Paxos

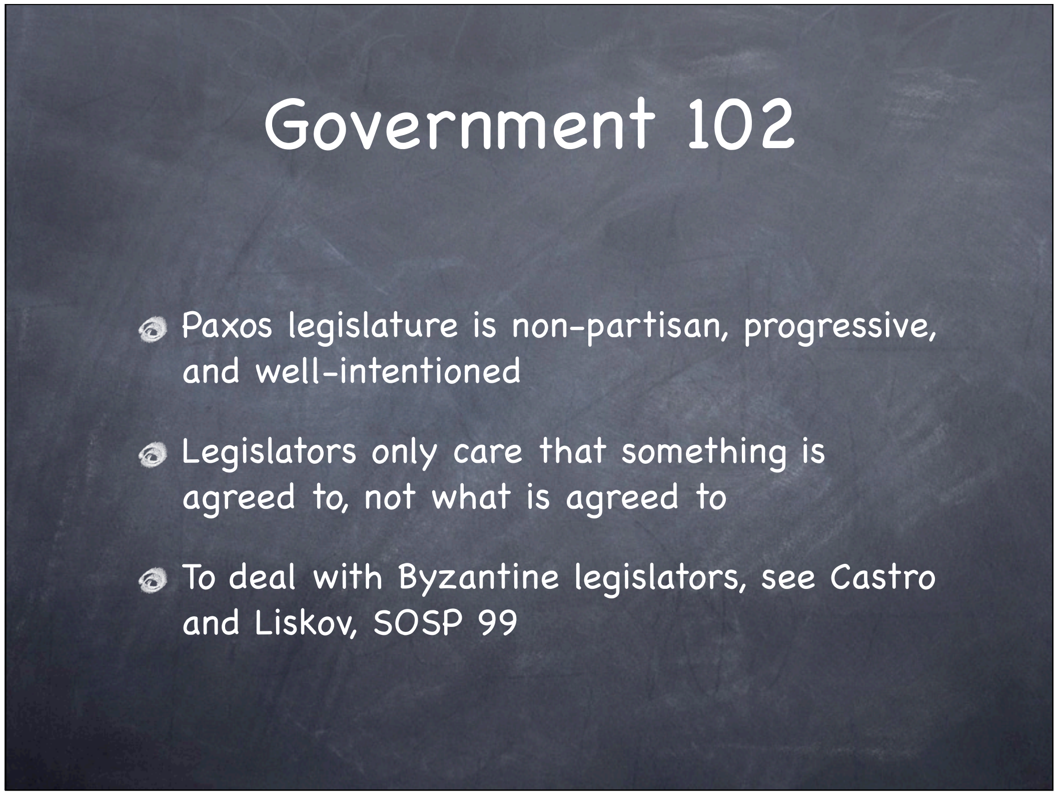
(slide deck: Lorenzo Alvisi)

The Part-Time Parliament

- Parliament determines laws by passing sequence of numbered decrees
- Legislators can leave and enter the chamber at arbitrary times
- No centralized record of approved decrees– instead, each legislator carries a ledger

Government 101

- No two ledgers contain contradictory information
- If a majority of legislators were in the Chamber and no one entered or left the Chamber for a sufficiently long time, then
  - any decree proposed by a legislator would eventually be passed
  - any passed decree would appear on the ledger of every legislator

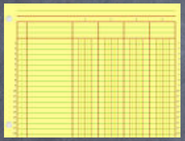


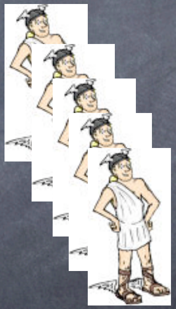
Government 102

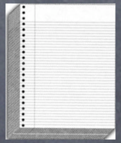
- Paxos legislature is non-partisan, progressive, and well-intentioned
- Legislators only care that something is agreed to, not what is agreed to
- To deal with Byzantine legislators, see Castro and Liskov, SOSP 99

Supplies

Each legislator receives


ledger


lots of messengers

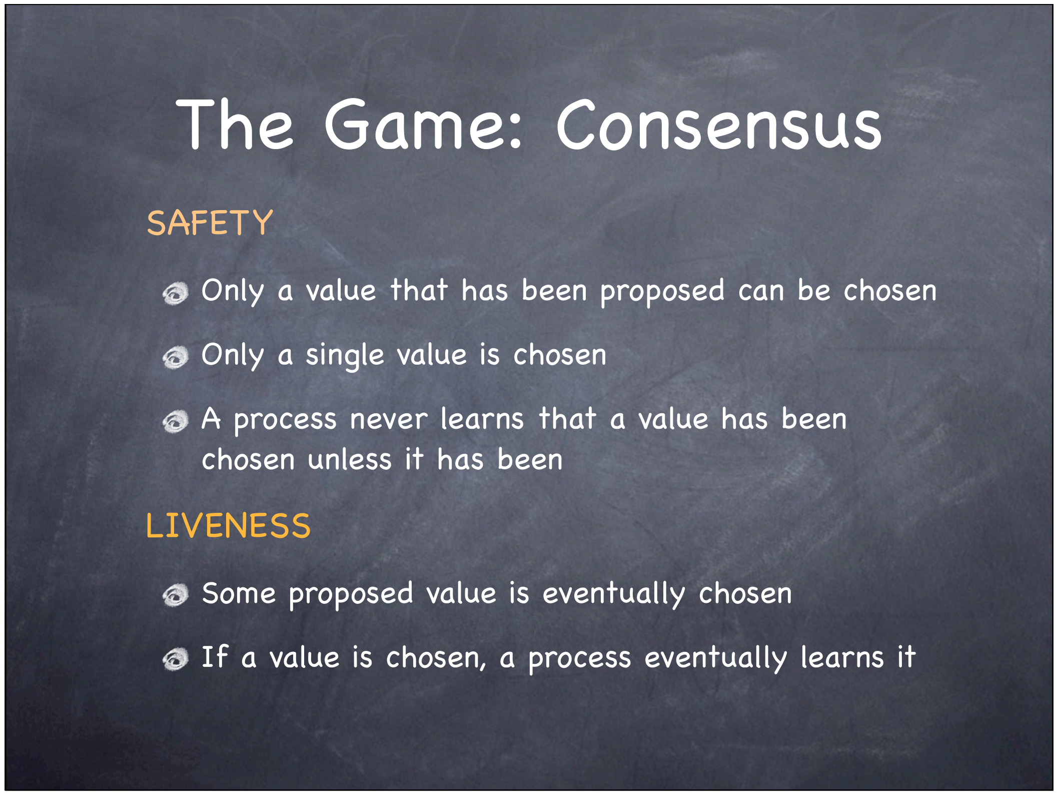

scratch paper

pen with indelible ink

hourglass

Back to the future

- A set of processes that can propose values
- Processes can crash and recover
- Processes have access to stable storage
- Asynchronous communication via messages
- Messages can be lost and duplicated, but not corrupted



The Game: Consensus

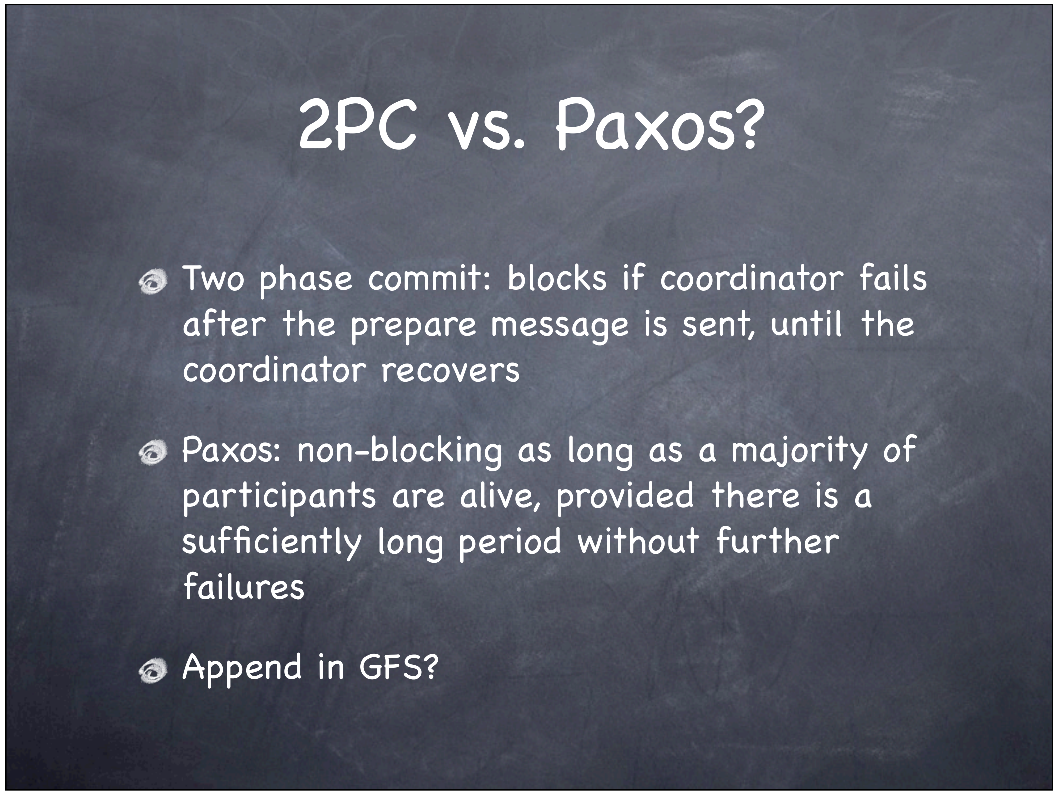
SAFETY

- Only a value that has been proposed can be chosen
- Only a single value is chosen
- A process never learns that a value has been chosen unless it has been

LIVENESS

- Some proposed value is eventually chosen
- If a value is chosen, a process eventually learns it

Consensus about one value can be generalized to consensus about a sequence of values: the sequence of operations to apply to a replicated state machine. Essentially, consensus about "what is the next operation to apply?"
Note that in general, we don't care what the order of operations is, as long as there is an order, we all agree on it, and we can continue to make progress during failures.



