Table of Contents

Zookeeper

What problem is this paper trying to solve?

Coordination in large-scale distributed systems
Examples: configuration, group membership, leader election, locks

Why not build coordination directly into each distributed application?

Need at least three servers for fault tolerance
  Otherwise, network partition could lead to two masters/inconsistency
  But maybe only want two replicas of a server
  So use 3+ zookeeper nodes to coordinate master/backup replica
Distributed coordination is hard to get right
  More chance to get it right *once* than in every app
  But couldn't you just use a high-quality library?
    Still administrative overhead leaving room for operator error
    Must ensure servers know about each other and how to talk to each other
    Dynamic servers can use zookeeper to find each other
Google does it (published chubby paper so everyone wanted to do that)

What are the design goals of Zookeeper

General "coordination kernel" API that supports a lot of use cases
Good performance
Fault tolerance / high availability

Let's go over the API

Main abstraction: znodes--what's this?
  Hierarchically named like file system
  But each znode can have both data and children (like file+dir in one)
  What state does zookeper keep for each znode?
    - Type: regular or ephemeral (which disappears when client does)
    - Metadata: timestamp, version
    - Data: up to 1MB, configurable
    - Children: other znodes
    - Counter: used when creating sequential child znodes
What operations can we do on znode?
  - create(path, data, flags)
    flags specifies regular/ephemeral and optionally sequential
      sequential appends counter++ to znode name
    fails if file exists (at least in ephemeral mode--p. 6)
  - delete(path, version)
    fails if version != -1 and doesn't match znode
  - exists(path, watch)
    Does path exist? If no and watch, then will get notified when created
    Note watch is a one-time notification
  - getData(path, watch)
    Get znode data, if watch then will get notified when znode changes

- setData(path, data, version)
- getChildren(path, watch)
- sync(path) - path is ignored

What do they mean by asynchronous API?

Can issue requests and get called back when they complete
A common alternative to threads for issuing concurrent requests

What's a zookeeper session?

Connection between a client and servers (associated with watches, etc.)
State replicated, so if server fails, client can connect to another one

What are the consistency guarantees of the API?

1. Linearizable writes
2. FIFO client order

What is linearizability? A consistency model (See reference [14])

Observed effects equivalent to all operations happening in some sequential order in which non-overlapping operations are temporally ordered

```
E.g.,  |---A---|       |---C---|
                  |---B---|
       A must happen before B
       B and C could happen in either order, but everyone sees same order
```

Linearizability is a nice property for distributed systems
  Objects can implement locally (provided no cross-object transactions)
  But makes it easy to reason about systems
  Example: write value, pick up phone and call friend, friend reads value
    Linearizability guarantees friend will see your write
What is A-linearizability? That's where #2 (FIFO client order) comes in
Lets single client issue overlapping operations & respects FIFO order
Otherwise, concurrency requires "virtual clients" w. no ordering

Wait, how can only the writes be linearizable?

Means example of write, call friend, friend reads won't work?
Why do they do this?
  Read performance--any server can answer read requests
  Means total reads/second increases with growing number of servers
What would zookeeper need to do to provide linearizable reads?
  Could handle reads just like writes--replicate everywhere
    Would send reads through atomic broadcast protocol
  Could send all reads to an elected "primary" zookeeper server
    Would require primary to have *lease* so it knows it is still primary
What's the workaround? Issue sync [c.f. reading question]

What's an atomic broadcast protocol?

Consensus applied to a series of messages
  (All nodes agree m1 is first message, m2 second message, etc.)

Why does zookeeper need an atomic broadcast protocol?
All servers must converge on identical state, even after failures
Common technique is replicated state machine (RSM):
1. All servers agree on system's initial state (hard-coded)
2. All servers agree on each deterministic update before applying it

What are leases?

A time-limited promise by system to notify you before changing some state
E.g., primary lease means primary knows it is still primary for 30 seconds
Paper knocks Chubby for using client leases--why are these bad?
Server promises "For 30 sec, I'll tell you before changing value x"
Then client fails, so server can't notify it
Delays updating x until lease expires
Why doesn't zookeeper need client leases?
Zookeeper delivers watch notifications *after* changes happen
But note session timeout mechanism can effectively be used for leases
Need ephemeral lock node to disapper after client failure?
Waiting for client session to time out

Example: Ready znode

Master deletes ready znode, changes config, re-creates ready znode
Clients call getData(ready, true) to get notified when deleted
What happens if client sees ready before new master deletes it?
E.g., client: getData(ready, true); getData(config1); getData(config2)
Don't want to see inconsistent config1, config2 as new master writing
FIFO ordering guarantees delivery of watch from ready before config data
So client knows to ignore the results of config1, config2
Just calls exists(ready, true) and restarts read when ready exists

Example: Simple lock

Create ephemeral lock vnode--if succeeds, you have the lock
If fails, call getData(lock, true), notified when deleted
If client session fails, lock automatically released
What's wrong? Herd effect
All clients woken up when lock released, but only one can get lock

How to do lock w/o herd effect?

Define lock holder has creator of znode with lowest sequence number
To acquire create ephemeral and sequential child of lock znode
Call getChildren()
Now you know if you have the lock
Otherwise, know who is before you in line to get the lock, call it p
Call exists(p, true) [in case deleted in meantime, and to watch for it]
When p no longer exists, repeat previous step--why?
p is ephemeral, might disappear from session timeout, not getting lock
To release lock, delete your child znode

How to do Read/write lock?

Same as previous, but readers wait for previous writer, not any locker
Encode read/write intent in znode name ("lock/read-SEQ" or "lock/write-SEQ")

Double barrier?

Use ready node trick, and create when you are nth to join

Are zookeeper's API functions idempotent?

Definitely not. E.g., sequential znode create bumps counter
Why does section 4.1 say "transactions are idempotent"?
  API calls get translated into transactions
  Transactions sent through atomic broadcast, replicated on all servers
What's the issue with calculating "future state"?
  Zookeeper is pipelines operations
  So may have multiple transactions where atomic broadcast not complete
    Can't apply state if transactions not committed
    But to make idempotent, need to know result (e.g., new counter value)
    So calculate state based on previous pending transactions, too
What is the advantage of idempotent transactions?
  Allows write-ahead logging
  Also, makes fuzzy snapshot mechanism work

What evaluation questions should we ask?

Performance (reads/writes per second)
Correctness/fault tolerance (not really evaluated)

Table 1 (p. 10)--why more reads fewer writes with more servers?

Reads can be handled by any server, so more servers = more reads/sec
Writes go through atomic broadcast,
  requires work from all servers, plus communication overhead + tail latency

A New Presumed Commit Optimization for Two Phase Commit

- Say you want to move $100 from account A to account B at a bank
- Such a transaction could fail depending on state of ledger
- E.g., Account A has insufficient funds, or either account deleted
- Want serializability - as if all transactions executed in some order
- Concurrent transactions on A and B must not interfere
    - Read B, write B+100 must be equivalent to atomic B += 100
- Everyone must agree on exactly what transactions preceded this transfer
    - Otherwise, say A only has $100 but attempts two payments to B and C
    - If neither transaction sees the other, both will succeed
    - Or say A and B each move $1M (they don't have) to the other
    - Transfers could succeed if each transaction thinks other happened first
- Want recoverability - transaction is atomic (all or nothing)
- Debit A iff credit B, even with power failures, crashes, etc.

Say bank ledger is in an RDBMS--how do we achieve these properties?

- Serializability - take out locks on both accounts before moving money
- Recoverability - generally use a write-ahead log
    - Write atomic idempotent description to log before changing B-trees
    - Post-crash, log replay has same effect regardless of B-tree state
    - Alternatives: keep undo log (SQLite), DO-UNDO-REDO logging, etc.
- Note: locks can fail or be revoked, in which case RDBMS aborts transaction
    - Aborted transaction has no effect on database contents
    - Must be reported to client, which tries again or gives up

Say ledger is large, must be sharded, with A and B in separate RDBMSes, Naive approach, two transactions on two RDBMSes--what goes wrong?

- Serializability - say concurrent payments T1:A->B($100) and T2:C->A($100)
    - A in DB1, B+C in DB2: DBs could see to payments in different orders
    - T2: lock C
    - T1: lock B, lock A, commit, release locks
    - T2: lock A, commit, release locks
    - Even if linearizable, T2 can order T2 before overlapping T1
- Recoverability - What if T1 commits on DB2 but aborts on DB1?

Solution? Use 2PC to ensure *all* DBs either commit or abort

- Acquire a bunch of locks, finish your transaction, decide to commit
- Phase 1: Preparing
    - Coordinator broadcasts PREPARE to all cohorts (DBs in transaction)
    - Up to this point, each cohort can abort (e.g., if it revoked lock)
    - If a cohort aborted, responds with ABORT-VOTE
    - Otherwise, respond COMMIT-VOTE: now can't abort or touch locks!
- Phase 2: Committing/aborting
    - If any cohort voted ABORT-VOTE, coordinator broadcasts ABORT message
    - Otherwise, if all voted COMMIT-VOTE, then coordinator broadcasts COMMIT

    - If COMMIT, cohorts *must* commit transaction
    - Only now can cohorts release any locks associated with transaction
    - Cohorts reply with ACK so coordinator knows they received outcome
- Does this solve recoverability? Yes if everything logged
- Does it solve serializability? Possibly
    - DB2 must commit T1 before learning whether or not T2 committed
    - But T2 could still have a lower timestamp than T1 on DB2
    - Okay if you assign timestamp in Phase2 (e.g., on receipt of PREPARE)
    - At that point, all associated locks held on all cohorts

When must a cohort force-write something to disk?

