Choosing a Value: Pseudocode

Phase 1:

a. Proposer c:
   i. Selects a unique proposal number $n > crnd_c$, sets $cval_c$ to *none* and $crnd_c$ to n.
   ii. Sends a *prepare*(n) to all acceptors.

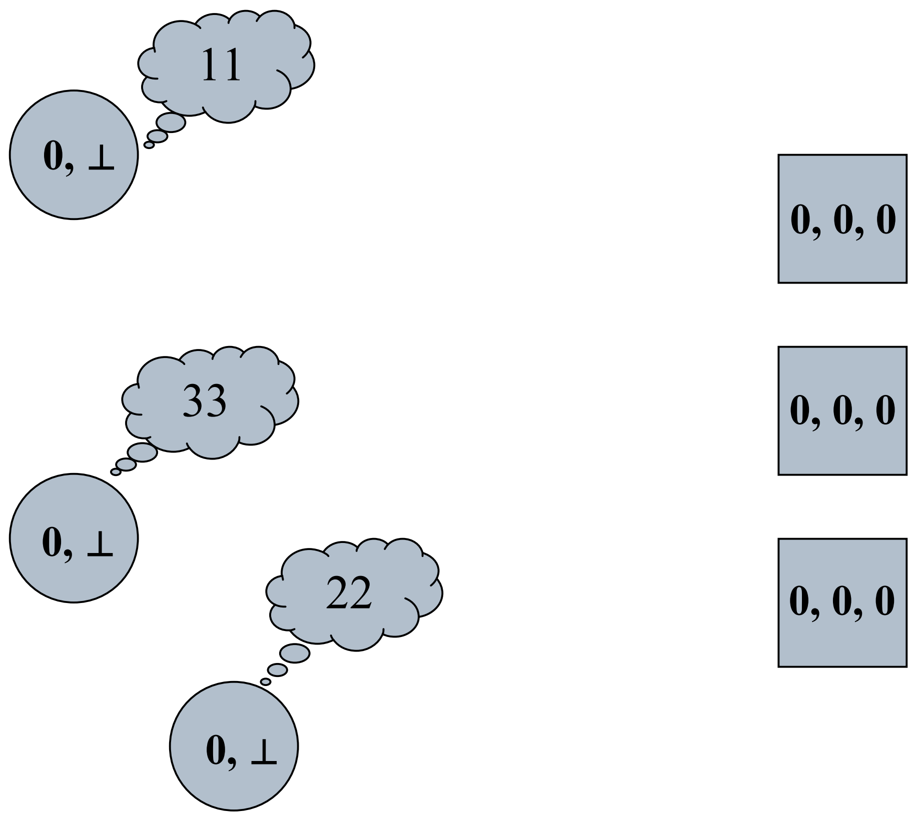
b. Acceptor a receives *prepare*(n) from c:
   i. If $n > rnd_a$ then set rnd_a to n and send *promise*(rnd_a, $vrnd_a$, $vval_a$) to c.
   ii. Else ignore request.

Phase 2:

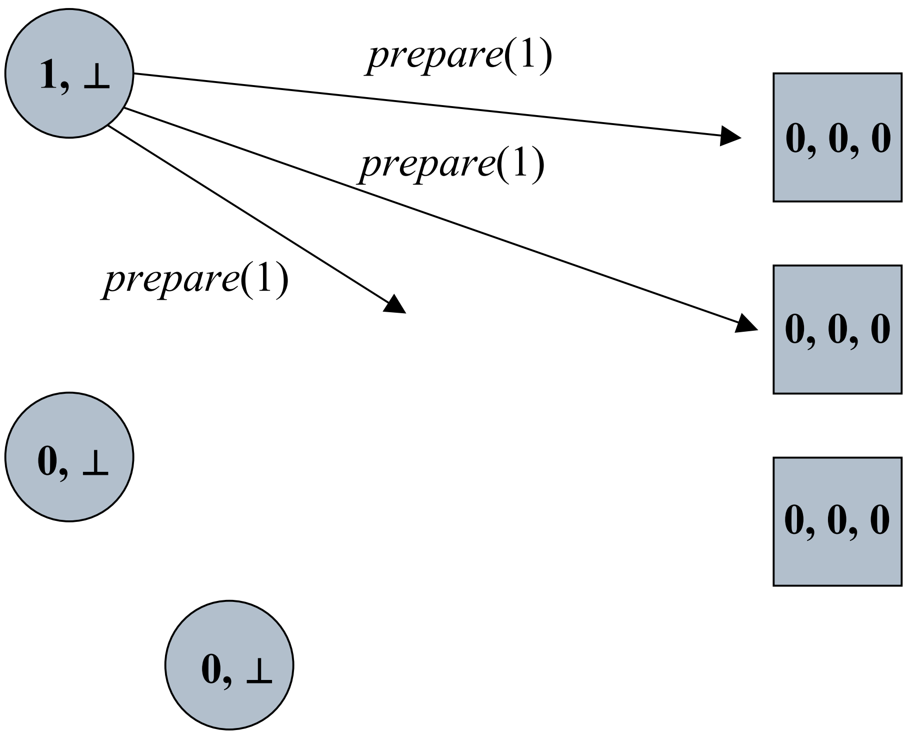
a. Proposer c receives *promise*(rnd_a, $vrnd_a$, $vval_a$) from a majority of acceptors with $rnd_a = crnd_c$:
   i. If all reply with $vrnd_a = 0$, then set $cval_c$ to any proposed value
      Else set $cval_c$ to $vval_a$ associated with largest received value of $vrnd_a$.
   ii. Send *accept*($crnd_c$, $cval_c$) to all acceptors.

b. Acceptor a receives *accept*(n, v):
   i. If $n \geq rnd_a$ and $vrnd_a \neq n$ then set $vrnd_a$ and rnd_a to n and $vval_a$ to v, and send *learn*(n, v).
   ii. Else ignore request.