2PC vs. Paxos?

- Two phase commit: blocks if coordinator fails after the prepare message is sent, until the coordinator recovers
- Paxos: non-blocking as long as a majority of participants are alive, provided there is a sufficiently long period without further failures
- Append in GFS?

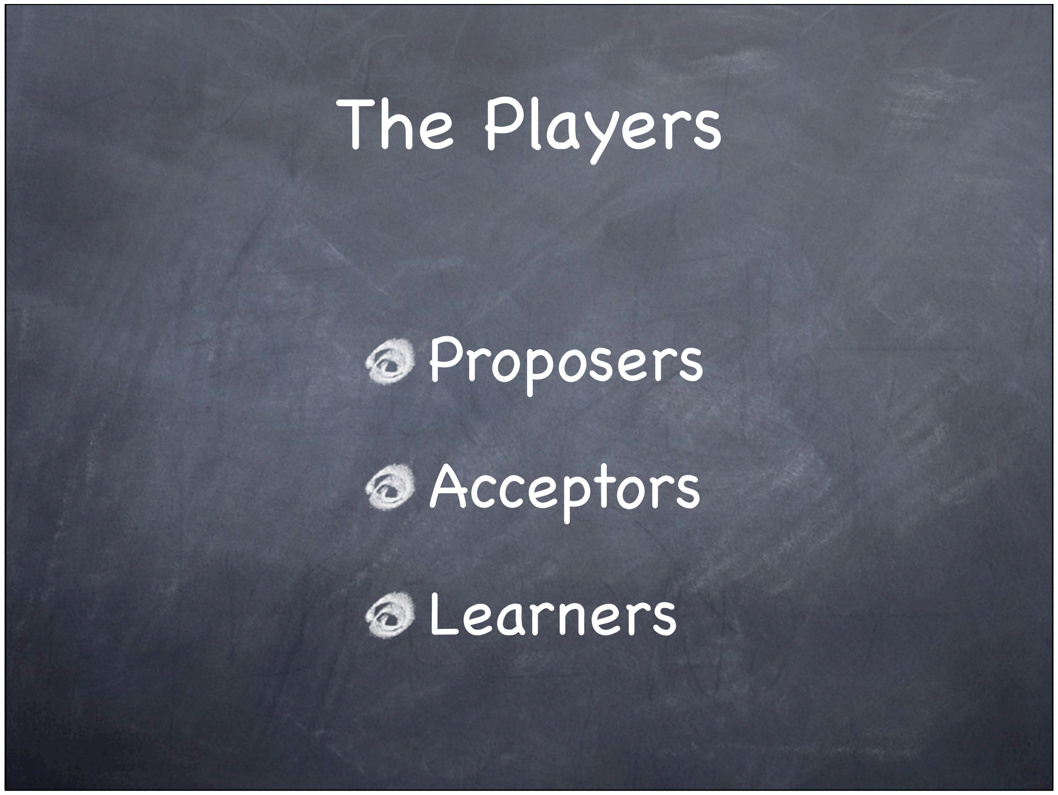
Of course, we can't get both safety and liveness. So Paxos does safety, and liveness where possible: if a majority is still up, and there are no failures for a sufficient period of time.

How do these constraints compare to 2PC, where "chosen" = "commit"?

Realize how hard it is to be non-blocking! A majority can decide to do an operation, and all but one of those nodes could fail, replaced by nodes that have no idea whether something was approved, yet despite that, we still need to be able to make progress.

How? The one that remains cannot possibly know how the others had voted. The key is: it doesn't matter how they voted.

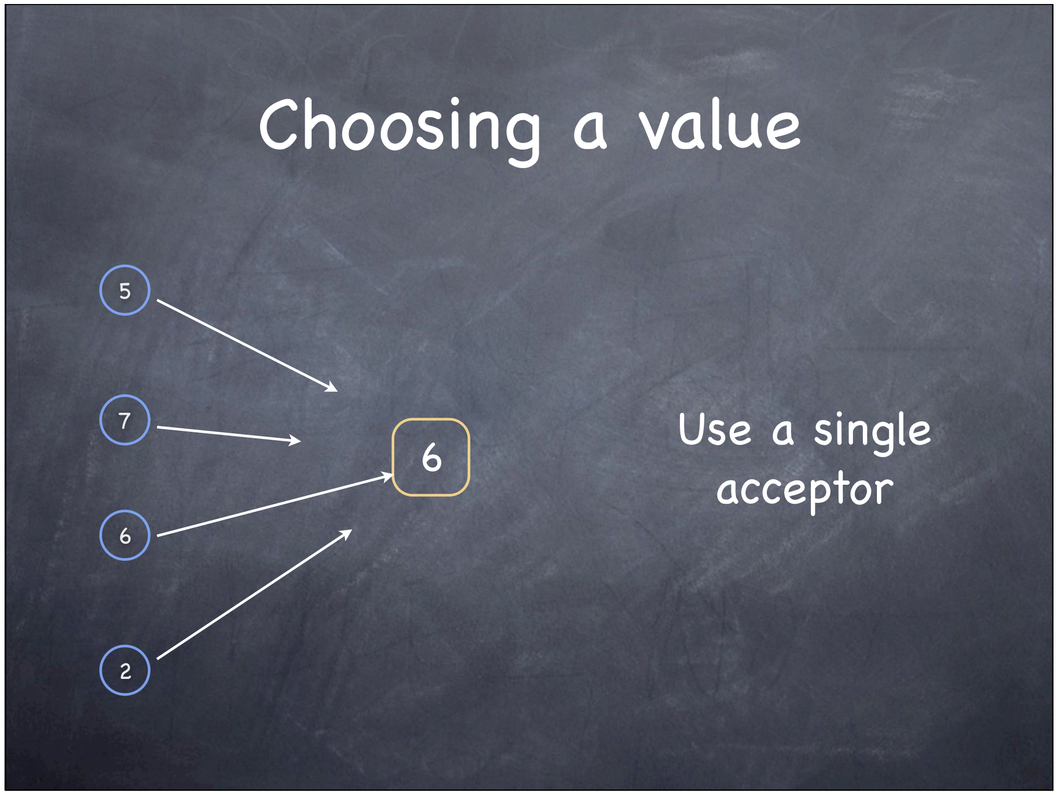
What about GFS? if there are failures, don't know if append was committed at all of the replicas. So we make progress by redo'ing the operation, and making the applications work even if append happens more than once.



The Players

- Proposers
- Acceptors
- Learners

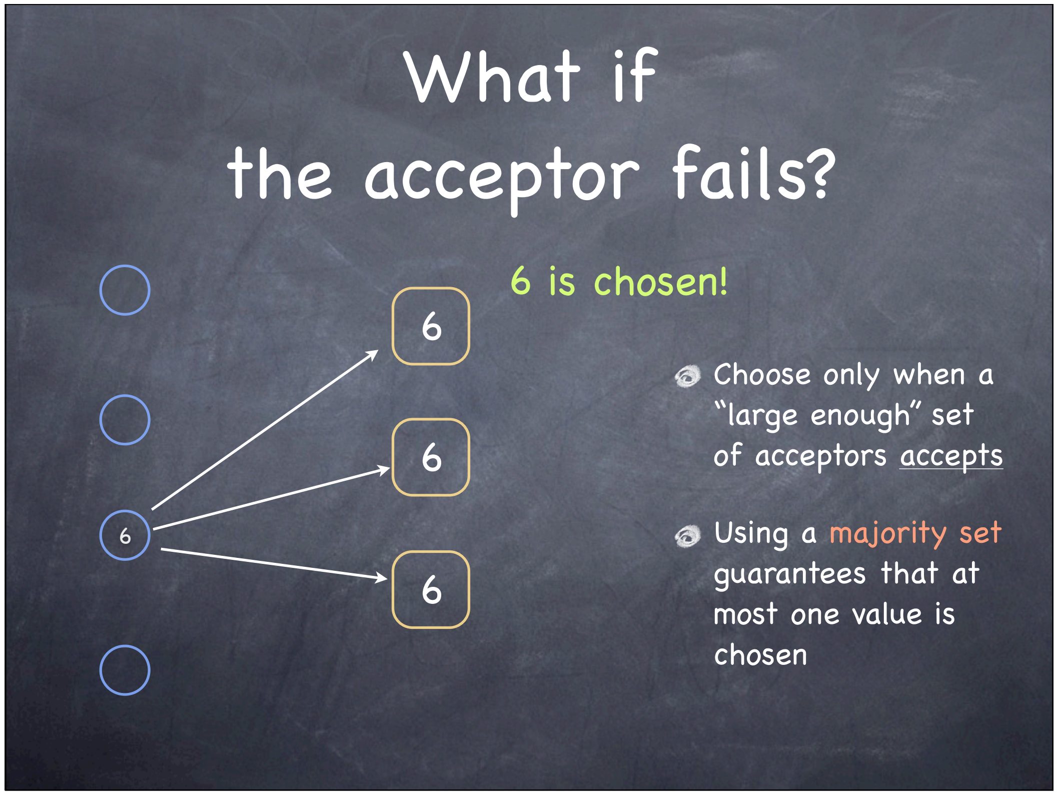
In a real implementation, they are all the same.



Choosing a value

Use a single acceptor

options? have a single acceptor. Problems with this? What if the acceptor fails. Then we don't know which value was chosen until the acceptor recovers.



What if the acceptor fails?

- Choose only when a "large enough" set of acceptors accepts
- Using a majority set guarantees that at most one value is chosen

Majority = commit. However, unlike a legislative "majority", none of the acceptors cares which value is chosen, just that one eventually is chosen.

Note that is unlike 2PC, where commit is only at the coordinator, commit now depends on distributed state.