Before sending COMMIT-VOTE? Always

- COMMIT-VOTE is a promise not to abort transaction or release locks
- If cohort reboots with no record of promise, can't possibly keep it
- E.g., before hearing from coordinator of transaction it forgot about,
    - might agree to commit conflicting transaction with other coordinator

Before sending ACK? Depends on variant

- Naively yes, because it should permanently commit/abort transaction
- But if it crashes, will know transaction existed from COMMIT-VOTE record
- If coordinator remembers what happened, cohort can just ask it
- Whether an ACK message is even required (and associated write) depends on
    - What information coordinator retains
    - Whether transaction committed or aborted

How much to we care about these forced writes?

- COMMIT-VOTE is on the critical path for transaction latency
- ACK is not--coordinator already knows transaction committed
    - So disk latency won't affect transaction latency
    - Maybe delay, piggyback on another log write for better throughput
    - Possibly unfair of paper to lump these together in single "n" value

When must coordinator force-write in paper's PrN variant?

- Before sending PREPARE? No (only if pedantically presuming nothing)
    - Until it sends COMMIT, coordinator can unilaterally ABORT a transaction
    - Cohort inquires about unknown transaction? coordinator "presumes" abort
- Before sending COMMIT? Yes
    - At this point transaction happened, so need to record it durably
    - Otherwise, couldn't properly respond to cohort inquiries after crash
- Before sending ABORT? Yes
    - This is the part about presume nothing
- Upon receiving ACK? No (non-forced write)

OP table

OP	PrN	PrA	PrC	NPrC
log before PREPARE	N	N	Y	N
log before COMMIT	Y	Y	Y	Y
log before ABORT	N	N	N	N
ACK after COMMIT	Y	Y	N	N

ACK after ABORT	Y	N	Y	Y

(Note: cohorts log before COMMIT-VOTE in all schemes)

How does presumed abort (PrA) differ from PrN?

- Don't write to disk before sending ABORT
  - Coordinator never writes aborted transactions--no garbage to collect
  - No matter when coordinator crashes, nothing on disk
  - So cohorts inquiring about transaction will always get same answer
  - Hence, cohorts don't even need to send ACK message to ABORT
  - And hence obviously no force-write before (non-existent) ACK message
- But COMMITs (common case) are exactly same cost as PrN

What are costs for traditional presumed commit (PrC)?

- Coordinator writes before PREPARE? Yes
  - Otherwise, after crash would presume committed if cohort inquired
  - Without all votes, coordinator can unilaterally ABORT but not COMMIT
- Coordinator writes before COMMIT? Yes
  - Have to log to "undo" effect of log record before PREPARE
  - Otherwise, would see transaction after crash and abort it
  - But cohort doesn't have to ACK COMMIT
- Coordinator writes before sending ABORT? No
  - But cohorts must ACK ABORT
  - And coordinator has one more non-forced cleanup write when all ACKs in

What's the read-only optimization?

- Transaction might be read-only at one cohort
  - Only effect is to hold locks for duration of transaction
  - So cohort doesn't care whether transaction commits or aborts
- Cohort replies to PREPARE with READ-ONLY-VOTE
  - Also releases all locks when it sends READ-ONLY-VOTE
  - any locked data unmodified since transaction read-only at cohort
  - Coordinator doesn't send COMMIT/ABORT message to read-only cohort
- If all cohorts reply READ-ONLY-VOTE, then whole transaction read-only
  - If coordinator didn't log PREPARE message, doesn't need to log anything
  - Cohorts don't care whether committed or aborted
  - With PrC, must write (non-forced) record to delete logged PREPARE message

What's the idea behind the NPrC optimization?

- Trade garbage collection after crash for messages+writes
- Window of recent transaction ids (tid_l,...,tid_h) presumed aborted
  - All other transactions presumed committed
- Must stably log the window parameters (tid_l,tid_h)
  - tid_h can be implicit based on highest stable transaction
  - or can be explicitly logged but amortized over many transactions
  - must guarantee no tids >= tid_h ever used
  - tid_l piggybacked onto other log writes
  - points to oldest "undocumented" transaction

- Now no need to log before sending PREPARE when in window (common case)
  - Can always log PREPARE of slow ("recalcitrant") transaction later
  - Logging PREPARE allows tid_l to advance despite a few stragglers
- But also no need for cohorts to ACK COMMIT messages!
  - Because tid_l suffices for coordinator to "remember" an arbitrary
  - number of committed transactions cohorts might ask about
- Only common-case forced log write is before sending COMMIT

What happens if NPrC coordinator crashes while committing tid?

- If tid >= tid_l, then disk will contain COMMIT log record
- If tid < tid_l, then will presume committed, so no need for record

What happens if NPrC committed crashes while aborting tid?

- Won't advance tid_l > tid until all cohorts ACK the abort
- So either tid > tid_l and presumed aborted, or no cohorts will inquire

What is garbage collection issue?

- Never logged PREPARE? won't know what cohorts involved in transaction
  - Can't collect ACKs for ABORT, have to be prepared for inquiries
  - No time bound on unknown cohorts inquiring about unknown transactions
- Solution is keep permanent record of presumed abort ranges after each crash
- Could alternatively keep track of all possibly active cohorts

Viewstamped Replication

Clarification on serializability (I was unclear Monday)

Two-phase locking is acquiring all locks before releasing any
Two-phase locking guarantees serializability even in distributed system
Assumes each participant orders transactions by COMMIT-VOTE time
Because at that point (in prepare phase), all locks are held by all cohorts

Almost nobody seems to understand how to implement consensus

They know you use "Paxos" for consensus, but don't know what that is
My theory: We've been teaching wrong
Notably, actual paxos paper (The Part-Time Parliament) unreadable
Also, VR (today) blurs abstraction barrier between consensus and 2PC
Another theory: These protocols are wrong because they are hard to teach
Plan for the next three lectures:
Today: The protocol many people incorrectly call Paxos
Monday: The actual Paxos protocol, and how to think about it
Wednesday: Raft--a protocol designed to be easier to understand
Guest lecture from Raft designer, representing the other opinion
(I endorse Raft, just think it should be understood in broader framework)

The big picture for VR

We have *modules*, which are collections of objects (think database)
We want serializable, recoverable transactions across different modules

Modules may be on different machines, so use 2-phase commit
For availability/reliability, replicate modules into *module groups*
  A module group is a set of cohorts storing an entire copy of the module
Replication poses challenges
  Implies the need for consensus across cohorts in module group
  Need to reconfigure group if cohorts fail or are unreachable
  Some state/history could be lost during reconfiguration
  Must recognize transaction depending on lost state and vote to ABORT
Design point also eschews stable disk writes for replication (homework)
  Questionable... can't survive ubiquitous power outage without UPSes
  But techniques introduced can be adapted to survive power failure

Let's first think about how to implement module groups

To simplify, say module group does single-RPC transactions, so no 2PC
Example: credit card payment server with RPC-driven state machine:
* CHARGE (transaction-id, CC #, merchant, amount) -> {approved, denied}
  - Merchant issues RPC when customer pays for item
  - If system says approved, customer can leave with merchandise
Obviously don't want to forget about transactions if server fails
So replicating in case of server/disk failure is a good idea
Note also that we care about order of transactions
  If CC# gets two large charges, only first executed one might succeed

First attempt: Simple protocol like two-phase commit

Designate one server the primary (~coordinator), and the others backups
Client:
  - sends CHARGE RPC to primary
Primary:
  - assigns the RPC a consecutive sequence number
  - implicitly votes to COMMIT the transaction, logs it
  - broadcasts PREPARE message containing: { sequence number, CHARGE args }
Backups:
  - if sequence number not greater than all others seen, reply ABORT-VOTE
  - otherwise, log PREPARE message to disk and send COMMIT-VOTE
Primary:
  - Transaction commits if *all* backups vote COMMIT
  - Log transaction to disk either way
  - Broadcast transaction result to backups
  - If transaction ABORTed, conservatively send back denied to client
  - If transaction COMMITted, execute and send result to client
Backups:
  - Log outcome and reply ACK to primary
  - If don't hear from primary, ask it
Primary:
  - Periodically send END record for highest seq# ACKed by all cohorts
  - Cohorts can forget about transactions before END

What happens if one of the backups fails?

System grinds to a halt
But N-1 replicas of the data still exist
So human operator can just configure system to have only N-1 replicas
  If remaining replicas all voted to COMMIT, safe to do so
Now can continue operation with no lost data

What happens if primary fails?

