of distributed lock, and the value is the Client ID who owns the distributed lock.



Distributed Lock Design (2)

- Support multiple clients to preempt/release a distributed lock, and check the owner of a distributed lock.
  - For preempting a distributed lock
    -- If the lock doesn't exist, preempt success;
    -- Otherwise, preempt fail;
  - For releasing a distributed lock
    -- If the client owns the lock, release success;
    -- Otherwise, release fail;
  - For checking a distributed lock
    -- Any client can check the owner of a distributed lock

Distributed Lock Design (3)

- To ensure the data consistency of the system, the follower servers send all preempt/release requests to the leader server.
- To check the owner of a distributed lock, the follower server accesses its map directly and sends the results to the clients.
- When the leader server handling preempt/release requests:
  - If needed, modify its map and sends a request propose to all follower servers
  - When a follower server receives a request propose

    -- modify its local map

    -- check the request is pending or not

    -- if the request is pending, send an answer to the client

Distributed Lock Design (4)

- In this system, all clients provide preempt/release/check distributed lock interface.
- When a client is initialized
  - Define the IP address of the target server
  - Generate the Client ID information based on the user information (UUID)

Distributed Lock Design (5)

- Reference
- Data structure of a client in the consensus system

```
class DistributedLock
{
public:
    DistributedLock(std::string serverAddr); /*Generate ClientId and establish a connection to a Server*/
    ~ DistributedLock();
    bool TryLock(std::string lockKey);
    bool TryUnlock(std::string lockKey);
    bool OwnTheLock(std::string lockKey);
private:
    std::string GetClientId(); /*Generate ClientId based on UUID*/
    bool ConnectToServer(std::string serverAddr); /*Attempt  to connect to a Server*/
    std::string clientId;
    bool isConnected;
    int fd; // the descriptor used to talk to the consensus system
}
```

Thank you!