Accepting a value

- Suppose only one value is proposed by a single proposer.
- That value should be chosen!
- First requirement:

  P1: An acceptor must accept the first proposal that it receives

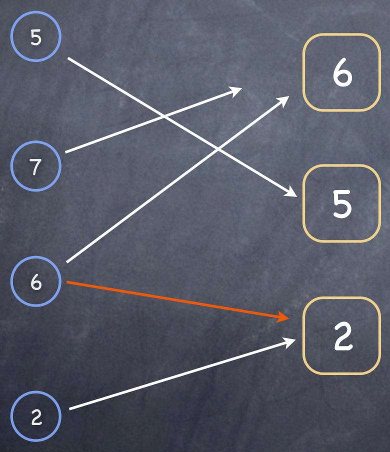
Accepting a value

- Suppose only one value is proposed by a single proposer.
- That value should be chosen!
- First requirement:

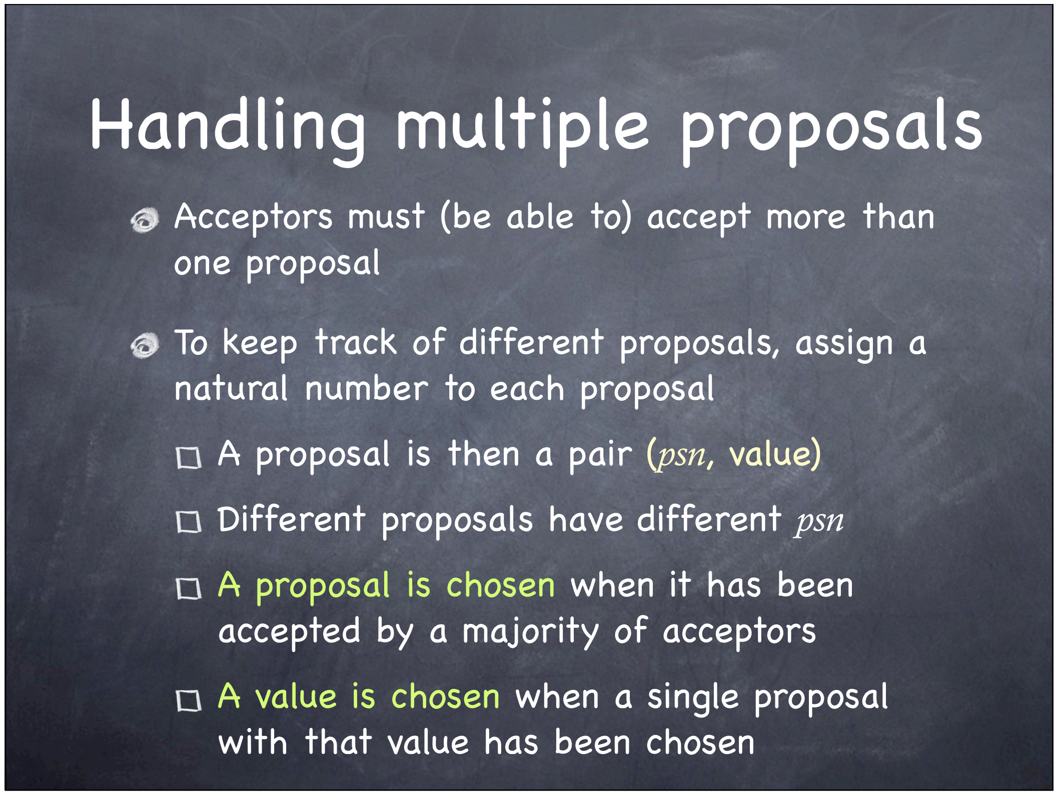
  P1: An acceptor must accept the first proposal that it receives

- ...but what if we have multiple proposers, each proposing a different value?

P1 + multiple proposers



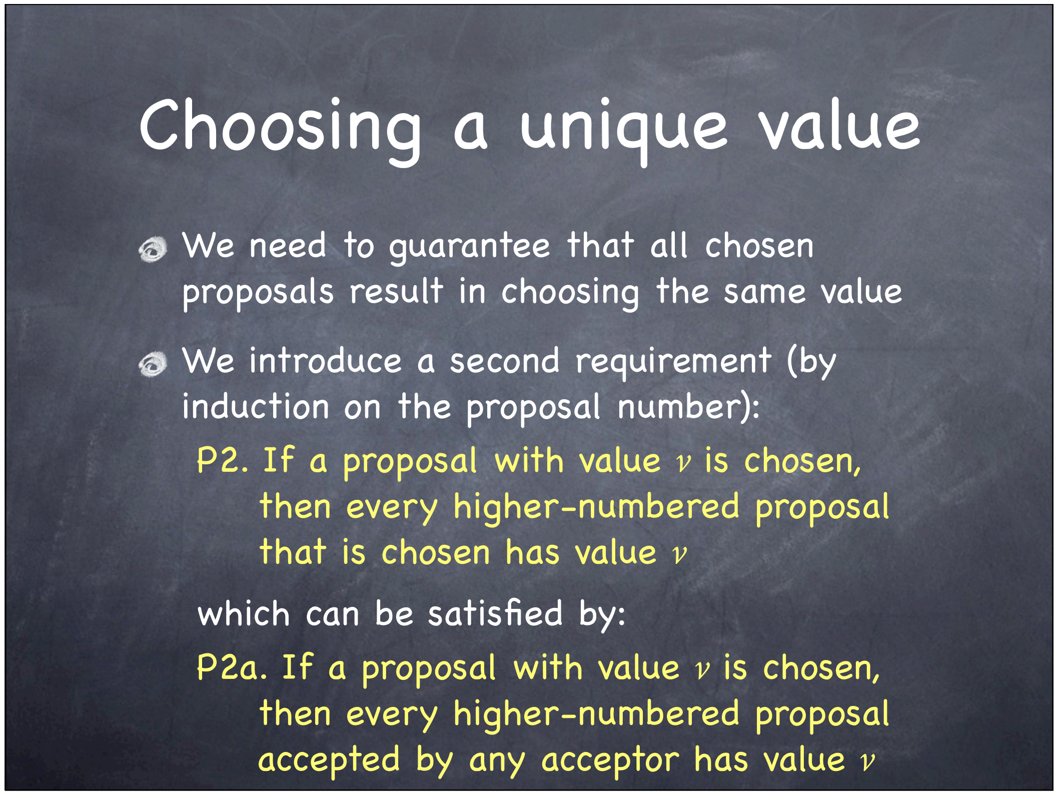
No value is chosen!



Handling multiple proposals

- Acceptors must (be able to) accept more than one proposal
- To keep track of different proposals, assign a natural number to each proposal
  - A proposal is then a pair (*psn*, value)
  - Different proposals have different *psn*
  - A proposal is chosen when it has been accepted by a majority of acceptors
  - A value is chosen when a single proposal with that value has been chosen

Recall: just a question of what is the next operation in the sequence. Clients suggest “do this operation next”, so there might be multiple competing possibilities. We want to choose exactly one as the next operation, realizing that way we’ll get around to doing the rest.



Choosing a unique value

- We need to guarantee that all chosen proposals result in choosing the same value
- We introduce a second requirement (by induction on the proposal number):

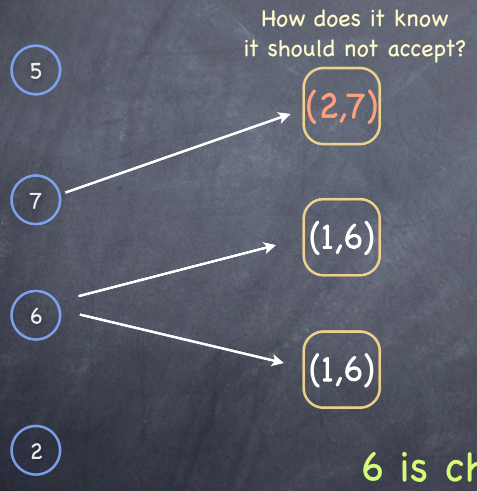
  P2. If a proposal with value v is chosen,
  then every higher-numbered proposal
  that is chosen has value v

  which can be satisfied by:

  P2a. If a proposal with value v is chosen,
  then every higher-numbered proposal
  accepted by any acceptor has value v

Key idea behind Paxos: ok if multiple proposals are chosen, as long as they all have the same value! We'll proceed by a series of steps, each one refining P2 with something more restrictive, that if true would imply the previous step. P2a implies P2: a proposal can't be chosen, unless it is accepted. So if we can make P2a hold, we're done.

What about P1?



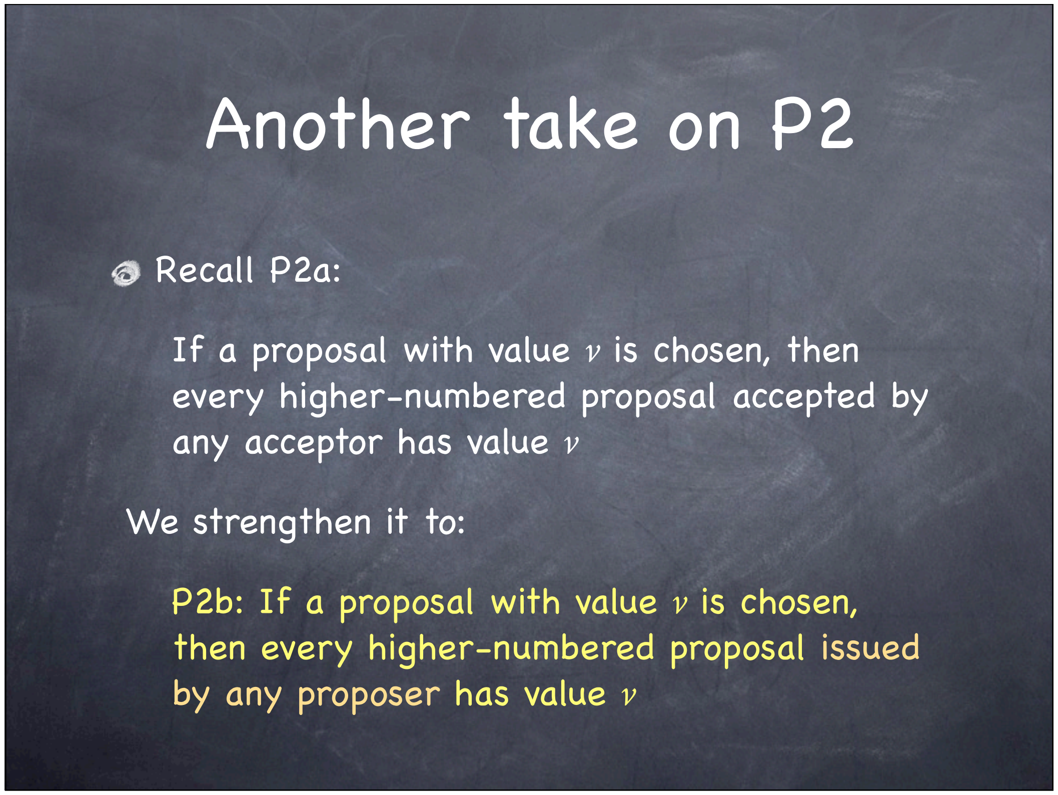
- Do we still need P1?