Customer either walked out with item or not, want charge to match
If any replica didn't vote COMMIT, customer doesn't have item, so ABORT
  E.g., replica might never even have received the PREPARE
If all replicas voted COMMIT, then customer may have gotten item
  or may still be waiting for approval in store
  So put charge through; client can re-transmit CHARGE RPC if necessary
  Note transaction Id in CHARGE RPC makes it idempotent
Once you determine transaction status, make another node the new primary

What happens if primary and another replica both fail?

If all surviving replicas voted COMMIT, we are in trouble
  Customer may have left with item, may have left w/o item, may be waiting
So 2PC was probably the wrong protocol to use

Second attempt: Use *three*-phase commit

Problem was "prepared" state where replicas can go either way
So now after PREPARE, if everyone votes COMMIT, broadcast PREPARE-COMMIT
Once all participants "PRE-ACK" PREPARE-COMMIT, broadcast COMMIT
After failure, COMMIT if and only if any survivor saw PREPARE-COMMIT
Why is this better?
  2PC: execute transaction once everyone willing to COMMIT it
  3PC: execute transaction once everyone *knows* everyone willing to COMMIT
  Customer only gets item when everyone knows charge safe to COMMIT
But still two problems:
  3PC is expensive to do for each operation
  Need human operator to configure primary and #replicas after each failure

Can we fix 3PC to recover from failure automatically? No.

3PC is guaranteed to reach consensus if no machine fails
Remember FLP? Can't guarantee termination if protocol is fault-tolerant
  But in an asynchronous system, failed machine looks like slow machine
  ... and consequently slow machine looks like failed machine
  E.g., might time out and initiate recovery for machine that didn't fail
So counter-intuitively, to achieve fault-tolerance we must use...
  a protocol that's *not* guaranteed to terminate even without failure!
  But we can make it terminate in practice even with failure

Some basic VR module group concepts:

GroupId	Unique identifier of group

Mid	Module Id uniquely identifies cohort
Configuration	The total set of cohorts (failed and not) in group
View	Functioning subset of configuration with designated primary cohort
ViewId <counter, mid>	Pair uniquely identifying a particular view. Totally ordered with counter most significant field. mit is Id of cohort that chose ViewId, to guarantee uniqueness
ViewStamp<ViewId, TimeStamp>	Pair uniquely identifying an event. Totally ordered with ViewId most significant. TimeStamp is int consecutively assigned to each event by view's primary
Pset	Set of <GroupId, ViewStamp> associated with a particular transaction

Normal case execution of a single atomic operation by module group

Client sends request to primary
  Request includes client chosen unique call-id to make it idempotent
    Like transaction-id in CHARGE RPC
  Module state includes client table so can reply to duplicate requests
Primary assigns next ViewStamp to call event, sends it to backups
Majority of cohorts (sub-majority of backups) acknowledge operation
  Acknowledgments are cumulative, meaning have <v,ts> if ack <v,ts+1>
Primary replies to client
  Once client hears, majority of cohorts know of transaction

View change protocol

Cohort suspects failure/recovery, demands view change
  Becomes view manager, other cohorts underlings for that view
View manager picks new ViewId using cur_viewid.counter+1 and its own mid
Broadcast invitation to all other cohorts in group
Form new view if majority accept and any of the following 3 holds (p.14):
  1. Majority of cohorts accept new view and didn't crash
  2. All crashed nodes have lower viewid than the highest seen
  3. Some crashed nodes have same viewid, but primary from that view
     didn't crash and accepted the new view
  Why #1?
    If client got ack for call event, majority of cohorts saw it
    So at least one cohort in new view will know all acked events
  Why #2?
    You know a majority of nodes knew all forced operations in last view
    Problem is some of them might have forgotten because of a crash
    #2 tells you crashed nodes didn't forget, because they never new
  Why #3? Because primary will know of all transactions in view
Chose primary for new view as:
  1. Primary of highest known previous successful view (first choice), or
  2. Cohort with knowledge of operation with highest known ViewStamp
Some nodes may need to download missing operations before joining new view
After view formed, force VieStamp <ViewId, 0> so everyone knows formed
  All cohorts force current view_id to disk
If can't form view or get invitation for higher numbered View, give up
Why might you be unable to form new view?
  Too many cohorts already trying to form view with higher ViewId
  Or had majority but some failed and others onto higher ViewId

Now how do we combine this with 2PC?

Operations assigned to ViewStamps will be events
Some events are "completed-call for aid, locked these objects"
Other events are "committed" or "aborted" transaction aid
Keep psets associated with each transaction
Tells primaries what to force to backups on prepare
If events lost after view change, pset also lets primary ABORT-VOTE

Normal-case transaction processing by a module group

Client looks up (primary, ViewId) of target module group
Make one or more RPCs to primary, including ViewId
Primary rejects if ViewId incorrect (could be old message bouncing around)
Otherwise executes call and assigns it a ViewStamp
Adds <ViewStamp, obj-list, aid> to tentative history
Returns pset including ViewStamp (+ nested calls) with RPC result
Client decides to commit, acts as 2PC coordinator
Includes merged pset with PREPARE message
Primary behavior on receipt of PREPARE
- ABORT-VOTE if any unknown ViewStamps for its group in pset
Determined by *compatible* predicate (p. 11), as follows:
history = highest view stamp seen for each view
For each viewstamp in pset that applies to this group:
Make sure it is bounded by history--otherwise lost in view change
But is this correct? (see *** below)
- Otherwise force all events of group in pset to backups before COMMIT-VOTE
But note these might already be at backups, so often no work to do!
Why might ViewStamp in pset be missing from history?
completed-call events are not flushed to backups before replying to client
so could be lost in a view change, requiring transaction abort
On receipt of COMMIT, primary forces "commit" event to backups before ACK
On receipt of ABORT, primary creates "abort" event to record
After COMMIT or ABORT, can release locks

Notes:

Suppose node A is the primary in view v1, and assigns viewstamp <v1,10> to a completed-call event for an RPC to which it has replied. Now say there is a transient network partition that causes a view change at a point where only nodes A and B have heard about <v1,10>. So suppose the following series of views form, where * designates the primary in each view:

v1: A*, B, C, D, E
v2: C*, D, E
v3: C*, B, D, E

v4: B*, D, E

Now say the coordinator for the transaction that depends on <v1,10> finally decides to commit and sends a PREPARE to B while the cohorts are active in v4. The definition of compatible on p. 11 implies that a pset containing <v1,10> is compatible with B's history, even though it's not compatible with the histories of D and E who have not heard of <v1,10>. Of course, B is supposed to flush all events with ViewStamps in the pset to the backups. But B was not the primary for <v1,10>, and did not participate in the formation of view v2, so it doesn't really have a way to know that D and E don't know about <v1,10>. <v1,10> wasn't ever in B's buffer, so it won't get sent to C and D by forcing B's buffer.

This isn't fatal. The simplest solution might be to ABORT-VOTE any transaction whose pset contains ViewStamps from any earlier views. Obviously the authors didn't want to do that. A slightly milder version might be for the primary to abort if the pset contains ViewStamps from when it was not the primary (so that if it was primary for several views in a row, it can avoid ABORT-VOTEs).

Another solution would be for cohorts to delay updating history until they know that a majority of cohorts has heard of an event (but still report the latest seen ViewStamp during view changes). The primary could piggyback history updates on other messages when it has acknowledgments from a sub-majority of backups. Such piggybacking of sufficiently replicated event Ids is actually a classic technique for consensus algorithms. When using consensus to pick a sequence of inputs to a deterministic replicated state machine, cohorts commit (apply) operations once they know a majority of cohorts has them and hence that the operations won't be rolled back. Unfortunately, ViewStamped replication commingles this low-level "committing" at the consensus layer with higher-layer 2PC "committing" of distributed transactions, and as a result the protocol doesn't seem to track stable events.

Yet another solution might be that when the primary forces the buffer for a PREPARE, it asks backups to check their histories if the transaction's pset has older ViewStamps of views where the backups' histories could be behind the primary's. There is no mention of such a check in the paper. Indeed, the primary is supposed to reply immediately to a PREPARE if there's nothing in the primary's buffer.

Paxos

State machine replication review

Boils down to a consensus problem for each slot in an operation log

FLP review: In a deterministic, asynchronous consensus system

Pick at most two of: safety, liveness, fault-tolerance

How might you achieve safety and liveness?

Two-phase commit--safe, and guaranteed to terminate in two phases

Or simpler: one-phase commit (take whatever lowest-numbered node says)

How about liveness and fault-tolerance?

E.g., Every node broadcasts its input value

wait 5 seconds then chose lexicographically lowest value seen

Guaranteed to terminate in 5 seconds (if any node non-failed)

But network partition could prevent agreement

What about safety and fault-tolerance?

In theory, this is what viewstamped-replication gives us

But VR gives us more... how about something simpler?