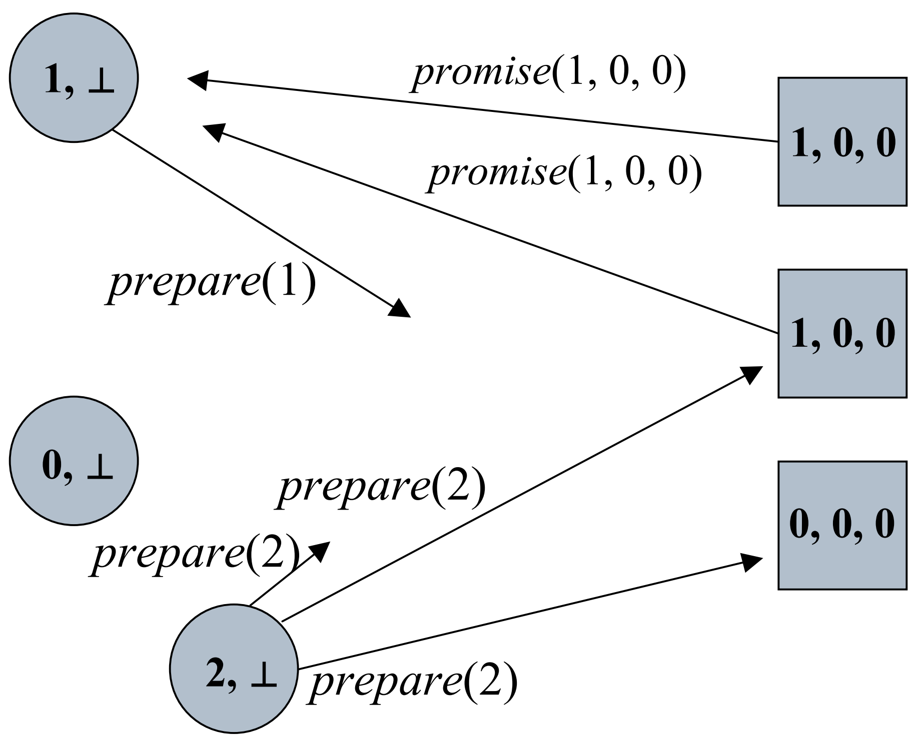
Paxos in Action: I



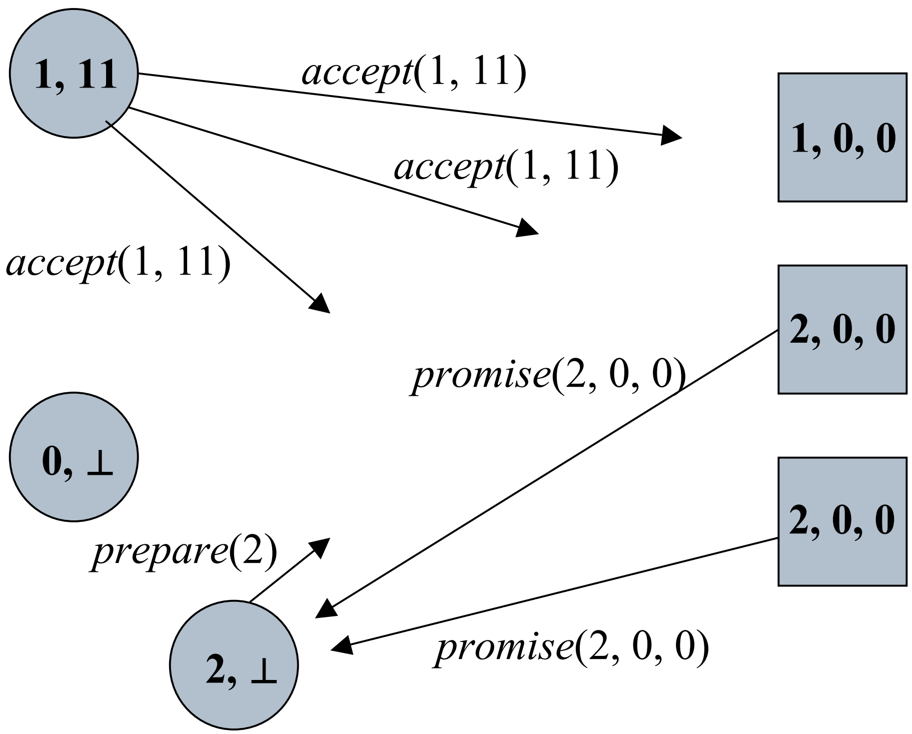
Paxos in Action: II



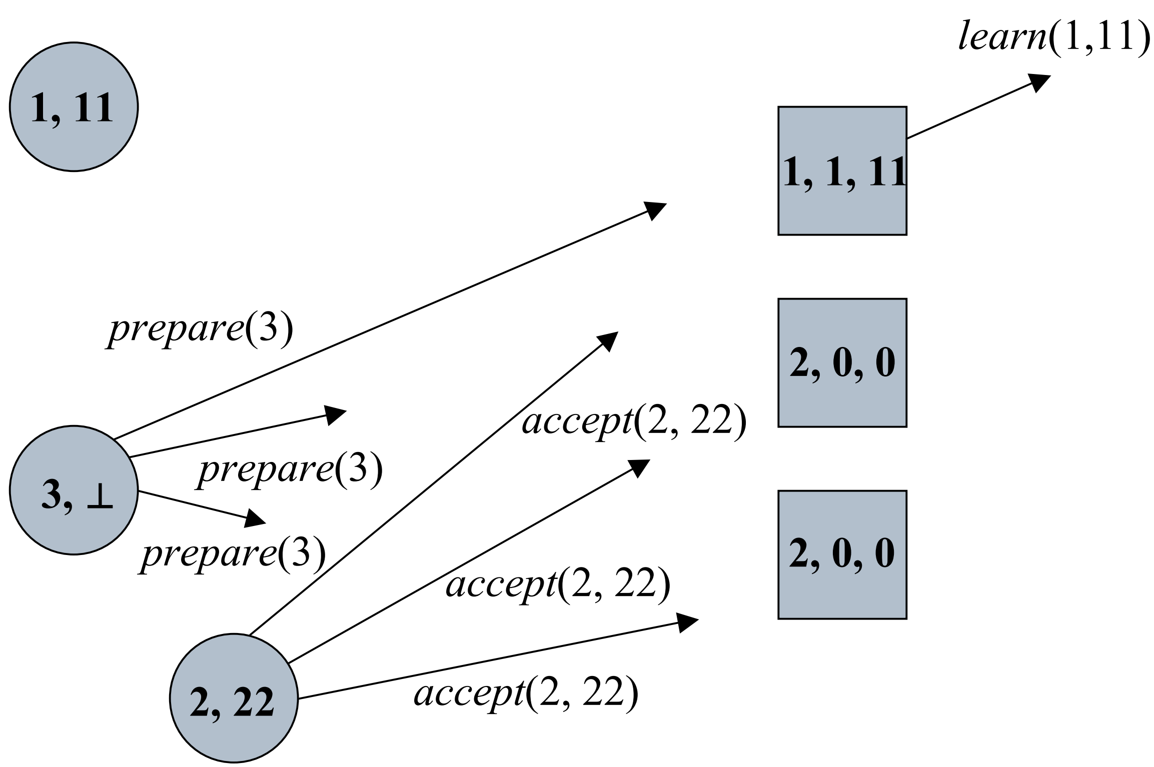
Paxos in Action: III



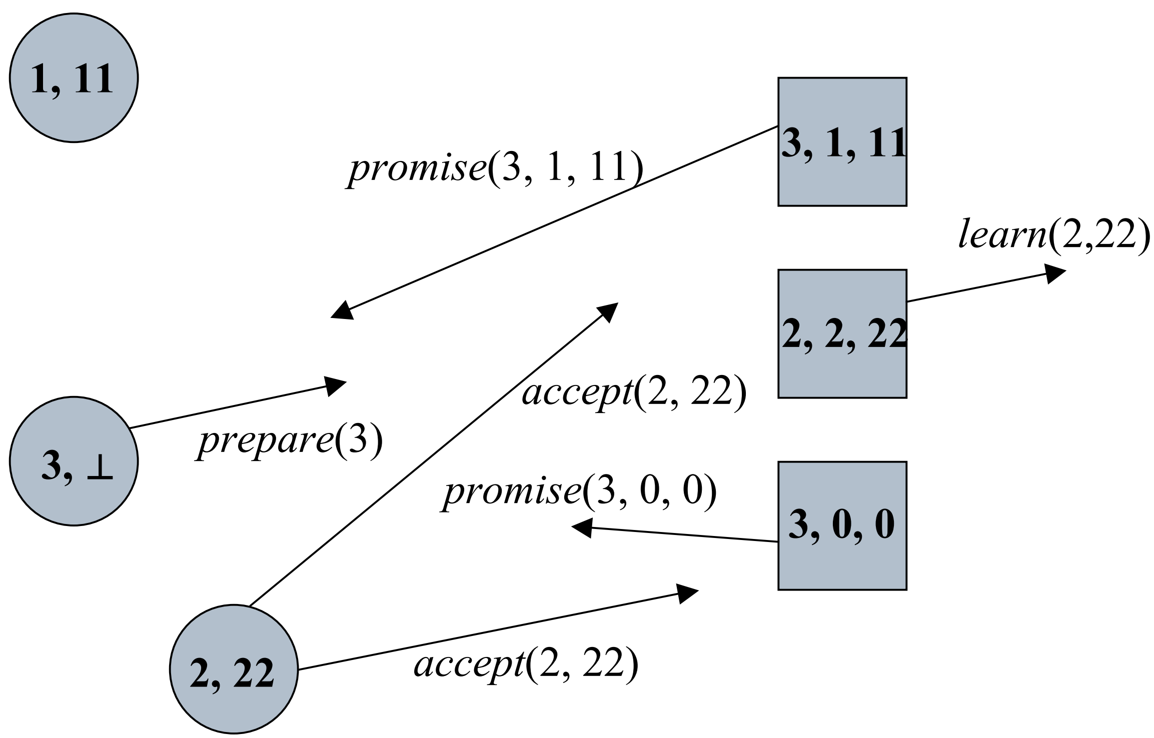
Paxos in Action: IV



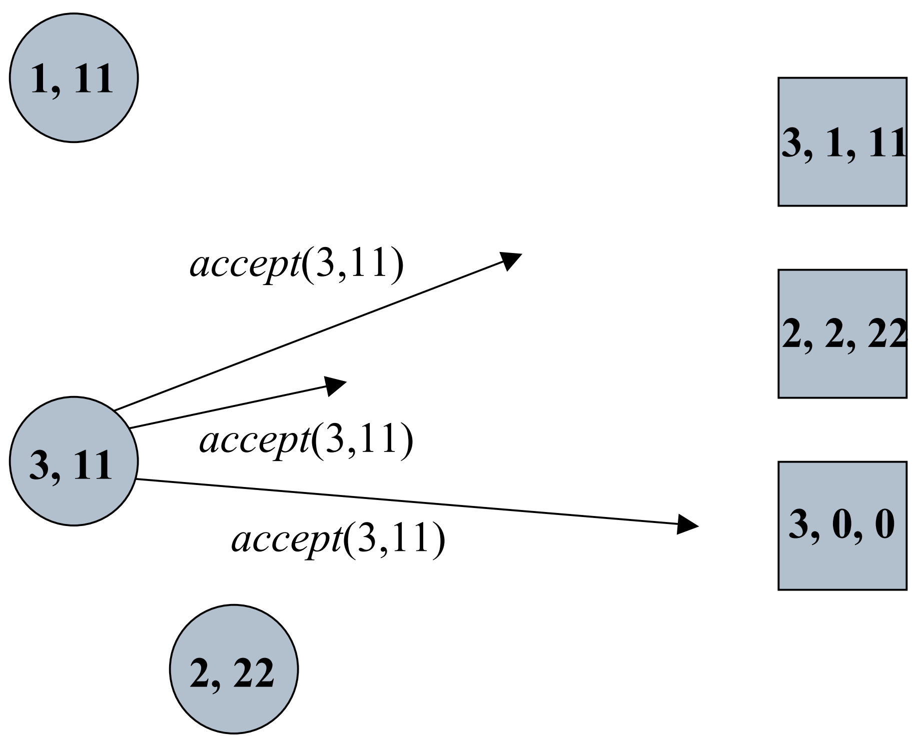
Paxos in Action: V



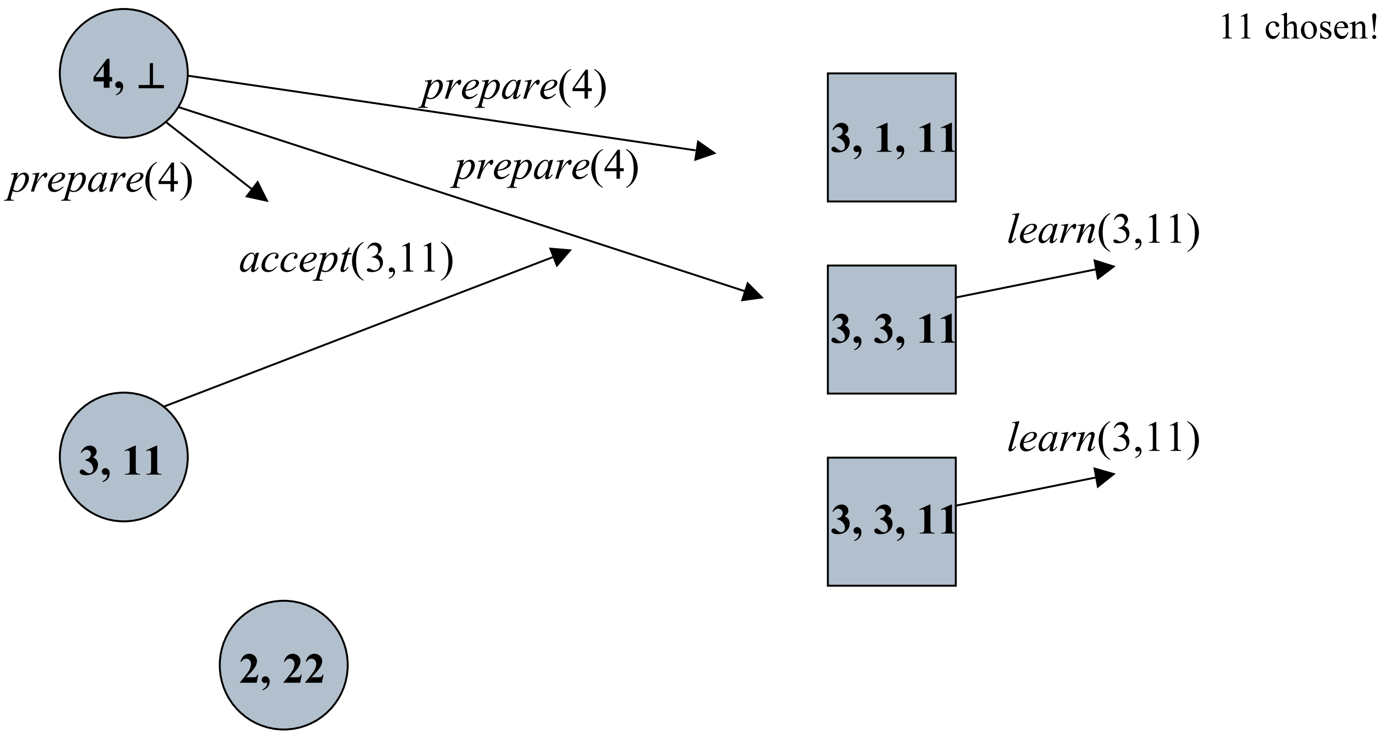
Paxos in Action: VI



Paxos in Action: VII



Paxos in Action: VIII

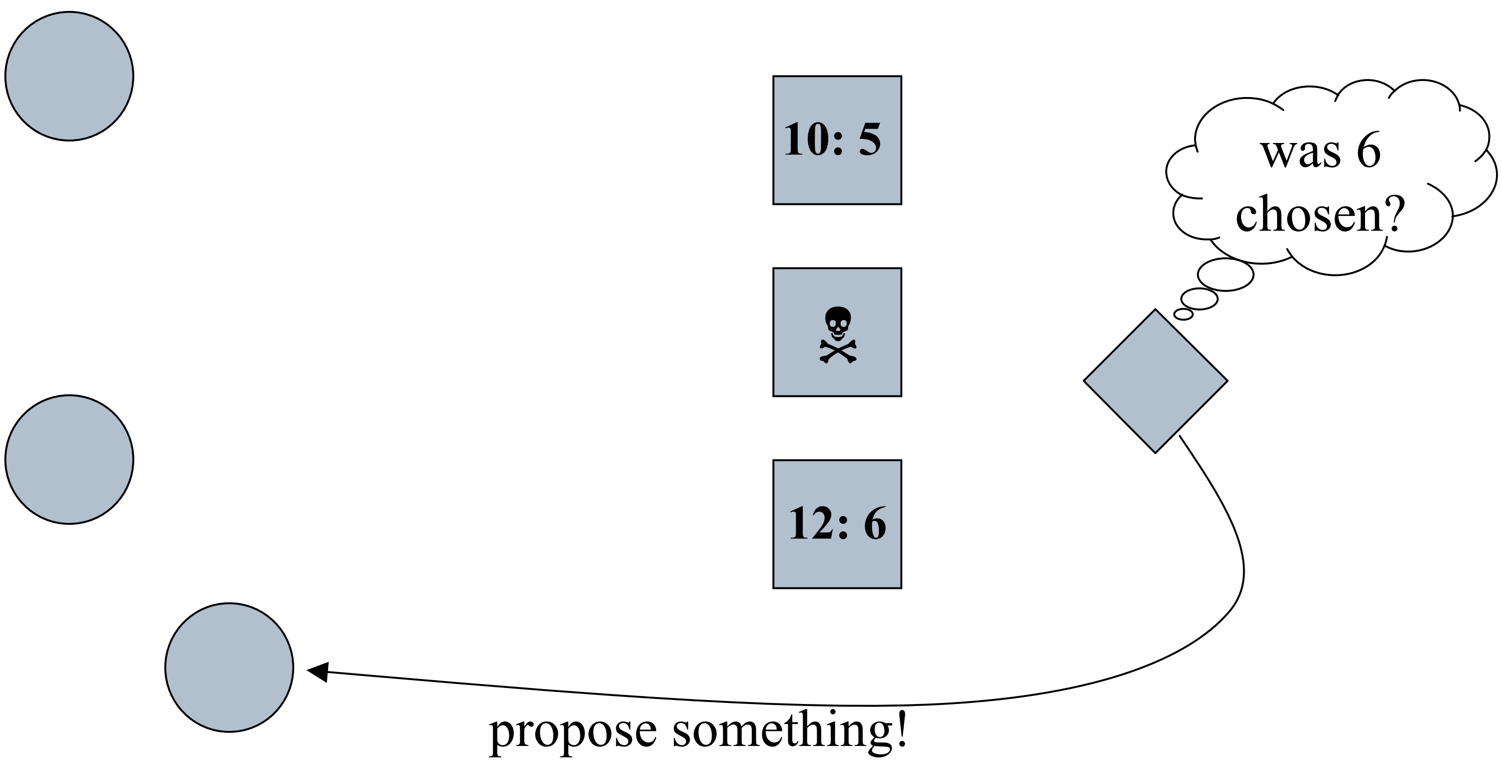


Learning a Chosen Value (I)

- A learner must find out that a proposal has been voted for by a majority of acceptors.
  - Can have each acceptor send a message to each learner whenever it accepts a proposal. When it receives the same message from a majority of acceptors, then it knows that the value in these messages was chosen.
  - Can have a *distinguished learner* (or set of such learners) that take on this role, and can inform other learners when a value has been chosen.