  YES, to ensure that *some* proposal is accepted

- How well do P1 and P2a play together?

  Asynchrony is a problem...

6 is chosen!



Another take on P2

- Recall P2a:

  If a proposal with value v is chosen, then every higher-numbered proposal accepted by any acceptor has value v

  We strengthen it to:

  P2b: If a proposal with value v is chosen, then every higher-numbered proposal issued by any proposer has value v

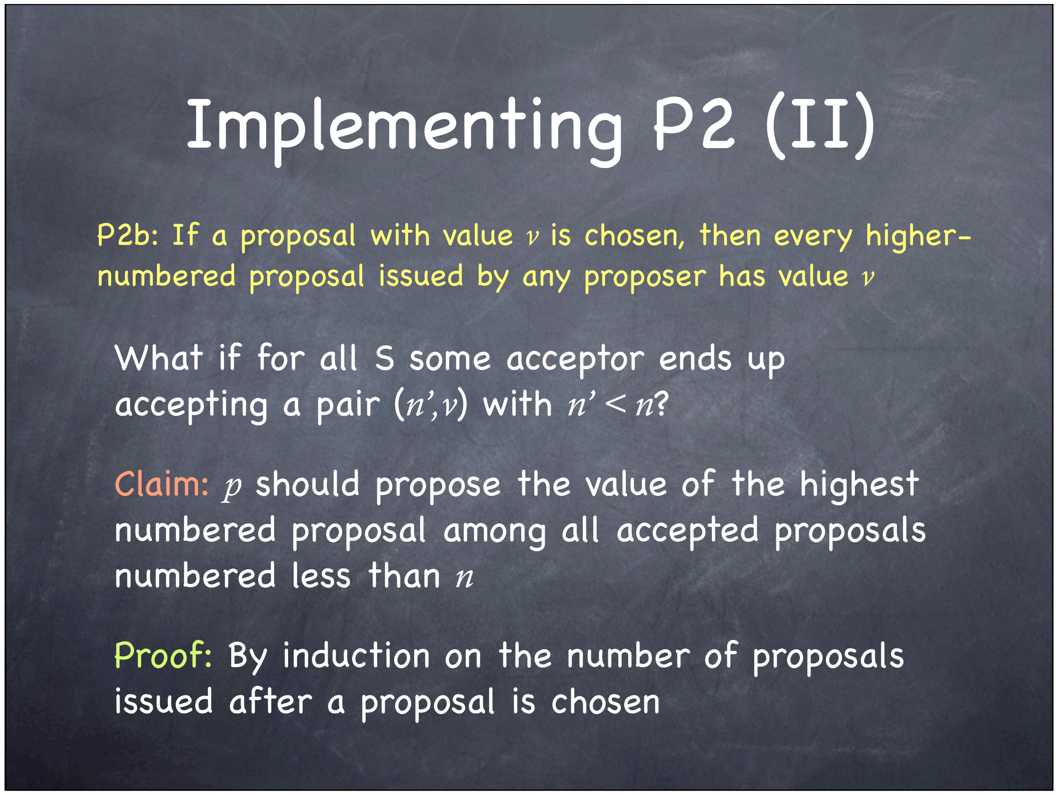
P2b is more restrictive than P2a: can’t accept a proposal, if it isn’t issued.

Implementing P2 (I)

P2b: If a proposal with value v is chosen, then every higher-numbered proposal issued by any proposer has value v

Suppose a proposer p wants to issue a proposal numbered n. What value should p propose?

- If (n',v) with $n' < n$ is chosen, then in every majority set S of acceptors at least one acceptor has accepted (n',v)...
- ...so, if there is a majority set S where no acceptor has accepted (or will accept) a proposal with number less than n, then p can propose any value



Implementing P2 (II)

P2b: If a proposal with value v is chosen, then every higher-numbered proposal issued by any proposer has value v

What if for all S some acceptor ends up accepting a pair (n',v) with $n' < n$?

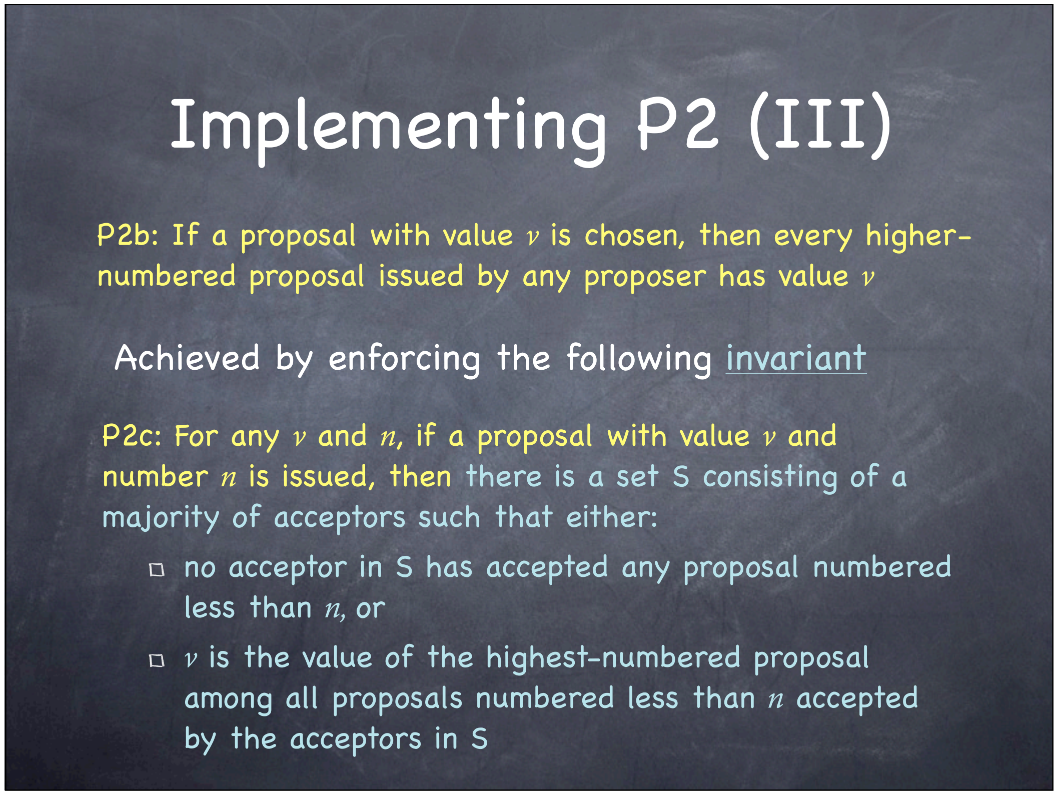
Claim: p should propose the value of the highest numbered proposal among all accepted proposals numbered less than n

Proof: By induction on the number of proposals issued after a proposal is chosen

Claim is an invariant, that if met, will satisfy P2b. Proof by contradiction, that is, claim is true, but P2b is false. Assume some higher numbered proposal is issued with a different value. Pick the smallest such proposal number. Someone in that set must have accepted the chosen value v, by the definition of majority. Since this is the first one after the chosen proposal with a different value, the highest numbered proposal with an accept must have been v. Contradiction.

Lorenzo’s notes:

Every acceptor in *C* has accepted a proposal with number in
m : : (*n* ¡ 1), and every proposal with number in *m : :* (*n* ¡ 1)
accepted by any acceptor has value *v*.
Since any set *S* consisting of a majority of acceptors contains at least one
member of *C*, we can conclude that a proposal numbered *n* has value *v* by
ensuring that the following invariant is maintained:



Implementing P2 (III)