Straw-man non-live consensus protocol:

Everybody broadcast value, wait 5 seconds

Every node votes for lowest value seen (and can never change vote)

If some value receives a majority, output it

Otherwise, you are "stuck" and there is no liveness

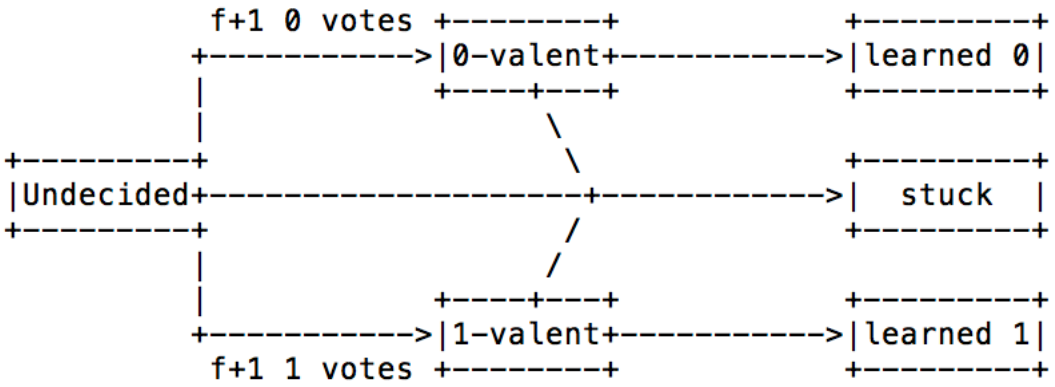
E.g., 2f+1 nodes total, f voted for 0, f voted for 1, and 1 node failed

Or no failures but three-way split between values 0, 1, and 2

Neither VR nor straw-man have liveness. Why is VR better?

Straw-man gets stuck with split votes

VR never gets stuck; even if it hasn't terminated, there's always hope



Some important conclusions from discussion thus far:

- Voting is a key technique providing both safety and fault-tolerance
- But votes can split and produce permanently indeterminate outcomes
- Or failed cohorts + lost votes mean never learn system 0/1-valent
- So better NOT vote on questions you can't afford to leave indeterminate (e.g., what is the 5th operation in the state machine log)

Key idea in many of these protocols:

Carefully craft statements on which nodes vote

Either avoid possibility of split vote

Or ensure indeterminate outcome won't permanently block log consensus
Again, can't directly vote on log entries (e.g., "log index 5 is x")

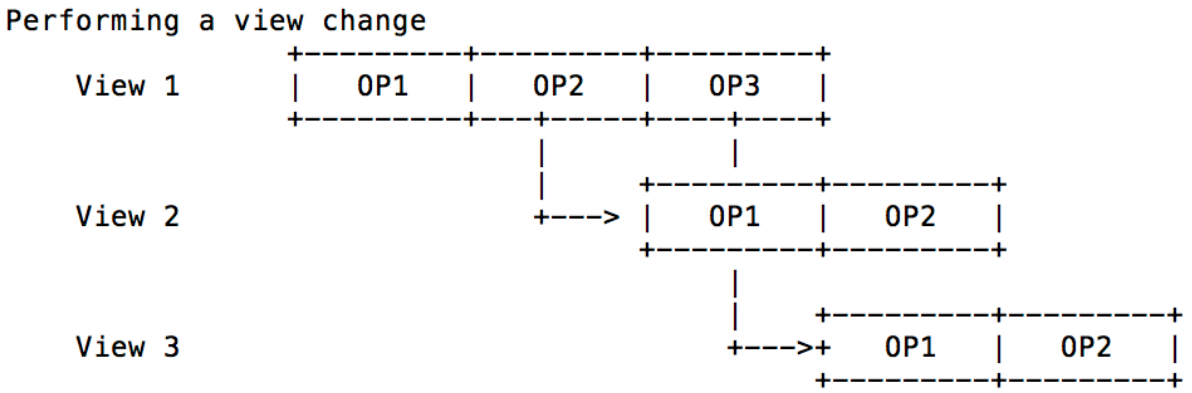
Two types of statement are safe to vote on:

* *Irrefutable statements:* while relevant, no one ever votes against
  In VR, at most one possible event for a given viewstamp
  So during view change, always free to vote for <vs, event>
    Either a cohort never acked any event at vs, so free to ack event
    Or a cohort already acked <vs, event>, and happy to repeat
* *Neutralizable statements:* indeterminate outcome won't prevent consensus (e.g. vote to form a new view)
  In VR, might fail to form a particular view
    Nodes might timeout, move on to higher view
    View manager might fail
    Or might try to form with too low a previous viewstamp
      [might depends on exact instantiation of viewstamped replication]
      E.g., f+1 nodes accept to form view vid with normal_viewid vs
      1 of the f fails, and remaining (slower) cohorts saw vs' > vs
  No big deal, just "neutralize" the view by trying a higher viewid

Viewstamped replication (and Raft) from 50,000 feet:

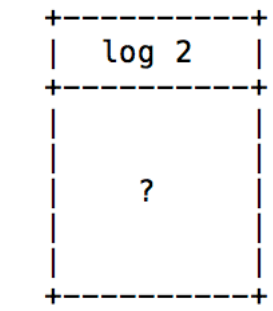
Have a single leader sequentially number all operations <viewstamp, op>...
Have nodes vote on the operation corresponding to each viewstamp
  Only one leader per view, so this mapping is irrefutable
  Hence liveness guaranteed until you lose (or think you lost) leader
Lost leader? Must vote on (new viewid, new leader, last included viewstamp)
  By irrefutability, stale nodes can always increase last include viewstamp!
  Leader vote could be indeterminate, but it's neutralizable!
  A failed view doesn't preclude successful views with higher viewids

Performing a view change



For View 2: no other operations contest fact that view2 is off view2
(voted to agree that view2 is the one followed)
For View 3: too many nodes see OP3 and follow it, so OP2 is neutralized.

Neutralizing statements with Ballot Numbers




```
try
  vote ? = OP2
if gets stuck,
  vote ? = OP3
```

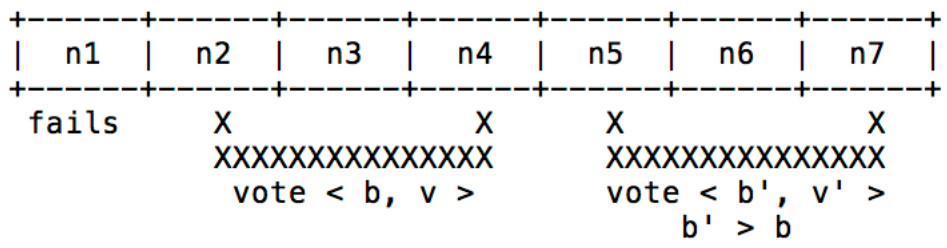
Now let's try a different approach: numbered ballots

Vote for log entries, but associate each vote with a ballot number
E.g., "I vote that log index i contains value v *in ballot #b*"
Approximate idea: Suppose ballot b is indeterminate
...neutralize it by proceeding with a higher-numbered ballot

The complication: An indeterminate ballot didn't necessarily fail

A node might irreversibly *externalize* a value and then fail
The network could lose the last messages sent by such a node
Yet once externalized, a value must be chosen (e.g., authorized purchase)

Example: 7 nodes (n1, ..., n7).



- n2-n4 vote for value v in ballot #b
- n5-n7 never heard of v, voted for v' in ballot #b' (b' > b)
- n1 died at some point, but might have voted

Now what? Both ballots b and b' are stuck. Did they fail?

Or did n1 vote for and externalize v? Or did it externalize v'?
Can't completely neutralize an indeterminate ballot!

Solution: Conservatively assume indeterminate ballots may have succeeded

Note, if v' = v above, there is no problem
So if ballot b indeterminate, ensure all future ballots use same value

The (single-decree) Paxos protocol:

* Paxos is all about making sure all successful votes and stuck votes are for

same value!
* Establish irrefutable mapping between ballot # and value
  Ballot = referendum on one value, not choice between candidate values
  Each ballot proposed by a single leader, has single value
  Embed leader's unique machine-id inside ballot number
* Before voting on value in ballot b, *prepare* b by checking previous ballots
  Leader broadcasts "PREPARE b"
  Cohorts reply "PREPARED b { NULL | <b_old, v_old> }"
    b_old is the last ballot in which cohort voted before b
    v_old is the value in ballot b_old
    Cohort promises conditions will still be true when acted on by leader
      ...implies promise never to vote for any ballot between b_old and b
  Ballot b prepared after getting PREPARED from a majority of nodes
  If all PREPARED messages have NULL, leader can use any value v
    -> Because you can assume the outcome was learned (new value will not contradict a nonexistent older value)
  Otherwise, highest b_old is indeterminate (or successful)
    Leader must set v = v_old corresponding to highest b_old
  (This prepare phase is used to ensure that if someone submits multiple
  ballots, they are the same number. So it's okay to submit multiple ballots!)
* Now can vote on value v
  Leader broadcasts "COMMIT b v" [a.k.a. "accept b" in paper]
  Cohorts reply "COMMITTED b"
  After collecting COMMITTED from majority of replicas, consensus on v
    (Learners record this outcome)

When is it safe for backups to apply message to replicated state machine?

After learning acceptance decisions of all acceptors and noting majority COMMIT.

Can add extra field "a" to ballot that is true or false:

if true, all future slots in logs are managed by a leader
(all at once, leader prepares log, and at steady state, send commit messages)

No need to tell backups about transaction succeeding, they will just rerun Paxos and the outcome is the only value they'll be able to choose.

If two nodes attempt to become leaders, the one with the higher ballot number will become leader; other node will simply adopt leader's ballot number.