Learning a Chosen Value

- Due to message loss, a learner may not know that a value has been chosen.



Some light tuning

- Acceptor a receives phase 1a or 2a message from c for proposal number $n < rnd_a$ then a informs c that proposal number rnd_a has started.
- Coordinator c takes action only if it believes itself to be the current leader. It starts phase 1 only if $crnd_c = 0$ or learns that round $n > crnd_c$ has started.

$\diamond$S Consensus: 1/2

propose(v_p) {
    $estimate_p = v_p$;
    $state_p$ = **undecided**;
    $r_p = ts_p = 0$;
    while ($state_p$ == **undecided**) {
        $r_p = r_p + 1$;
        $c_p = r_{(p \textbf{ mod } n) + 1}$;
        send (p, r_p, $estimate_p$, ts_p) to c_p;   // *phase 1*
        if ($p == c_p$)   // *phase 2*
            receive (q, r_p, $estimate_q$ ts_q) into $msgs_p[r_p]$
                until have received from a majority;
            t = largest ts_q in $msgs_p[r_p]$;
            $estimate_p$ = one of the $estimate_q$ in $msgs_p[r_p]$ with $ts_q = t$;
            send (p, r_p, $estimate_p$) to all;

◇S Consensus: 2/2

```
    wait until suspect c_p or receive (c_p, r_cp, estimate_cp);     // phase 3
    if (received)
        estimate_p = estimate_cp;
        ts_p = r_p;
        send (p, r_p, ack) to c_p;
    else send (p, r_p, nack) to c_p;
    if (p == c_p)                                                   // phase 4
        wait until receive (q, r_p, ack/nack) from majority;
        if (all ack) R-broadcast (p, r_p, estimate_p, decide);
  }

when R-deliver (q, r_q, estimate_q, decide) {
  if (state_p == undecided) { decide(estimate_q); state_p = decided; }
  }
```

$\Diamond$S Consensus as Paxos

- All processes are acceptors.
- Each round has a *distinguished proposer* and a *distinguished listener* (r mod n) + 1;
- Unique proposal numbers from the round structure.
- The value that a proposer proposes when no value is chosen is not determined.
- The conditions under which the protocol terminates are clearly evident.

Asynchronous consensus…

◇W is the weakest failure detector that solves consensus.

It's equivalent to ◇S.

It's also equivalent to Ω:

- Each process *p*'s failure detector outputs $trust_p$: a single process *p* believes is correct.
- Ω ensures that eventually all correct processes always trust the same correct process.

Implementing State Machines (I)

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

Implementing State Machines (II)

- In normal operation, a single server is elected to be a *leader*, which acts as the distinguished proposer in all instances of the consensus algorithm.
  - Client 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.

Implementing State Machines (III)

Normal operation: a new leader λ is selected.

- 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 will execute phase 2 of 14 and 16, thereby choosing the commands numbered 14 and 16.

Implementing State Machines (IV)

- λ can execute -- or has already executed -- commands 1-10, but it can't execute 13-16 because 11 and 12 haven't yet been chosen.
- λ can take the next two commands requested by clients to be commands 11 and 12, but it could also immediately fill the gap by proposing them to be *null* commands that have no effect on the state machines. l proposes these commands by running phase 2 of consensus for instance numbers 11 and 12.
- Once consensus is obtained, λ can execute all commands through 16.

Implementing State Machines (V)

- How can we have λ execute phase 1 for an infinite instances of consensus (command 16 and higher)?
  - Since all instances are with the same servers, λ can send a message for all instances of consensus larger than some sequence number, and an acceptor can respond with a set of messages for which it has already accepted a value.
- The overhead of this approach, ignoring the transient overhead of starting up a new leader, is running phase 2 of the asynchronous consensus, which is optimal in terms of delay.

Implementing State Machines (VI)

- Based on *leader election* that in some situations may result in no leader or multiple leaders.
  - If there are no leaders, then no new commands will be proposed.
  - If there are multiple leaders, then they could propose values for the same instance of consensus, which may result in no value being chosen.

... in both cases, safety is preserved.

Implementing State Machines (VII)

- If the set of servers can change, then there needs to be some way to determine which set of servers implements which instance of consensus.
  - The most straightforward way to do this is via the state machine itself: have the set of servers be part of the state.
  - One can then chose a parameter α of the number of commands a leader can get ahead, and allow the state for instance $i+\alpha$ be specified after execution of the i^{th} command.

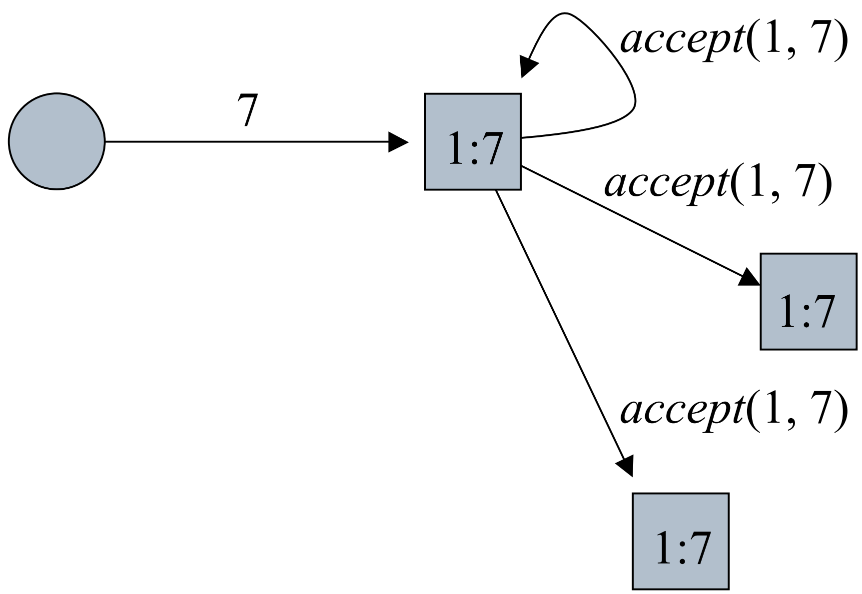
Making Paxos Faster

- The normal-case Paxos communication pattern is proposer → leader → acceptors → learners
  - In the common case for Paxos, the leader is unconstrained in the value it chooses for *accept*(*n, v*).
  - So, why not have the leader say *take any value* and have acceptors get values directly from a proposer?

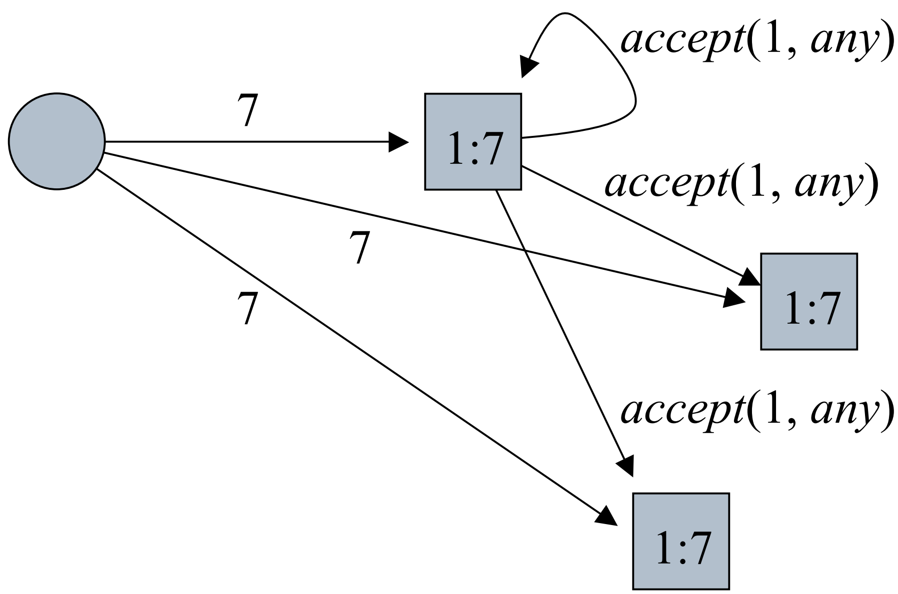
Fast Paxos

- Will call proposal numbers *rounds*.
- Create two kinds of rounds: *fast* and *classic*.
  - In a fast round, if the coordinator can pick any proposed value for 2a, it can instead send *propose*(*i*, ***any***).
  - When an acceptor receives *propose*(*i*, ***any***) it can treat any proposer's message proposing a value as if it were an ordinary round *i* phase 2a message with that value.
    - It can, however, execute a round *i* phase 2*b* action only once, for a single value.