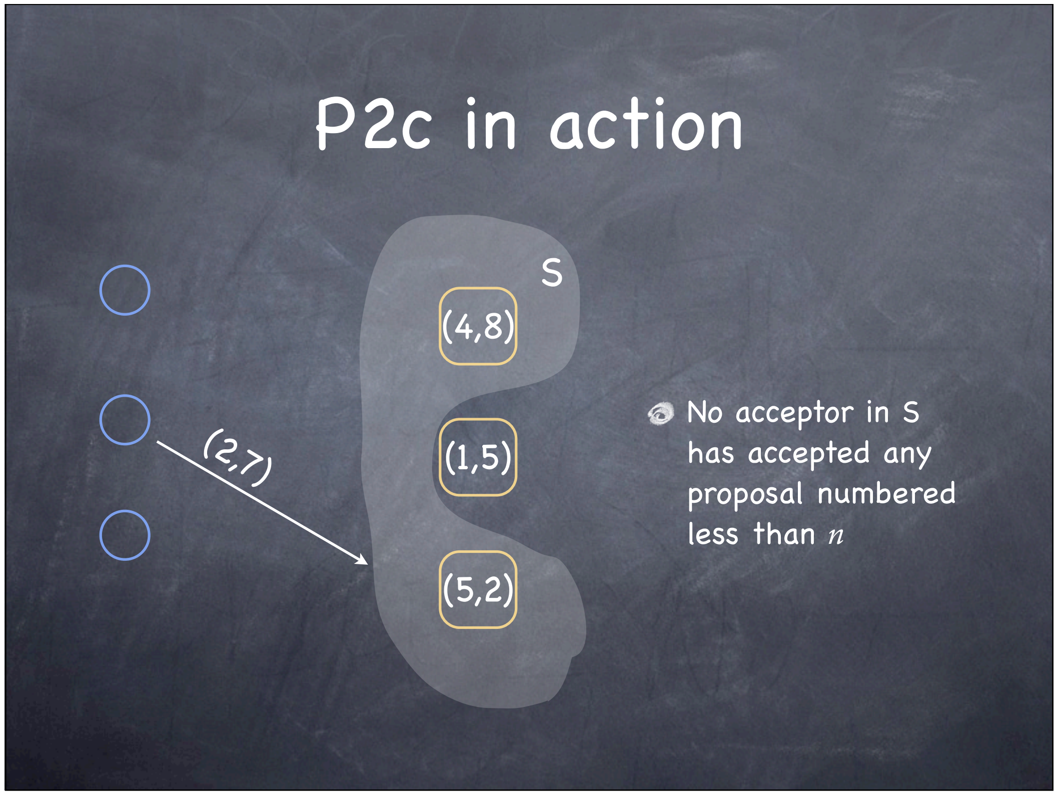
P2b: If a proposal with value v is chosen, then every higher-numbered proposal issued by any proposer has value v

Achieved by enforcing the following invariant

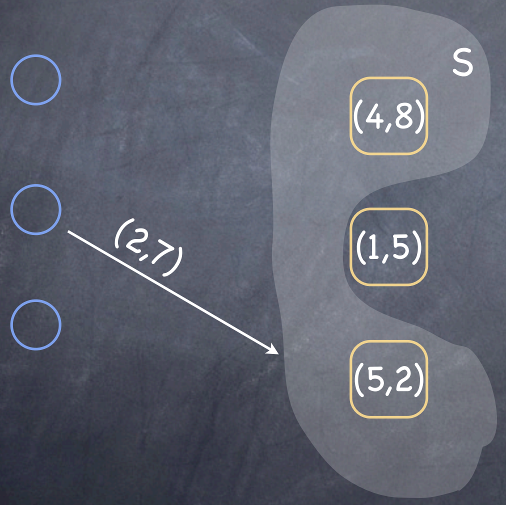
P2c: For any v and n, if a proposal with value v and number n is issued, then there is a set S consisting of a majority of acceptors such that either:

- no acceptor in S has accepted any proposal numbered less than n, or
- v is the value of the highest-numbered proposal among all proposals numbered less than n accepted by the acceptors in S

Again, more restrictive. Any majority will overlap with at least one member with the majority that chose v.



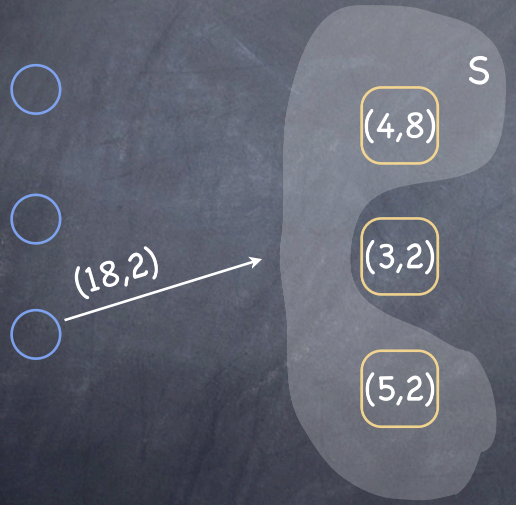
P2c in action



- No acceptor in S has accepted any proposal numbered less than n

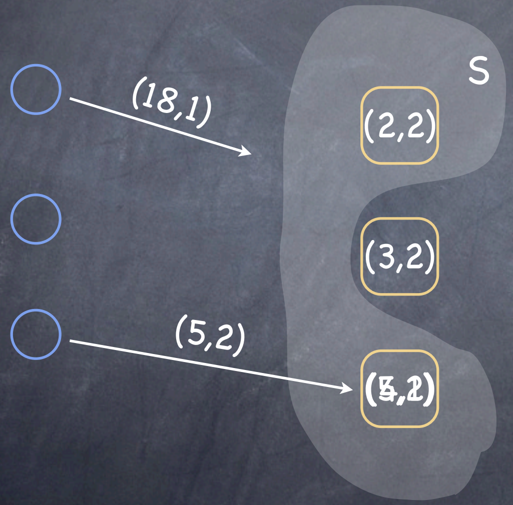
notation: (x,y) is proposal #x, value #y. In this case, ok to propose any value with proposal #2. Of course, this can't happen! Won't propose a value that's smaller than the proposals already accepted, because the two nodes will simply ignore it. But might have a late message, and invariant still needs to hold regardless of how late the delivery is.

P2c in action



- v is the value of the highest-numbered proposal among all proposals numbered less than n and accepted by the acceptors in S

P2c in action



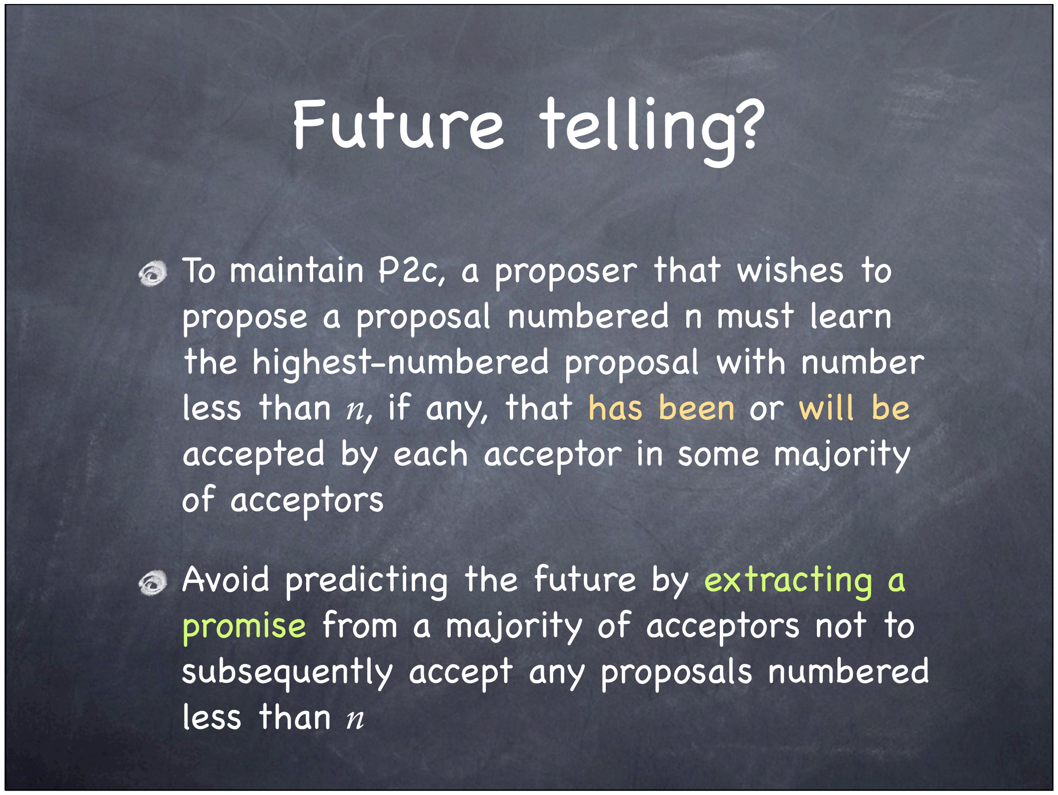
- v is the value of the highest-numbered proposal among all proposals numbered less than n and accepted by the acceptors in S

The invariant is violated

(to my thinking, it would be clearer if (5,2) was to a different set S’, consisting of the top two. Then the invariant is still violated, even though 5,2 was issued “after” (in a temporal sense) 18,1.

Future telling?

- To maintain P2c, a proposer that wishes to propose a proposal numbered n must learn the highest-numbered proposal with number less than n, if any, that has been or will be accepted by each acceptor in some majority of acceptors



Future telling?

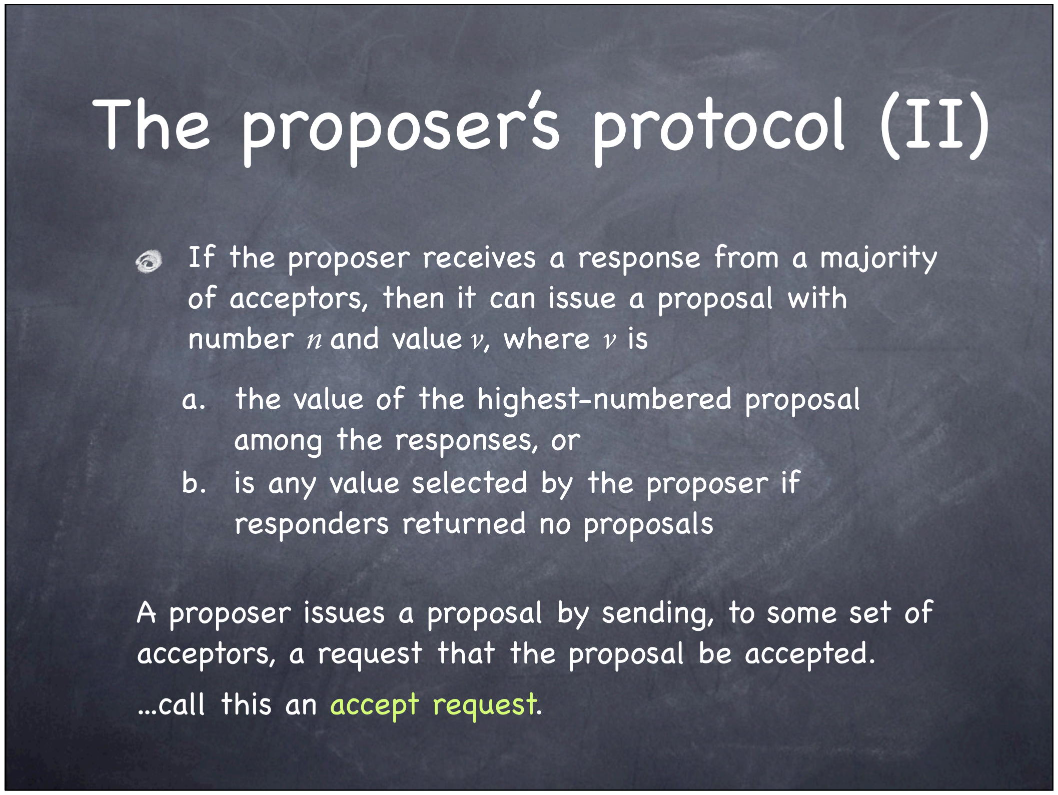
- To maintain P2c, a proposer that wishes to propose a proposal numbered n must learn the highest-numbered proposal with number less than n, if any, that has been or will be accepted by each acceptor in some majority of acceptors
- Avoid predicting the future by extracting a promise from a majority of acceptors not to subsequently accept any proposals numbered less than n