Why is Paxos hard to understand?

Conjecture: it conflates irrefutable and neutralizable messages
Another way to view Paxos: Translate concrete messages to conceptual ones
The conceptual protocol:
  * commit b v: Vote for consensus on value v in ballot #b
  * abort b v: Vote that ballot #b not reach consensus on value v
  Invariant: For each b, can only vote "commit b v" for a single value v

Value is determined by leader of ballot b (whose id is embedded in b)
Makes mapping of b -> v irrefutable
Invariant: can only vote "commit b v" after a majority votes for
{ abort b' v' | b' < b && v' != v }
When such abort messages garner majority, we say (b, v) is prepared
The concrete to conceptual mapping
* PREPARED b <b_old, v_old>:
{ vote for "abort b' v'" | b_old < b' < b } [for all values v']
{ assert "abort b' v'" got majority | b' <= b_old && v' != v_old }
* COMMITTED b
vote for "commit b v", { "abort b v'" | v' != v }
v is (irrefutable) value from COMMIT b v
So Paxos uses same tools (irrefutable/neutralizable statements)
... but in much more complicated way

Simple way to use paxos with replicated state machine:

One paxos instance per log entry (slot), specified in messages:
PREPARE i b, PREPARED i b {NULL|...}, COMMIT i b, COMMITTED b

Can we optimize multi-paxos for only one round trip in common case?

Yes. Use same ballot number for all paxos instances
* PREPARE i b: prepares ballot b for all slots i' >= i
* PREPARED i b {NULL|<b_old,v_old>} a:
- <b_old, v_old> is last COMMIT sent for ballot b
- a is bool, if TRUE, means use NULL for "all future slots" (i' > i)
This is the common case; means all slots >= i are now prepared
If a is FALSE, then must issue PREPARE (i+1) b

When can you apply an log index to state machine and reply to client?

When leader receives COMMITTED i b from a majority.

How do cohorts know?

Can piggy back on new messages:
* COMMIT i b v i_min:
- i_min is says all slots i' < i_min committed with ballot # <= b
So cohort that voted COMMIT i b v can mark (i, v) as committed
Cohort that voted COMMIT i b' v' for b' < b can't commit v'
must contact peer to learn value of v; but this is not common case

Convenient extensions to avoid full protocol or other delays where unnecessary

* PREPARE i 0
Special ballot 0 probes slot without trying to become leader
* LEARN i v: Reply to request concerning already committed slot i
A single LEARN message enough to externalize v, no need to hold vote
Alternatively, LEARN i v =~ PREPARED i <INFINITY, v> FALSE
* STATUS i_min b: Reply to request with current slot but stale ballot number
Says, sorry, I've promised to ignore all ballots <= b
Note these are only optimizations, not required for correctness

Reconfiguration: How to add and remove servers?

Use the log! Have distinguished log values that add/remove nodes

Danger: Better not commit values unless you know what a majority is!
Better not even prepare values unless you know what a majority is!

Lamport's solution?

Log index i changes system composition for slot i+alpha
Places bound alpha on number of uncommitted requests (operations in flight)
Fill log with nops if you had fewer than alpha outstanding operations

Your humble instructor's solution:

Unlimited pipelining of normal requests
But only commit values in order (probably want this anyway)
Ensures you always know what majority is before deciding COMMIT majority
Don't pipeline *any* COMMIT messages behind COMMIT i b v that reconfigures
Ensures you always know ballot is prepared before sending COMMIT.
(NEVER SEND COMMIT MSG BEFORE BALLOT IS PREPARED!)

What kind of synchrony assumptions could guarantee liveness with Paxos/VR?

If you know range of message latencies, set timeout appropriately
If you don't, make more limited assumptions
Example: Assume message latencies don't grow exponentially
Can keep doubling timeout before starting new ballot/leader election

Dirty secret: Most people think Paxos and VR are same protocol

(Your instructor wrote paper about VR called Paxos Made Practical...oops)
See https://www.cs.cornell.edu/fbs/publications/viveLaDifference.pdf

Does Paxos have any advantages over VR/Raft? Maybe

Leader change shares more code paths with normal operation
Conjecture: may be easier to test, leave fewer weird corner cases
Greater symmetry makes it adaptable to less centralized situations
Topic of research I'll tell you about in last lecture

RAFT

Goal: consistent key-value store

- Want it to be consistent (like single machine) and deal with failures at the same time
- Recovers from server failure autonomously:
  * minority of servers fail: no problem
  * majority fail: lose availability but retain consistency

Replicated state machine

- Every machine has event log and voting permissions
- Consensus ensures proper log replication
- System makes server as long as majority of servers are up
- failure model: fail-stop (non-Byzantine) and delayed/lost msgs.

How is Consensus used in general?

- 2PC used to agree on database entries and replicate entire databases
- All clients connect to leader, who is supposed to ensure consistency with backups.

Raft Overview

1. Leader election
    a. select one of the servers to act as cluster leader (send heartbeats to remain leader)
    b. detect new crashes, or do not receive heartbeat before timeout -> become new leader candidate and start leader election, choose new leader
    c. heartbeats and timeouts to determine crashes
    d. randomized timeouts to avoid split votes
    e. majority voting to elect one server per term
2. Log replication (normal operation)
    a. leader takes commands from clients and applies them to its log
    b. leader replicates its log to other servers (overwriting inconsistencies)
    c. built-in consistency check simplifies how logs may differ
3. Safety
    a. only server w/ up-to-date log can become leader (only elect leaders with all committed entries in their logs)
    b. new leader defers committing entries from prior terms

Stuckness can happen when even # of backups vote for two candidates

- very rare edge case of edge case (leader crashes, multiple servers time out at same time, HAPPENS that vote splits across them)
- just let election time out and do it over.
- Check out visualization on https://raft.github.io/raftscope-replay/

When majority of cluster has entry

- leader can call for that entry to be committed.
- for other backups to learn entries, each backup maintains a "next index" that is backed up until a consistent value with leader is reached. Then the backup will copy entries from leader COMMITTED log until they match.

Voting Rule

- when voting, servers will send out lengths of their log entries. Longest log implies more up-to-date info and inclusion of entire committed logs.
- to be elected as leader, your log needs to be "better". Needs to contain all entries committed already.

DO NOT EXTERNALIZE (RETURN TO CLIENT) ANYTHING THAT IS NOT COMMITTED!

Harp

What are the goals of this work?

- Replicated network file system server with high
    - Durability, Availability, Performance

- Backwards compatibility with existing clients
- Backwards compatibility with existing on-disk file systems (VFS layer)

Why not just use viewstamped replication to implement file server?

- File systems are big; 2f+1 copies is expensive for surviving f faults
- Sacrifices compatibility with existing file systems
- Performance might not be good
    - File system layouts heavily optimized for access patterns
    - Somehow need to work client caching into system
    - VR's aid tracking is overkill, would likely hurt performance
    - More availability during view changes
    - Can bring rebooted nodes mostly up to date before interrupting service (VR reduces availability because it blocks responding to requestes until new view is switched to, Harp optimizes state transfer by getting up-to-date info before enacting view change, so it can continue responding to queries).
    - Optimizations for file reads (update atime in background)
- Semantics might not be as good
- Harp can survive simultaneous power failure (UPS not optional)
    - In contrast to VR, which will crash if too many machines crash at once.
- Harp maintains ordering of overlapping operations
    - Implies A-linearizability with TCP (or w. UDP if no packet loss)
- File systems not allowed to abort transactions like VR module groups
- Client compatibility? Maybe okay with server-side NFS-proxy

What is normal-case operation no failures?

- One designated primary, f backups, f witnesses
- Client sends request to primary
- Primary multicasts requests to backups
- Backups log requests, reply to primary
- Primary waits for ack from all f backups then replies to client
- In background, primary tells backups operation committed

Witnesses send/receive no messages, do absolutely nothing

Logs often appear below VFS layer (e.g., in ext3)--why does Harp use log?

- Logging is super fast because it doesn't go to disk
    - Rely on UPS and kernel hack to avoid wiping state in "soft crash"
- Helps ensure concurrent ops are applied in the same order everywhere
    - Enforces consistent order of linearizability
- Logs make it easy to bring machines up to date after network partition
- Weird hack in case FS code vulnerable to "packet of death" (sec 4.3)
    - Apply at primary first (to maintain invariant below)
    - Primary crashes? View change before applying at backup
    - View change causes witness to log, so no lost state if backup crashes

What are all the different log pointers held at each node (Fig 4-1)?

- top - most recent log entry
- CP - most recently committed (primary + all backups have it)
- AP - most recently applied at local node

- LB - most recent operation such that it + all prior changes applied to disk
- GLB - largest index that bounds LB at all nodes
- Invariant: GLB <= LB <= AP <= CP <= top

What operations lie between top and CP?

- Concurrent pending operations
- NFS clients use kernel threads to issue concurrent operations
- Also, could have multiple clients accessing server

Why not have CP == AP (apply as soon as all backups ack)?

- Ops are applied in background by a separate apply process
- Lowers client latency--primary replies as soon as CP passes request
- All writes async (AP advanced when writes issued, not when completed)

Why not just delay advancing AP until operations on disk, so LB == AP?