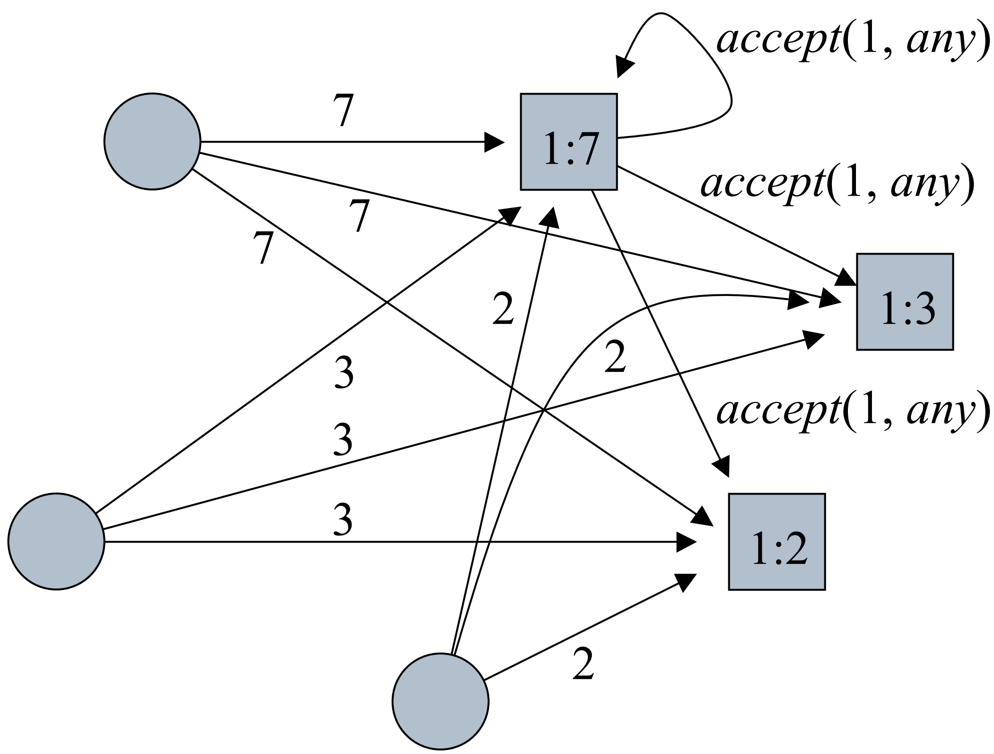
Classic Paxos



Fast Paxos



Fast Paxos



Problems with Fast Paxos

- How does a coordinator pick a value for 2a?
  - An issue if not in round 1...
- What happens if no value is chosen by 2b?
  - How can we tell this happened?

Picking value for 2a, round i?

- **let** Q be a quorum of acceptors that have sent phase 1b messages *promise*(vr_a, vv_a).
- **let** k be the largest vr_a for all $a \in Q$.
- **let** V be the set of values vv_a for all $a \in Q$ with $vr_a = k$.
- **if** $k = 0$ **then** pick any proposed value **else** pick the (only) element of V.

… this is no longer a sound rule because V can have more than one element!

Picking value for 2a round i

- The coordinator needs to pick a value v: for any round $j < i$, no value other than v has been or might yet be chosen in round j.
  - Recall k be the largest vr_a for all $a \in Q$.
    - if $k = 0$:
      - each $a \in Q$ sent $vr_a = 0$: none had voted for any $j < i$.
      - all $a \in Q$: $rnd_a \geq i$, so no value has been or ever might be chosen for any round $j < i$.
    - if $k > 0$:
      - three cases: $k < j$, $j = k$, and $j < k$.

Picking value for 2a round i

- if $k < j$: for each $a \in Q$
  - by time sent *promise*(vr_a, vv_a) had not voted on j.
  - since it promised, a won't vote on j after sending.
- if $k = j$:
  - acceptor only votes for value sent it by coordinator of round k, so all $a \in Q$ either voted vv_a or didn't vote.
- if $k > j$:
  - by induction, held through round k. So, no value other than v has been or might yet be chosen in round j.

Picking value for 2a round i

- if $k < j$: for each $a \in Q$
  - by time sent *promise*(vr_a, vv_a) had not voted on j.
  - since it promised, a didn't vote on j after sending.
- if $k = j$:
  - acceptor only votes for value sent it by coordinator of round k, so all $a \in Q$ either voted vv_a or didn't vote.
- if $k > j$:
  - by inducti d k. So, no value other than v has chosen in round j.

For a fast round, acceptor can vote for any proposed value!

Picking value for 2a round *i*

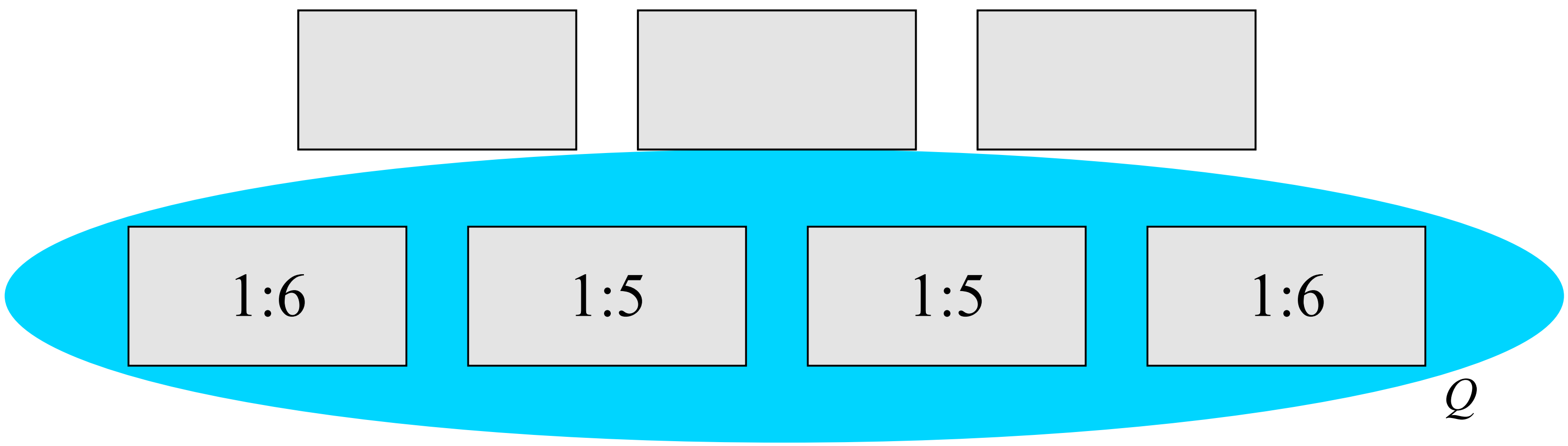
- Will be useful to allow different (sized) quorums for different rounds.
  - Quorum for round *i* is an *i-quorum*.
  - For any (not necessarily different) rounds *i* and *j*, any *i*-quorum and *j*-quorum intersect.

Picking value for 2a round i

- A value v might have been or might yet be chosen in round k only if there is a k-quorum R: for each acceptor a in R, a has $rnd_a \leq k$ or has voted for v in round k.
- Every $a \in Q$ has $rnd_a \geq i > k$ since it sent its promise.
- So, v might have been or might yet be chosen in round k only if there is a k-quorum R: for all $a \in Q \cap R$: $vr_a = k$ and $vv_a = v$.
- Call this *assumption* $A(v)$.
  - No value v: $A(v)$? Can pick any proposed value!
  - Only one value v: $A(v)$? Pick v!
  - Two values v, w: $A(v)$ and $A(w)$? Tough luck!