Doesn’t cause blocking since we can always issue a higher #’ed proposal

The proposer's protocol (I)

- A proposer chooses a new proposal number n and sends a request to each member of some (majority) set of acceptors, asking it to respond with:
    a. A promise never again to accept a proposal numbered less than n, and
    b. The accepted proposal with highest number less than n if any.

...call this a prepare request with number n



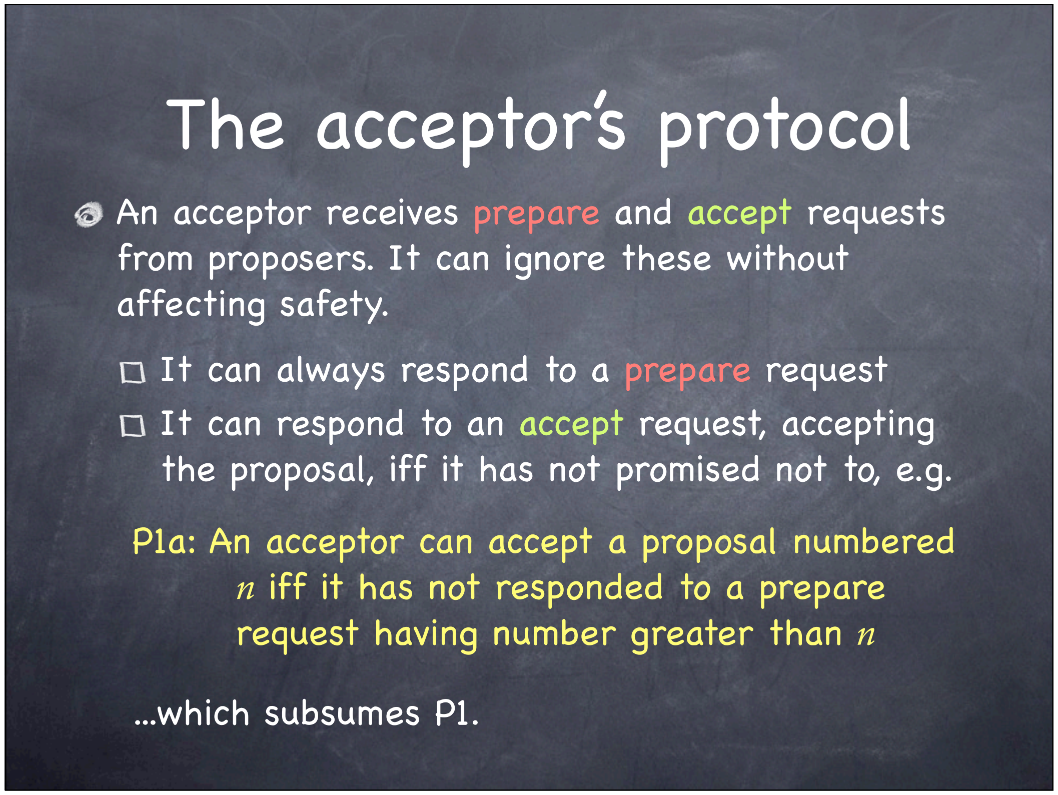
The proposer's protocol (II)

- If the proposer receives a response from a majority of acceptors, then it can issue a proposal with number n and value v, where v is
  a. the value of the highest-numbered proposal among the responses, or
  b. is any value selected by the proposer if responders returned no proposals

A proposer issues a proposal by sending, to some set of acceptors, a request that the proposal be accepted.

...call this an accept request.

Recall we don't care which value is chosen, just that some value is chosen, and we all agree on which one was chosen.



The acceptor's protocol

- An acceptor receives prepare and accept requests from proposers. It can ignore these without affecting safety.
  - It can always respond to a prepare request
  - It can respond to an accept request, accepting the proposal, iff it has not promised not to, e.g.

P1a: An acceptor can accept a proposal numbered n iff it has not responded to a prepare request having number greater than n

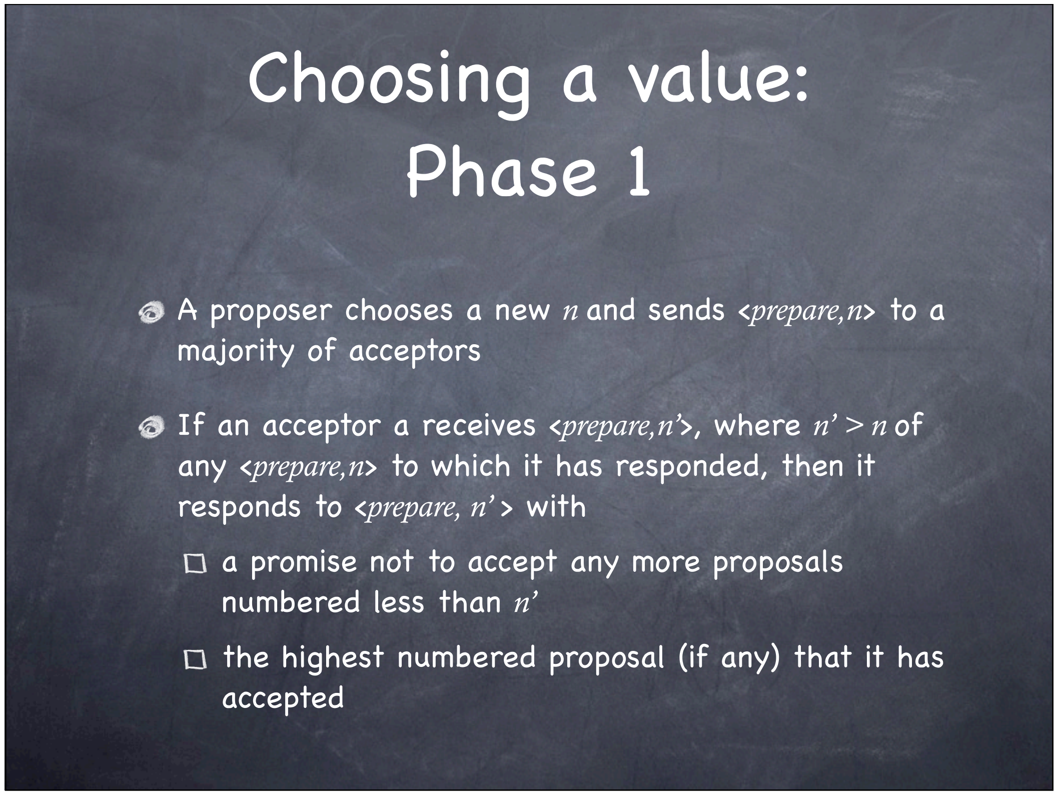
...which subsumes P1.

Small optimizations

- If an acceptor receives a prepare request r numbered n when it has already responded to a prepare request for $n' > n$, then the acceptor can simply ignore r.
- An acceptor can also ignore prepare requests for proposals it has already accepted

...so an acceptor needs only remember the highest numbered proposal it has accepted and the number of the highest-numbered prepare request to which it has responded.

This information needs to be stored on stable storage to allow restarts.



Choosing a value: Phase 1

- A proposer chooses a new n and sends <*prepare,n*> to a majority of acceptors
- If an acceptor a receives <*prepare,n'*>, where $n' > n$ of any <*prepare,n*> to which it has responded, then it responds to <*prepare, n'*> with
  - a promise not to accept any more proposals numbered less than n'
  - the highest numbered proposal (if any) that it has accepted

to recap

Choosing a value: Phase 2

- If the proposer receives a response to <*prepare,n*> from a majority of acceptors, then it sends to each <*accept,n,v*>, where *v* is either
  - the value of the highest numbered proposal among the responses
  - any value if the responses reported no proposals
- If an acceptor receives <*accept,n,v*>, it accepts the proposal unless it has in the meantime responded to <*prepare,n'*> , where $n' > n$