- For performance, want multiple concurrent disk writes from apply process
  - So apply process needs to remember where it is (AP)
- Also, want primary and backup to be able to overlap writes
  - "Packet of death" trick requires waiting for primary to apply op

Why track the GLB? Could delete log below LB

- Can't throw away log records before all backups apply them
- Might need log to bring other nodes up to date

What is needed for correctness in the face of server failures/view changes?

- Must ensure only one view active at a time (use majority)
- Must ensure no events lost across view changes (agree on last op)
- Must ensure f+1 copies of all committed events (promote witnesses)

What are steps of view change?

- Phase 0: Nodes bring themselves up to date after recovery
  - Suffered disk failure+replacement? Have to copy state from non-witness
  - No disk failure? Just get log from witness
- Phase 1: Coordinator solicits votes
  - At this point, Harp stops processing new client requests
  - Nodes ahead of coordinator send it missing log entries
  - Need f+1 nodes for new view, so can't lose committed op from last view
- Phase 2: Coordinator informs nodes of new view
  - Coordinator solicits votes
  - Votes will also contain previous view they were in, and the latest op I think happened
  - Coordinator writes new view number (majority voted) to disk
  - Coordinator brings other nodes up to date if missing log entries
  - If promoting witness, must send it all logs since GLB
  - Other nodes write log entry to disk
  - Witnesses demoted if enough backups in new view

What is the difference between a promoted witness and normal backup?

- Witness does not have file system state, just log since node failure

- Hence, can never truncate the log

What could go wrong if witnesses didn't log, but just voted on view changes?

- Would prevent simultaneous active views, but not prevent data loss
- E.g., designated primary P, designated backup B, designated witness W
  - Network partition causes B & W to form a view, execute some requests
  - B crashes and loses its disk
  - Network partition heals, P & W form view, but lost ops from previous view
  - (now have only f-1 backups. single copy is missing)
- Trick is: always imagine the node that failed was not the node you thought failed

How are read-only requests handled specially? (p. 5)

Primary can reply immediately without talking to backups

"they are serialized at the CP"--why?

Anything past CP might not complete successfully after view change
E.g., backup never heard of it, then formed view with witness
Meanwhile client timed out and gave up
Never want to expose uncommitted state
Don't want to show clients a write that never happened!

But what if backup and witness already formed a new view?

Primary of old view could violate linearizability--how?
New view could complete a write operation
Meanwhile, primary of old view replies to read request with stale data
How to avoid? Assume roughly synchronized clocks for primary lease
Backups promise not to form new view for delta milliseconds

Is reading a file a read-only file system operation?

Not technically, because you are supposed to update the atime
But maybe we don't care if atime not updated in exceptional circumstances
So Harp offers a second "weaker" mode of operation
Allows small chance that atime update could get lost with failure

Is the file system VFS layer a deterministic replicated finite state machine?

I.e., apply same operation on two identical file systems, get same result?
No, because file times may not be identical
(fully identical results are expected of deterministic replicated FSMs)
How does Harp address? change VFS to allow caller to specify mtime (p. 9)

What's in an event record (Sec 4.4-4.5)?

A call to execute on the file system state
Enough information to describe the "outcome" of the call completely

Why is this tricky for file permissions?

E.g., write A (permission denied), chmod A
If you replay this log a second time, get different result
(permission no longer denied)

Why is this tricky for file creation (Sec 4.4, p. 8)?

Directory state is more than just file name -> inode mapping
At the time, you could read raw directory structures with "cat /dir/"

Even today, files stored in a particular order
Cookies/offsets used to encode entry positions
All of these things depend on order of directory operations

CREATE B, CREATE A, UNLINK A, CREATE A, UNLINK B != UNLINK A, CREATE A, UNLINK B

Replay just the last 3 and will get different result

Note simpler example might just be inode number
New directories may choose a different inode number than expected

Why not store applied actions to log to avoid situation above?
Update LB in log to know exactly what has been applied (an async action)
Cannot guarantee all info about applied actions will be written to log in advance of a crash.

Multiple things could go wrong here:

Harp has to reply to requests at CP, not AP
So needs to be able to predict outcome before applying event record
Harp must ensure backups maintain identical state

What's the solution?

Keep shadow state for inodes and directory pages (Sec 4.4, p. 9)
Also presumably need to bypass VFS layer for directory ops as well

Why is fsck a problem? (Sec. 4.3, p. 8)

fsck: file system scavenger program, run occasionally on reboot
Might have filesystem produce different outcome than that recorded in the log
Makes it hard to keep clients in identical state.
Attempt to avoid fsck on system except for extreme conditions
Completely messes up outcomes like position in directory entry

How to organize multiple file systems with Harp?

Spread the load so nodes are primary, backup, witness for different FSes

What happens to duplicate RPCs (same xids, Sec. 4.5)?

Client sends request to P, P executes request or sends it to B, P dies,
B becomes P, C retransmits request (not idempotent).
Need to store client entry in event entry, and cache result to ensure consistent result
In single NFS server, should cache reply to avoid re-executing
Harp must embed RPC xids in replies so new primary can build replay cache

What if disk write gets corrupted?

PBFT will fix this with digests :)

What is comparison point for evaluation? Single NFS server

Is this a fair comparison point?
Could also have compared to single NFS server with UPS hack
Pro argument: People tolerate existing NFS performance
Harp shows availability/reliability is possible at existing performance
Don't need to beet hypothetical world's fastest NFS server to be useful
Con argument: Maybe the UPS story is scary

What if UPS fails and you don't learn until it's too late?
What if log somehow corrupted during a reboot?
Today's datacenters do often have UPSes, so in retrospect looks okay

Why graph x=load y=response-time?

Why does this graph make sense?
Why not just graph total time to perform X operations?
One reason is that systems sometimes get more/less efficient w/ high load.
And we care a lot how they perform w/ overload.

Why does response time go up with load?

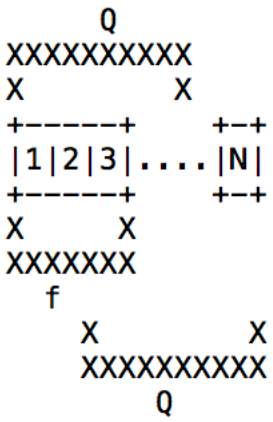
Why first gradual...
Queuing and random bursts?
And some ops more expensive than others, cause temp delays.
Then almost straight up?
Probably has hard limits, like disk I/Os per second.

Practical Byzantine Fault Tolerance

Suppose you have N replicas, f of which might crash (non-Byzantine failure)

What quorum size Q do you need to guarantee liveness and safety?

- Liveness: (or pseudo-liveness, i.e., avoiding stuck states)
  - There must be a non-failed quorum (*quorum availability*)
  - Hence: Q <= N – f
- Safety: Any two quorums must intersect at one or more nodes
  - Otherwise, two quorums could independently accept operations, diverge
  - This property is often known as the *quorum intersection* property
  - You want to make sure that there's at least an overlap size of 1 between
  - two quorum regions Q, as shown below:




```
Hence: 2Q - N > 0
So: N < 2Q <= 2(N - f)
Note highest possible f: N < 2N-2f; f < N/2
And if N = 2f + 1, smallest Q is 2Q > 2f + 1; Q = f + 1
```

Now say we throw in Byzantine failures...

Say you have N nodes, f of which might experience Byzantine failure.

First, how can Byzantine failures be worse than non-Byzantine?

Byzantine nodes can make contradictory statements to different nodes

E.g., violate voting rules by voting twice for different outcomes

Consequences

Risks driving non-failed nodes into divergent states (no safety)

Risks driving non-failed nodes into "stuck states"

E.g., cause split vote on seemingly irrefutable statement

VR example: nodes can't agree on event associated with last viewstamp

What quorum size Q do we need in Byzantine setting?

f is # of Byzantine failures

Liveness: Q <= N - f

- As in non-Byzantine case, failed nodes might not reply
- Second threat to liveness: beyond safety, a bad node can also create a stuck state.

Safety: Quorum intersection must contain one *non-faulty* node

- Idea: out of f+1 nodes, at most f can be faulty (can't get away with just 0 < faulty nodes like in non-Byzantine failure!)

```
    Hence:  2Q - N > f    (since f could be malicious)
So: N + f < 2Q <= 2(N - f)
Highest f:  N+f < 2N-2f; 3f < N; f < N/3
And if N = 3f + 1, the smallest Q is:
  N + f < 2Q; 3f + 1 + f < 2Q;  2f + 1/2 < Q; Q_min = 2f + 1
```

So how does PBFT protocol work?