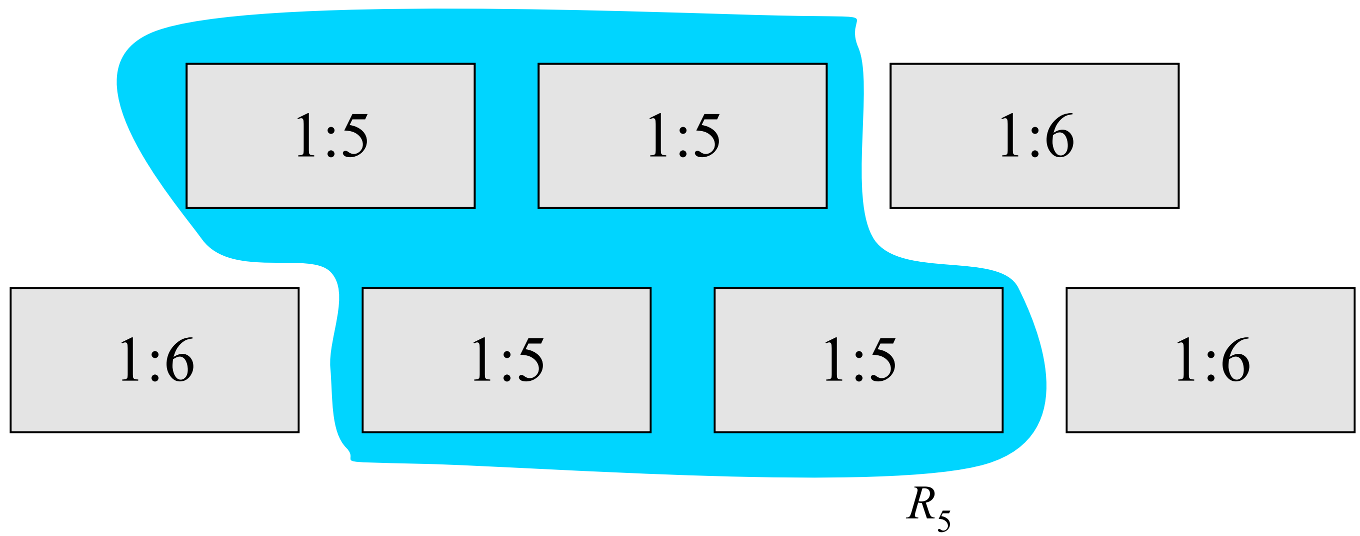
Condition $A(v)$

$N = 7,\ t = 3$, majority = 4



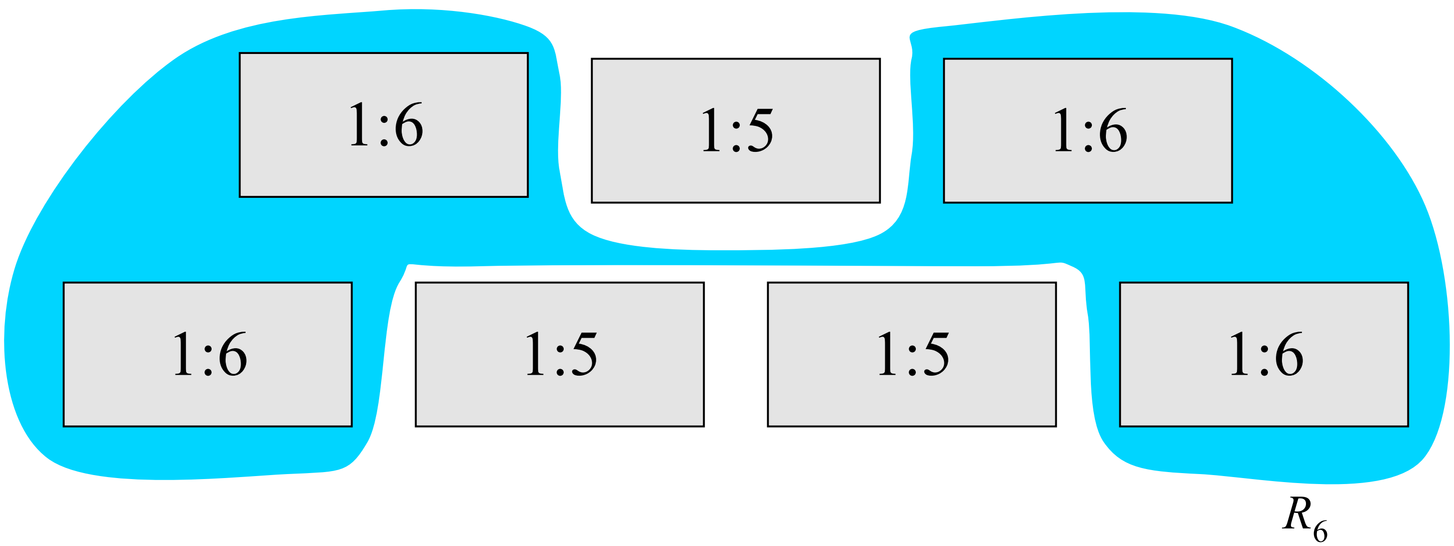
Condition $A(v)$

$N = 7,\ t = 3$, majority = 4



Condition $A(v)$

$N = 7,\ t = 3$, majority = 4

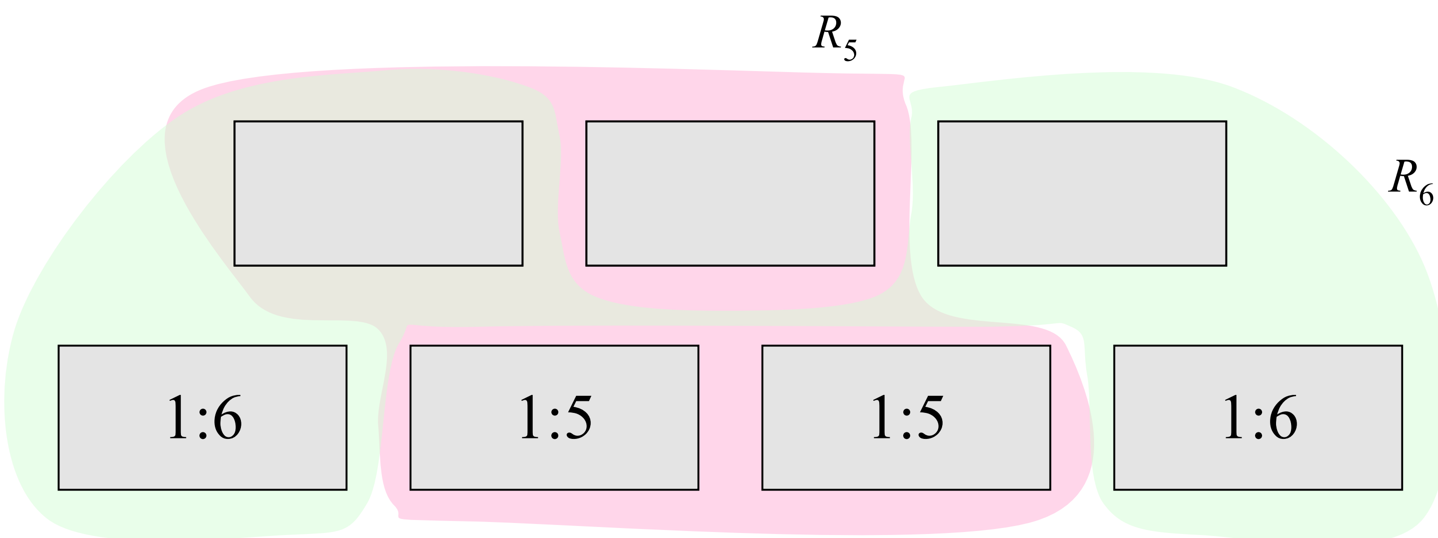


Picking value for 2a round i

- $A(v)$ and $A(w)$ for $v \neq w$:
  - There are two k-quorums R_v and R_w:
    - $\forall\ a \in R_v \cap Q$: $vv_a = v$.
    - $\forall\ a \in R_w \cap Q$: $vv_a = w$.
    - Note that $R_v \cap R_w$ $R_w \cap Q$, and $R_v \cap R_w$ are all non-empty, but $R_v \cap R_w \cap Q$ must be empty for this to hold.