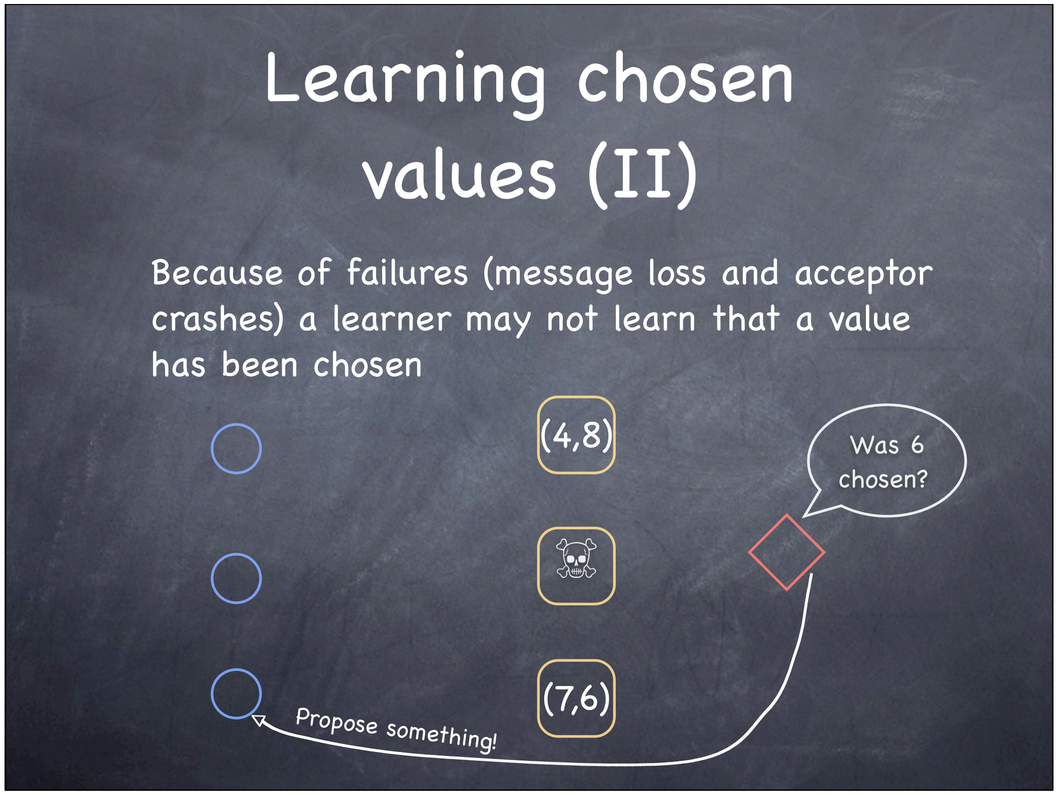
Learning chosen values (I)

Once a value is chosen, learners should find out about it. Many strategies are possible:

i. Each acceptor informs each learner whenever it accepts a proposal.

ii. Acceptors inform a distinguished learner, who informs the other learners

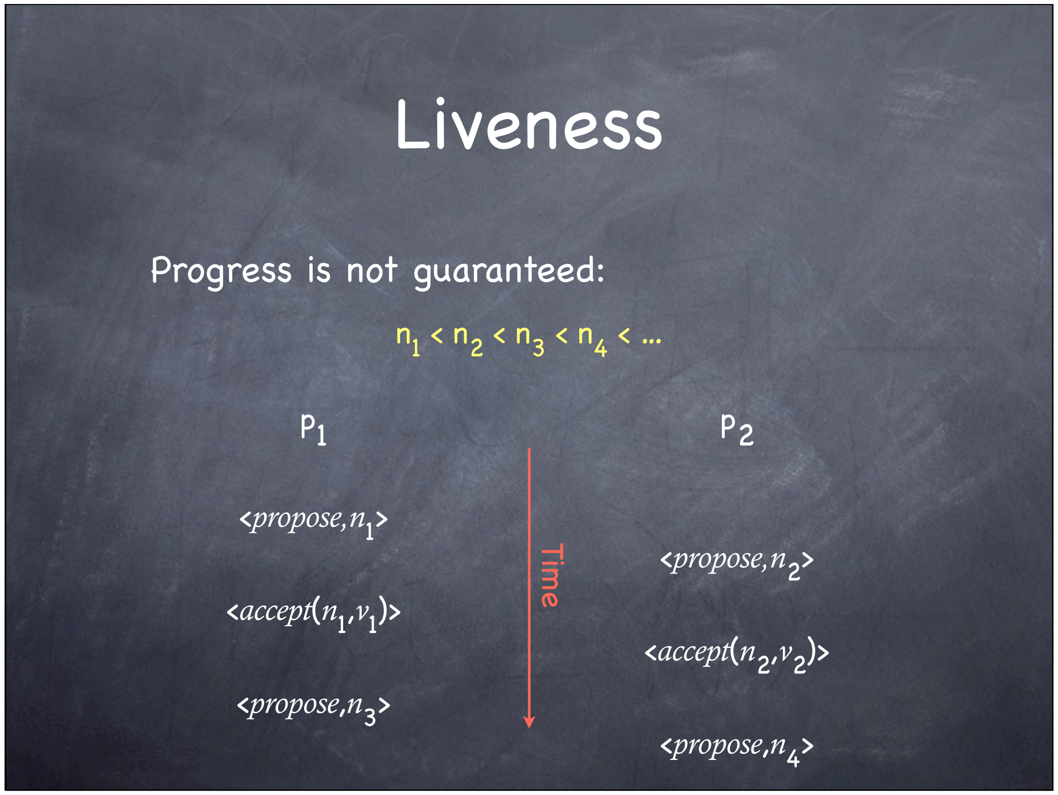
iii. Something in between (a set of not-quite-as-distinguished learners)



Learning chosen values (II)

Because of failures (message loss and acceptor crashes) a learner may not learn that a value has been chosen

Can’t tell if anything was chosen, but still want to be able to make progress if a majority is up. So propose a value that by construction must be consistent with what was chosen, if anything was chosen. And if nothing had been chosen, still works. Very similar in concept to GFS: don’t know if append succeeded, so just keep redoing it until it does. Note that paxos predates GFS (!)



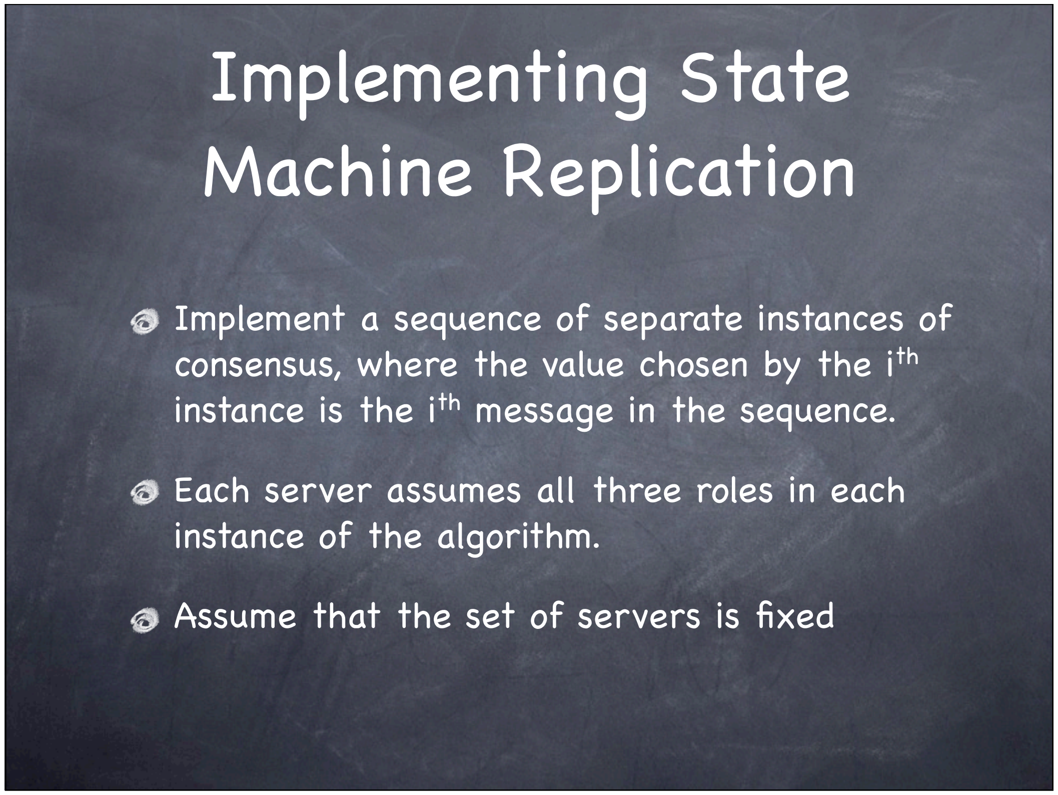
Liveness

Progress is not guaranteed:

$n_1 < n_2 < n_3 < n_4 < \ldots$

P_1		P_2
<*propose*,n_1>	Time	
		<*propose*,n_2>
<*accept*(n_1,v_1)>		
		<*accept*(n_2,v_2)>
<*propose*,n_3>		
		<*propose*,n_4>

Key to progress is having only one proposer at a time. How do we do that?



Implementing State Machine Replication

- Implement a sequence of separate instances of consensus, where the value chosen by the i^{th} instance is the i^{th} message in the sequence.
- Each server assumes all three roles in each instance of the algorithm.
- Assume that the set of servers is fixed

And recall, we want a sequence of operations, not just a single value. So how do we do this?

Another question: how do we relax assumption that set of servers is fixed?

The role of the leader

- In normal operation, elect a single server to be a leader. The leader acts as the distinguished proposer in all instances of the consensus algorithm.
  - Clients send commands to the leader, which decides where in the sequence each command should appear.
  - If the leader, for example, decides that a client command is the k^{th} command, it tries to have the command chosen as the value in the k^{th} instance of consensus.

A new leader λ is elected...