Number replica cohorts 1, 2, 3, ..., 3f+1
Number requests with consecutive sequence numbers (not viewstamps)
System goes through a series of views
  In view v, replica number v mod (3f+1) is designated the primary
  Primary is responsible for selecting the order of operations
    Assigns an increasing sequence number to each operation
In normal-case operation, use two-round protocol for request r:
  Round 1 (pre-prepare, prepare) goal:
    Ensure at least f+1 honest replicas agree that
      If request r executes in view v, will execute with sequence no. n
  Round 2 (commit) goal:
    Ensure at least f+1 honest replicas agree that
      Request r has executed in view v with sequence no. n

Protocol for normal-case operation

```
Let c be client
    r_i be replica i, or p primary, b_i backup i
    R set of all replicas

    _Kc is client signature, so you can verify REQUEST integrity
    t is timestamp to prevent replay attack
    o is operation
    m is the entire first msg sent from c -> p below
    n is sequence number

  c -> p:  m = <REQUEST, o, t, c>_Kc
  p -> R:  <PRE-PREPARE, v, n, d>_Kp, m     (note d = H(m))
b_i -> R:  <PREPARE, v, n, d, i>_K{r_i}
[Note all messages signed, so henceforth < > means implicit signature.]

replica r_i now waits for PRE-PREPARE + 2f matching PREPARE messages
   puts these messages in its log
   then we say prepared(m, v, n, i) is TRUE
```

Note:

If prepared(m, v, n, i) is TRUE for honest replica r_i then prepared(m', v, n, j) where m' != m FALSE for any honest r_j

  Why? Because each of r_i, r_j has 2f+1 signed [pre-]prepare messages
    Given 3f+1 replicas, messages at r_i and r_j must share f+1 sources
    But at most f nodes are malicious, so one of those won't double vote
    So the two sets of 2f+1 agreeing messages must agree with one another
  Means no other operation can execute with view v sequence number n

Votes in PREPARE messages can be sent out tentatively across a short

time period
BUT only clients will know of COMMIT status, not necessarily entire cohort.
Cohorts learn through the extra round of COMMITS. That's why the third round is necessary.

Are we done? Just reply to client? No

Just because some other m' won't execute at (v,n) doesn't mean m will
Suppose r_i is partitioned right after prepared(m, v, n, i)
Suppose no other replica received r_i's prepare message
Suppose f replicas are slow and never even received the PRE-PREPARE
No other honest replica will know the request prepared!
If p fails, m' != m might get executed at (v+1,n)
Would be bad if r_i had said m committed in slot n when it didn't
f bad nodes could "join" r_i, falsely convincing client m committed at n
f nodes could crash, won't have 2f+1 to agree on sequence number n

So we say operation doesn't execute until

prepared(m, v, n, i) is TRUE for f+1 non-faulty replicas r_i
We say committed(m, v, n) is TRUE when this property holds (b_i -> R)

So how does a replica *know* committed(m, v, n) holds?

Add one more message:

r_i -> R: <COMMIT, v, n, d, i> (sent only after prepared(m,v,n,i))

replica r_i waits for 2f+1 identical COMMIT messages (including its own)
committed-local(m, v, n, i) is TRUE when:
prepared(m, v, n, i) is TRUE, and
r_i has 2f+1 matching commits in its log

Note: If committed-local(m, v, n, i) is TRUE for any non-faulty r_i
Then means committed(m, v, n) is TRUE.
r_i knows when committed-local is TRUE
So committed-local is a replica's way of knowing that committed is TRUE

r_i replies to client when committed-local(m, v, n, i) is TRUE
Client waits for f+1 matching replies, then returns to client
Why f+1 and not 2f+1?
Because of f+1, at least one replica r_i is non-faulty
So client knows committed-local(m, v, n, i) (even if it doesn't know i)
Which in turn implies committed(m, v, n)

Garbage collecting the message log

make periodic checkpoints
Broadcast <CHECKPOINT, n, d, i>, where d = digest of state
A stable checkpoint has state + proof comprising 2f+1 signed CHECKPOINTs
restrict sequence numbers are between h and H
h = sequence number of last stable checkpoint
H = h + k (e.g., k might be 2 * checkpoint interval of 100)

Why? Don't want bad primary exhausting sequence number space
delete all messages below sequence number of stable checkpoint

View changes

When client doesn't get an answer, broadcasts message to all replicas
If a backup notices primary is slow/unresponsive:
- broadcast <VIEW-CHANGE v+1, n, C, P, i>
v+1 is next view you want to change to
n is sequence number of last stable checkpoint
C is 2f+1 signed checkpoint messages for last stable checkpoint
P = {P_m} where each P_m is signed PRE-PREPARE + 2f signed PREPARES
i.e., P is set of all PREPAREd messages since checkpoint
+ proof that the messages really are prepared
C and P are proof of stable checkpoint.

When primary of view v+1 sees 2f signed VIEW-CHANGE messages from others, then
- New primary broadcasts <NEW-VIEW, v+1, V, O>
V is set of at least 2f+1 VIEW-CHANGE messages (including by new primary)
O is a set of pre-prepare messages, for operations that are:
- after last stable checkpoint (assume everyone has executed up to sequence number n)
- appear in the set P of one of the VIEW-CHANGE messages
O also contains dummy messages to fill in sequence number gaps

Replicas may optain any missing state from each other
(e.g., stable checkpoint data, or missing operation, since reissued pre-prepare messages only contain digest of request)

What happens if primary creates incorrect O in NEW-VIEW message?

E.g., might send null requests for operations that prepared
Other replicas can compute O from V, and can reject NEW-VIEW message
What happens if primary sends different V's to different backups?
Still okay, because any committed operation will be in 2f+1 VIEW-CHANGE msgs of which f+1 must be honest, so at least one member of V will have operation
So new primary cannot cause committed operations to be dropped
Can only drop operations for which client has not yet seen the answer

How does PBFT deal with non-determinism (Sec. 4.6)? Two options

Option 1: Primary selects value, piggybacks on PRE-PREPARE
"replicas must be able to decide deterministically whether... correct"
Option 2: Add an extra round to protocol for backups to determine value
E.g., backups nominate non-deterministic candidate values
Value chose is average, or one with highest hash out of 2f+1 nominations
Primary must include 2f+1 nominations as proof this is valid
What does BFS use for file mtime? option 1
Weird because not actually deterministically checkable
But turns out it works anyway (homework question)

What is the tentative reply optimization?

Want to shave one network transmission latency from end-to-end client time
r_i tentatively executes m, sends tentative reply once prepared(m,v,n,i)
which is before committed-local(m,v,n,i)
Client accepts 2f+1 matching tentative replies
So f+1 of those replies must be from honest nodes
And at least 1 of those f+1 will be part of 2f+1 forming any new view
So that 1 node will make sure operation makes it to new view
So why do we even need COMMIT messages?
Tentative replied convince client that committed(m,v,n)
But replicas don't know this!
So can't determine outcome of subsequent dependent operations,
Can't compute checkpoints, etc.
Can tentatively executed requests be undone?
Yes, if there's a view change
But then client won't have 2f+1 tentative replies, so okay to undo

Strict vs RO

strict uses different permissions to access directory (x not r)
Does not change atime on directory
Remember,
atime = access (read) time,
mtime = time of last modification,
ctime = last time you changed attributes (e.g. deliberately setting mtime)
anyways, RO is 2% faster anyways (relying on replication)

How are read-only (RO) requests optimized?

Send directly to replicas, replicas send reply directly to client
Client waits for 2f+1 matching replies, or restarts as read-write request
Replicas must serialize after all prior tentative requests. Why?
For linearizability, RO request might not overlap tentatively executed one
What if replicas serialize at different points
Might not get 2f+1 matching replies, but that's okay
How would this break if REPLY didn't include view number?
Could get 2f+1 matching replies from different views
Problem if all replied result from tentative operations and none commit
Saved by the fact that only one op can prepare for a single view/seq#

BFS-nr: Quorum size of 1. This tells you pure overhead costs for replication

What about compromised clients? (p. 2)

Results of faulty client operations must be consistent
Can't violate server side invariants
So can rely on authentication and authorization
Just like faulty clients with non-replicated service

How do authors evaluate system:

Microbenchmark: null op with empty or 4K request/response size
End-to-end file system benchmark vs. NFS and non-replicated BFS
Are these the right things to measure?

End-to-end benchmarks are good, microbenchmark helps understand them
Could ask for more benchmarks, or more services than BFS
But compared to prior work, already these results very convincing
What's the deal with r/o lookup?
On NFS lookup RPC, BFS updates the directory utime, which is expensive
Is that right? No! Only need "x" permission on directory for lookup
Need to read/getdents ("r") a directory or file to change utime
Verified on linux and BSD, lookup does not affect directory utime
So the authors violated semantics and needlessly penalized themselves!

How might we extend PBFT to tolerate more than (n-1)/3 failures over lifetime?

- detect failed replicas using proactive recovery
  - recover the system periodically, no matter what
  - makes bad nodes good again
- tricky stuff
  - an attacker might steal compromised replica's keys

Honey Badger

Recall FLP (1983):

No deterministic consensus protocol can achieve all three of
safety, liveness, and fault-tolerance in an asynchronous system.