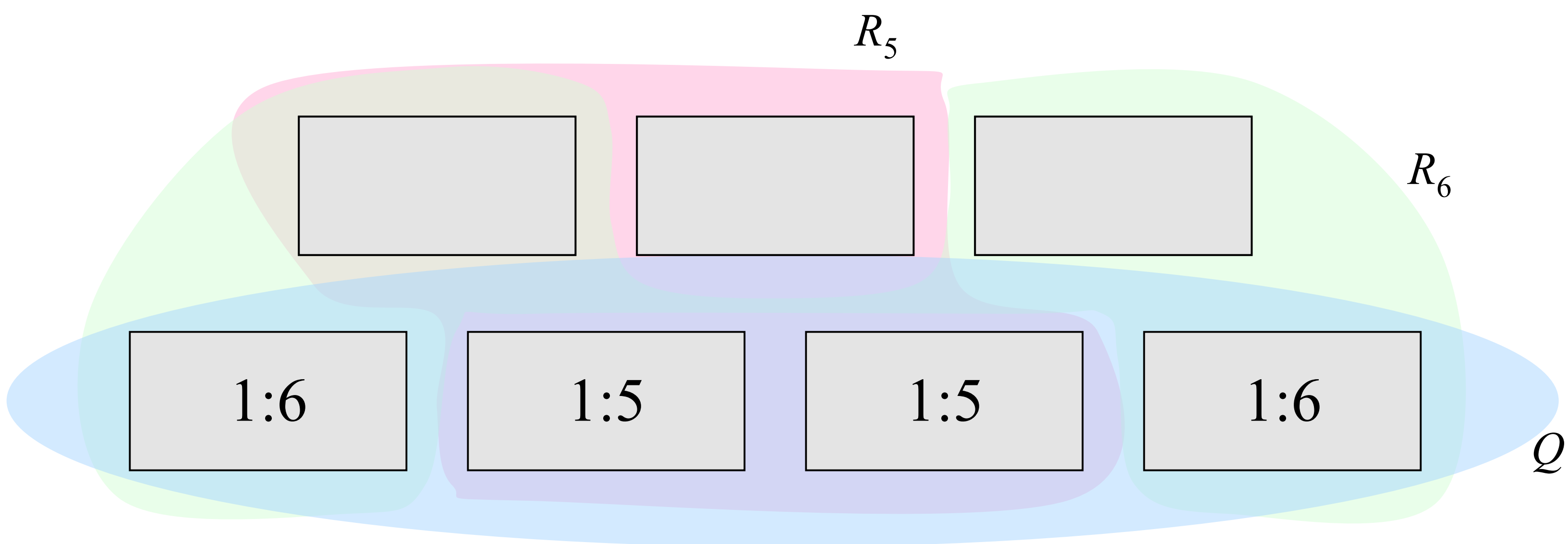
Condition $A(v)$

$N = 7,\ t = 3$, majority = 4



Condition $A(v)$

$N = 7,\ t = 3$, majority = 4



Picking value for 2a round *i*

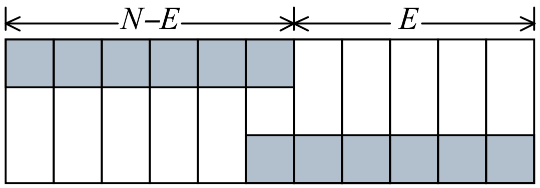
… so, let's ensure $R_v \cap R_w \cap Q$ is not empty!

For any rounds *i* and *j*:

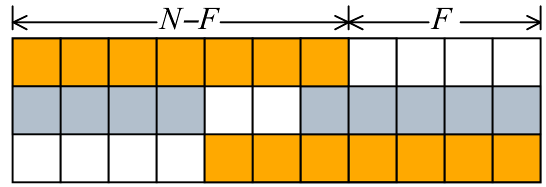
- Any *i*-quorum and *j*-quorum have a non-empty intersection.
- If *j* is a fast round, then any *i*-quorum and two *j*-quorums have a non-empty intersection.

Classic Quorums and Fast Quorums

- If there are N acceptors, choose E and F:
  - $N - E$ acceptors are a classic quorum
  - $N - F$ acceptors are a fast quorum
  - Since fast quorums have more stringent requirements, they should be at least as large as classic quorums: $E \geq F$.



$2(N-E) > N$, or $N > 2E$

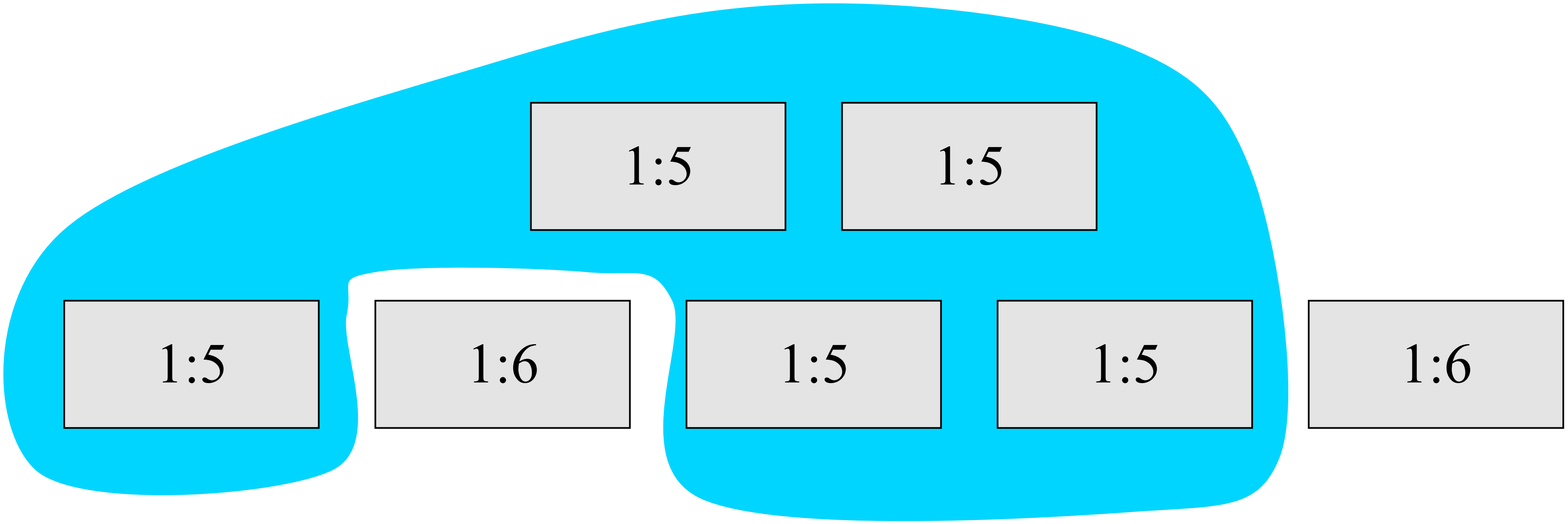


$N-E > 2F$, or $N > 2F + E$

- Maximize F: $E = F = \lceil N/3 \rceil$
- Maximize E: $E = \lceil N/2 \rceil - 1$, $E = \lfloor N/4 \rfloor$
    - eg, if $t = 2$, can have $N = 7$, $E = F = 2$.
    - or, can have $N = 5$, $E = 2$, $F = 1$ and run only classic runs if appears that there are two failures.

Condition $A(v)$

$N = 7,\ E = F = 2$



Picking value for 2a round i

- **let** Q be an i-quorum of acceptors that have sent round i phase 1b messages $promise(vr_a, vv_a)$.
- **let** k be the largest vr_a for all $a \in Q$.
- **let** V be the set of values vv_a for all $a \in Q$ with $vr_a = k$.
- **if** $k = 0$ **then** pick any proposed value or pick *any*
  else if V contains only one element w then pick w
  else if $w \in V$ that satisfies $A(w)$ then pick w
  else pick any value in V

Collisions

- Two or more proposers send proposals at about the same time, and they are received in different orders by different acceptors.
  - This may result in no value being chosen.
  - A collision in round *i* will be noticed by learners if they do not receive an *i*-quorum of identical values.
  - Coordinators can notice if they are also learner.
- Suppose *c* receives round *i* phase 2*b* *learn*(*i*, *v*) from *a* followed by round *i*+1 phase 1b *promise*(*r*, *v*) from *a*.
  - Both report *a*'s vote for round *i* and promise that *a* will not vote on any round less than *i*+1.
  - So, if coordinator has received round *i* *learn*(*i*, *v*) and is starting round *i*+1, it doesn't need *a*'s round *i*+1 *promise* message.

Coordinated Recovery

- i is a fast round and c coordinates both rounds i and $i + 1$.
  - Once c receives round i *learn* messages from an $(i + 1)$-quorum it starts phase 2a of round $i + 1$.
  - Since V is non-empty, c does not send *accept*($i + 1$, *any*).
  - This will succeed if the acceptors in a nonfaulty $(i + 1)$-quorum receive these messages before receiving any message from a higher round.