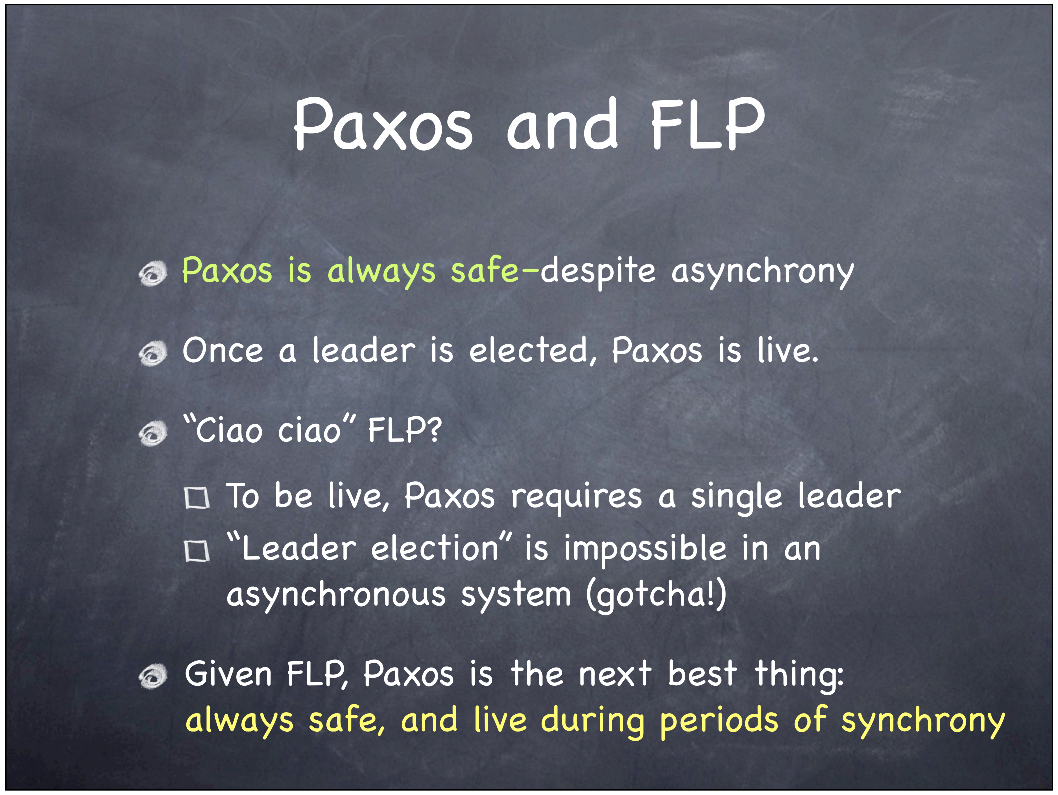
- Since λ is a learner in all instances of consensus, it should know most of the commands that have already been chosen. For example, it might know commands 1-10, 13, and 15.
    - It executes phase 1 of instances 11, 12, and 14 and of all instances 16 and larger.
    - This might leave, say, 14 and 16 constrained and 11, 12 and all commands after 16 unconstrained.
    - λ then executes phase 2 of 14 and 16, thereby choosing the commands numbered 14 and 16

Stop-gap measures

- All replicas can execute commands 1-10, but not 13-16 because 11 and 12 haven't yet been chosen.
- λ can either take the next two commands requested by clients to be commands 11 and 12, or can propose immediately that 11 and 12 be no-op commands.
- λ runs phase 2 of consensus for instance numbers 11 and 12.
- Once consensus is achieved, all replicas can execute all commands through 16.

To infinity, and beyond

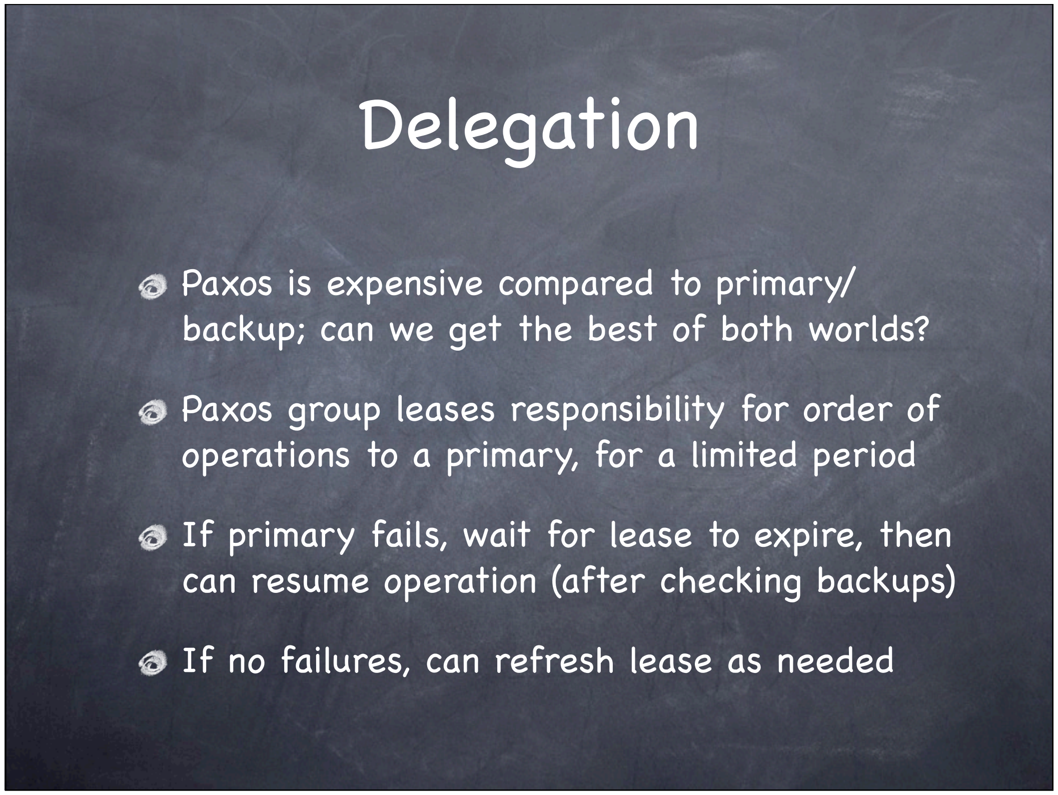
- λ can efficiently execute phase 1 for infinitely many instances of consensus! (e.g. command 16 and higher)
    - λ just sends a message with a sufficiently high proposal number for all instances
    - An acceptor replies non trivially only for instances for which it has already accepted a value



Paxos and FLP

- Paxos is always safe–despite asynchrony
- Once a leader is elected, Paxos is live.
- "Ciao ciao" FLP?
  - To be live, Paxos requires a single leader
  - "Leader election" is impossible in an asynchronous system (gotcha!)
- Given FLP, Paxos is the next best thing: always safe, and live during periods of synchrony

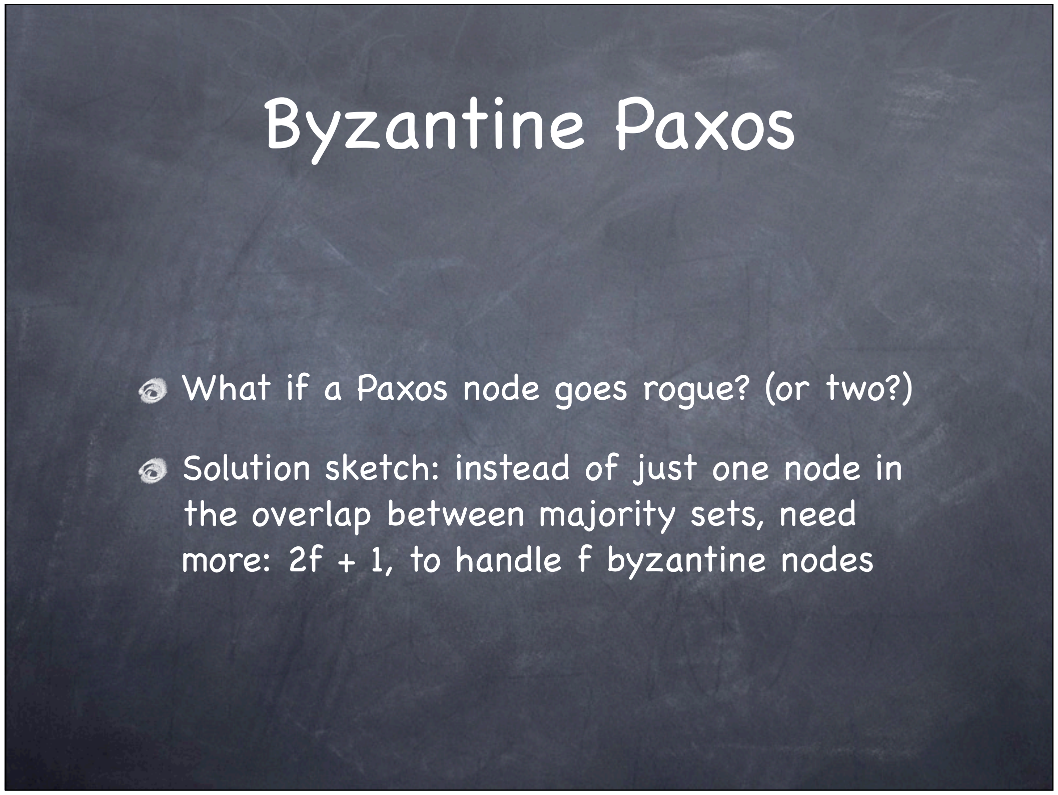
if paxos works, why not do this instead of GFS at least once semantics?



Delegation

- Paxos is expensive compared to primary/backup; can we get the best of both worlds?
- Paxos group leases responsibility for order of operations to a primary, for a limited period
- If primary fails, wait for lease to expire, then can resume operation (after checking backups)
- If no failures, can refresh lease as needed

if paxos is slow, how can we speed up the common case? Use paxos for selecting the primary, with a lease. If primary fails, wait for lease to expire, and vote on a new primary. Completely safe, yet allows operations to be sequenced at a central master in most cases.



Byzantine Paxos

- What if a Paxos node goes rogue? (or two?)
- Solution sketch: instead of just one node in the overlap between majority sets, need more: 2f + 1, to handle f byzantine nodes

What about Byzantine behavior? Clearly, Paxos is easy to corrupt -- if a proposer proposes a different value than what the acceptors returned; or if the acceptor says that a value was accepted when it wasn't, or vice versa.
How do we fix this? Instead of just one node in the overlap, need 2f+1, for f byzantine nodes. The extra f+1 outvote the f byzantine nodes, allowing you to make progress.