So far, we have mostly discussed sacrificing liveness

Idea: guarantee safety and fault tolerance, terminate in practice
Protocol must not get stuck, so always maintain hope of terminating

Another possibility: attack asynchronous system assumption

Suppose all nodes can always communicate within T seconds by everyone's clock
(So can communicate within T-epsilon real seconds to account for skew)

Famous Byzantine generals protocol (Lamport,Shostak,Pease) signed variant:

* Problem statement:

General G_0 (possibly a traitor) broadcasts an order (attack or retreat)
Other generals compare notes, reach consensus on order
If generals don't agree on order, will suffer catastrophic losses
At most f faulty generals (including possibly G_0)

* Protocol

```
Warm-up: at most 0 faulty generals, want consensus on G_0's order
  Have general G_0 digitally sign and broadcast the order
  All other generals wait T seconds, get signed order, execute it
Normal case: f faulty generals trying to mess up consensus
  Go through f+1 rounds (0,...,f) of T seconds each:
    Round 0: General G_0 broadcasts signed order v
    Round 1: Each other G_i signs G_0's signed order, broadcasts
    Round r: For each message G_i received in round (r-1) with order v
      If G_i hasn't yet seen a message with order v
         then G_i adds its own signature to message (now has r+1 nested sigs)
              broadcasts message to generals that haven't signed yet
      Reject messages with fewer than r distinct signatures in round r
  After round f, will have 0 or more messages each containing some order v
    General G_i knows that for each of these messages, either:
      - G_i broadcast the message to all generals that hadn't seen it, or
      - f+1 generals signed the message, at least one of them is honest
        the honest general must have broadcast the message to everyone
    So all honest generals will have same set of orders after round f
      Will have exactly 1 order if G_0 is honest, just execute it
      If 0 orders (e.g., G_0 crashed) just execute some default order v
      If >1 orders, combine deterministically--e.g., take median value
```

Interesting properties of Byzantine generals protocol:

Survives f failures out of n=f+2 nodes (not bound by 1/3 threshold)
(Technically works for n < f+2, but then problem is vacuous)
Completely loses safety if synchrony assumption violated

Yet another possibility: partial/weak synchrony (like PBFT)

Guarantee safety always
Guarantee liveness under certain synchrony assumptions
Never get stuck, so can always "fix" network and make progress
This is the "winner" in terms of what most real systems use

What if we play with "deterministic" instead of "asynchronous" in FLP?

Recall FLP proof considers delivering messages m and m' in either order
Assumes if different recipients, either order leads to same state
But logic only holds if messages are processed deterministically
We've already seen protocol that "never gets stuck"
Means there's always some network schedule that leads to termination
So keep trying "rounds" (views, ballots, terms, etc.) until one terminates
If network were random, we could talk about round termination probability
Unfortunately, network is hard to model / controlled by adversary
Can we instead make probability dependent on nodes' random choices?

In response to FLP, Ben Or proposed the following protocol in 1983:

Goal: consensus on one bit (a.k.a. asynchronous binary agreement, or ABA)
Assumes at most f faulty nodes out of N>5f (sic, not N>3f) nodes

```
Each node i starts with input bit x_i, then executes:
  int x = x_i; // i's input bit
  for (round = 0;; ++round) {
    broadcast <VOTE, round, x>
    wait for N-f VOTE messages in round (including i's own)
    if more than (N+f)/2 VOTEs have same value v
      then broadcast <COMMIT, round, v>
      else broadcast <COMMIT, round, ?>
    wait for N-f COMMIT messages in round (including i's own)
    if more than f+1 COMMIT messages have same value v != ?
      then set x=v; if more than (N+f)/2 COMMIT v
                          then output x as consensus value
      else set x to a random bit
  }
```

Analysis:

Note: > (N+f)/2 nodes includes a majority of non-faulty nodes
  Majority of non-faulty nodes is > (N-f)/2 non-faulty nodes
  Plus f faulty nodes means > (N-f)/2 + f = (N+f)/2
Hence, in a round, all non-faulty nodes must COMMIT same value unique v != ?
  But some or all non-faulty nodes may COMMIT ? instead
If you receive f+1 COMMIT v != ?, you know v must be the unique COMMIT value
  After all, at most f of those nodes will be lying
If you receive C COMMIT v and C > (N+f)/2
  Each other node i will see at least C-2f COMMITs for v
    because f of your C could double-vote, and another f could be slow to i
  But C-2f > (N+f)/2 - 2f = (N-3f)/2 > (5f-3f)/2 = f [since N>5f]
    So every other non-faulty node will see f+1 COMMITs for v
    Means all other non-faulty nodes guaranteed to terminate in next round
If you flip a coin
  Could luck out and have all non-faulty nodes flip same value
  So protocol guaranteed to terminate eventually with probability 1!

So why not use Ben Or instead of PBFT?

Only agrees on one bit, not arbitrary operation
Exponential expected number of rounds required when flipping coins

Rabin's idea (1983): What if everyone flipped the same *common coin*?

Can use a deterministic threshold digital signature scheme
  E.g., threshold RSA with full-domain-hash
Sign message "(consensus instance, round number)"
  Take low bit of SHA256(signature) as common coin value
Attacker doesn't learn coin until too late to send COMMITs
Rabin's trick: Randomize vote threshold between N/2 and (N-2f)
  Bad network can't split over threshold if it can't predict threshold
  But all nodes can use the same threshold based on common coin
But see MostÃ©faoui [43] for a much better protocol using this idea
Caveat: common coin requires trusted dealer or fancy crypto
  Somehow need to get nodes shares of the same private key
  (Or in Rabin's case direct shares of a sequence of one-shot random bits)

What is asynchronous Reliable Broadcast?

Set of nodes {P_i} including distinguish sender P_S that has input h
If P_S is non-faulty, all non-faulty nodes deliver h
else, either all non-faulty nodes deliver same h', or none terminate

Boils down to:

- agreement - all non-faulty node outputs are identical
- totality - all non-faulty nodes output a value or none terminate
- validity - if P_S non-faulty, then all non-faulty nodes output h

How would you do simple (Bracha-style) RBC w/o erasure coding, N > 3f?

```
P_S broadcasts VAL(h)
P_i receives VAL(h), broadcast ECHO(h)
P_i receives N-f ECHO(h) messages, broadcasts READY(h)
P_i receives f+1 READY(h), broadcasts READY(h) [if hasn't already]
P_i receives 2f+1 READY(h), delivers h
```

Why does this work?

N-f nodes includes majority of non-faulty nodes
So READY from all non-faulty nodes will have same value h => agreement
If P_S non-faulty, will all contain P_S's input h => validity
If 2f+1 nodes send READY(h), f+1 will be non-faulty
Those f+1 will convince all other non-faulty nodes to broadcast READY(h)
Since N>3f, will get 2f+1 properly broadcast READY(h) => totality

How does erasure coding improve efficiency?

What if h is big? P_S will have to send many copies
Instead, make h a cryptographic hash
But use merkle tree so can individually verify any block of message from h
Erasure coding: make n>k shares of k-block msg, so any k reconstruct msg
Change protocol to broadcast VAL(h,b_i,s_i), ECHO(h,b_i,s_i)
s_i is share of message, b_i is proof that it is in hash tree with root h
Wait for N-f ECHO messages (guaranteed to get if 2f+1 READY(h)), reconstruct

Why doesn't RBC give us consensus with following straw man?

Every node RBCs its own input (N instances of RBC)
Non-faulty nodes guaranteed to converge on same set of inputs
Combine inputs deterministically (e.g., output median value)
Red flag: can't be right, would violate FLP (not randomized)
Problem: guaranteed to converge, but nodes don't know when
Might not get RBC from faulty senders, but don't know which are faulty
A node risks outputting median value before hearing all inputs

What is asynchronous common subset (ACS)?

Each node P_i has input v_i, must output some set of values
Goal: all non-faulty nodes output same set V of values
V must encompass inputs of N-2f nodes

Boils down to:

- validity - any non-faulty node output contains N-2f non-faulty node inputs
- agreement - if any non-faulty node outputs set V, all output same set V
- totality - if N-f non-faulty nodes get input, all non-faulty produce output

How to build ACS from RBC and ABA?

Start out like straw-man above: each P_i RBCs v_i (N instances of RBC)
But now use N instances of ABA to pick at least N-f of these RBCs:

```
while (fewer than N-f RBCs have delivered a value
       && fewer than N-f ABA instances have output 1) {
  if (RBC_j delivers v_j)
    Supply 1 as input to ABA_j
}
Supply 0 as input to any remaining ABAs we haven't started yet
Output { v_j | ABA_j output 1 }  [waiting for RBCs if necessary]
```

Why does this ACS work?

RBCs and ABAs have identical outputs at all non-faulty nodes => agreement
N-f RBCs will eventually deliver a value (by totality of RBC) => totality
All nodes will exit the while loop
If ABA_j == 1 at any non-faulty node, then RBC_j will deliver v_j
At least N-f ABA must output 1 => validity
Because N-2f must correspond to non-faulty nodes

How to enhance throughput?

Try to run RBC on different values by picking randomly from large buffer
How does this enable censorship w/o threshold encryption?
Say attacker controls network and f nodes
Ensure whatever node RBCs victim transaction is always slow
Allows attacker to keep some transaction from ever being included
How does threshold encryption solve?
Attacker doesn't know who has RBCed what transactions until it is too late

Would you use HoneyBadgerBFT for a network file system like BFS?

No - very high latency (10s of seconds) would give unusable performance
Also doesn't take advantage of physical-layer multicast

Why use HoneyBadgerBFT instead of PBFT?

High throughput with many replicas, big batch sizes
No need to worry about tuning timeouts