Uncoordinated Recovery

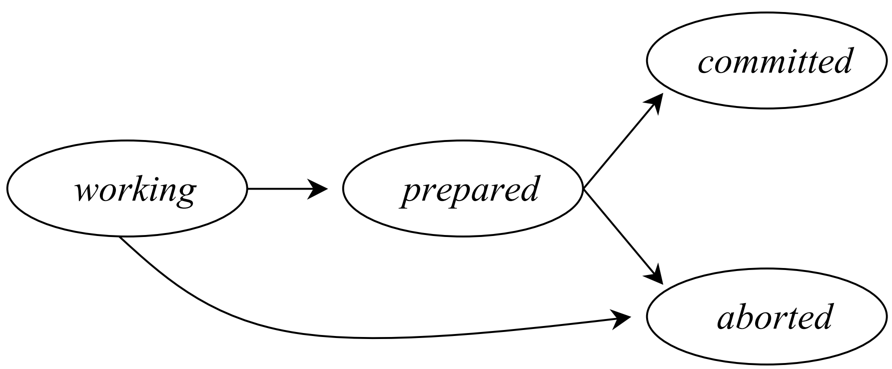
- Both i and $i + 1$ are fast rounds.
  - acceptors send *learn* messages to all other acceptors.
  - Each acceptor uses the same procedure as in coordinated recover to pick a value v the the coordinator could send in a round $(i + 1)$ *accept* message.
    - Nondeterminism means different acceptor could vote for different values.
    - Since $i + 1$ is a fast round,consistency is preserved and a higher-numbered round can still choose a value.

Paxos and Atomic Commit

- Recall *atomic commit* protocols.
  - 2 phase commit is blocking in the face of one (or two) well-placed failures.
  - 3 phase commit does not have this weakness, but it requires leader election (which is a perfect failure detector).
  - Consensus only requires ◇W…

Atomic Commit

- Resource managers (RM) agree on *commit* or *abort*.
- Decision directed by a transaction manager (TM), which can be a resource manager as well.
  - *Stability*: Once an RM has entered *committed* or *aborted* state, it remains in that state.
  - *Consistency*: It is impossible for one RM to be in the *committed* state and another to be in the *aborted* state.



Paxos Commit

- In 2PC, one TM collects the abort/commit votes from the RMs, decides the outcome, and disseminates the result to the RMs.
- In 3PC, if the TM fails, then its failure is detected and a new TM is elected.
  - Created a *precommitted* state to ensure there was no state in which it was possible for an RM to be *committed* and in which it was possible for an RM to be *aborted*.
- We could have TM use consensus once it knows all RMs are in *prepared* state.
  - … but there is a more message efficient approach: we will have $2f + 1$ acceptors in the role of transaction managers.

Paxos Commit: algorithm I

- Each resource manager has its own instance of Paxos to agree to *prepare* or *abort*.
  - If each instance chooses *prepare* then commit; else abort.
  - All instances use the same leader and set of acceptors.
    - Acceptors/leader know the set of RM involved in the transaction.

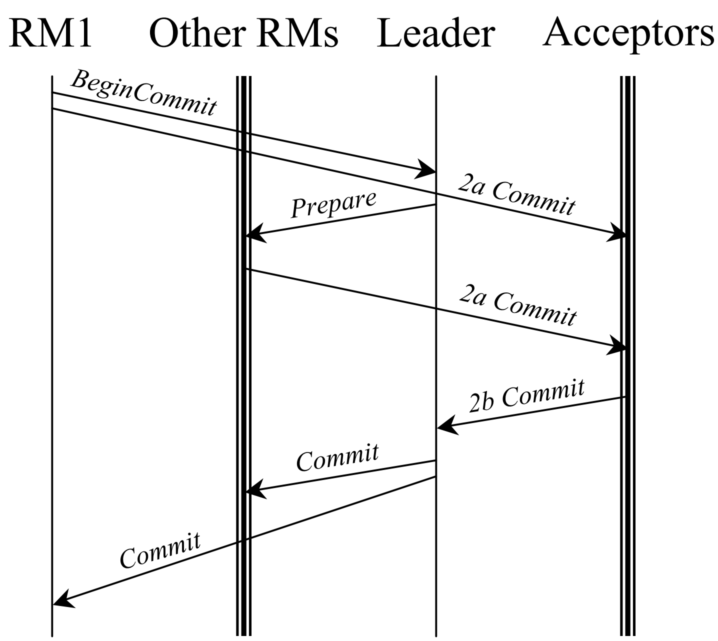
Paxos Commit: algorithm III

- When an RM *i* decides to prepare, it sends on behalf of the leader a phase 2a message *accept*(1, $vote_i$).
  - This is a phase 2a ballot 1 message. There's no need for this to have to come from the leader nor to run phase 1.
- When leader receives $f + 1$ *learn* messages, it can send phase 3 messages announcing outcome to RMs.
- Transaction committed iff every RM chooses prepared; otherwise is aborted.
  - For efficiency, each acceptor can bundle phase 2b *learn* messages for all instances into one message.
  - Similarly, the leader can distill all phase 3 *learn* messages into one message declared *commit* or *abort*.

Paxos Commit: algorithm II

- If a new leader starts ballot > 1, then it runs phase 1.
  - If it finds its choice unconstrained, then it should propose *abort* in phase 2.
- In fact, the only way that an instance of Paxos will decide *prepared* is if the associated RM sends *accept*(1, *prepare*).
  - Thus, if an RM i has $vote_i = abort$, then Fast Paxos can short-circuit and inform all processes that the decision is to abort.

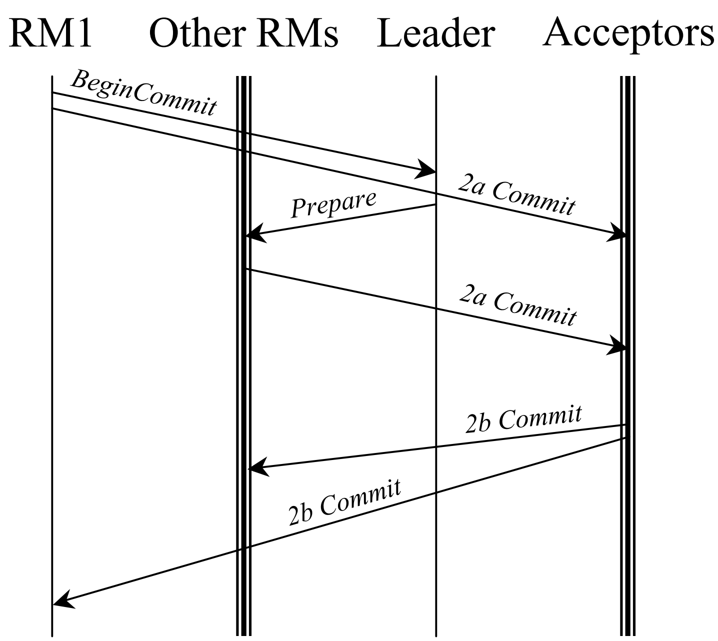
Cost: I



5 message delays, $(n+1)(t+3) - 4$ messages (with the leader an acceptor).

If each acceptor is on the same node as an RM, and the leader on RM1, then $n(t+3) - 3$ messages, but still 5 message delays.

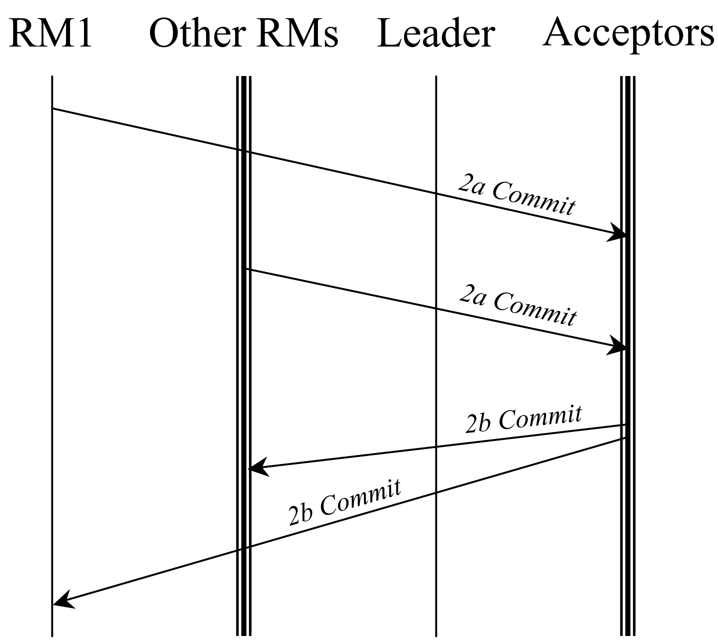
Cost: II



4 message delays, $n(2t+3) - 1$ messages

If each acceptor is on the same node as an RM, and the leader on RM1, then $(n-1)(2t+3)$ messages, but still 4 message delays.

Cost: III



2 message delays, which is optimal