Paxos

Paxos is a protocol for implementing the state machine approach in an asynchronous system.

The central part of Paxos is, like the $\diamondsuit$S protocol shown before, a crash-consensus protocol.

It may not terminate but it is always safe.

- Give a constructive argument for the consensus protocol.
- Show how it can be used to implement state machines.

Agents

There are three basic roles that take place in consensus:

- *Proposers* that propose a value for consensus;
- *Acceptors* that choose the consensus value;
- *Learners* that learn the consensus value.

A single process may take on multiple roles - that is, act as more than one *agent* - but we can ignore this detail for now.

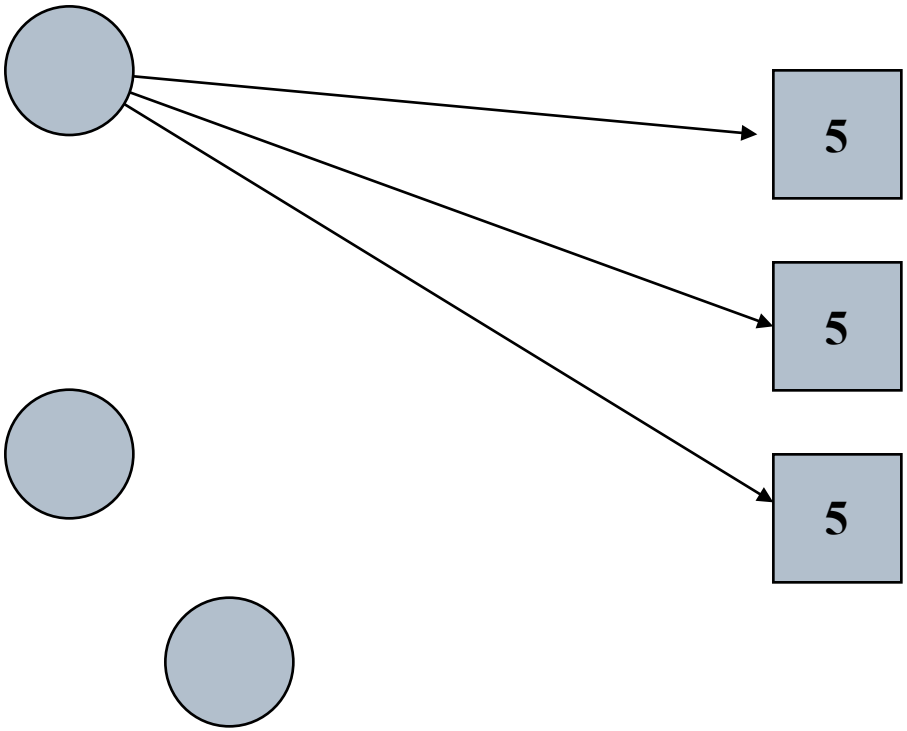
Failure Model

- Agents operate at arbitrary speed, may fail by stopping, and may restart. An agent can remember information across restarts, using local stable storage.
- Communications is by messages that can take arbitrarily long to deliver. Messages can be duplicated and lost, but not corrupted.

Choosing a Value (I)

- If there is only one acceptor, then choosing a value is easy: proposers send a proposal to the acceptor, which chooses the first proposal it receives.
- This solution assumes that the acceptor doesn't fail.
- Can instead have multiple acceptors. A value is chosen when a large enough set of acceptors have voted for it.
- If an acceptor can vote for at most one value, then a "large enough set" is a majority of acceptors.

Choosing a Value



5 is chosen

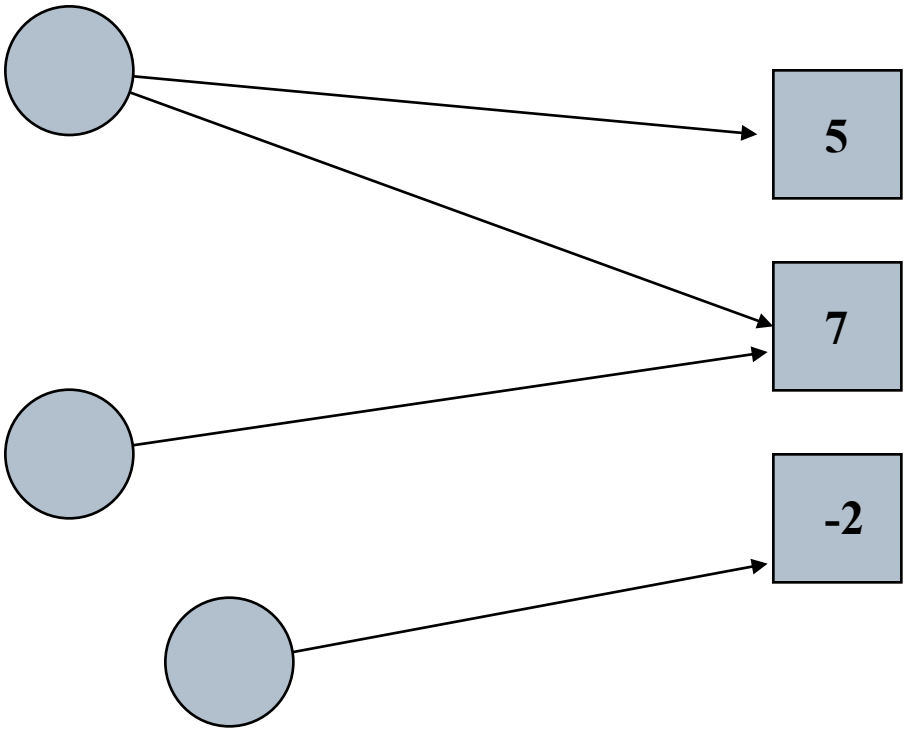
Choosing a Value (II)

- What if only one value is proposed by a single proposer? We would want that value to be chosen. So, we have the requirement:

P1: An acceptor votes for the first proposal that it receives.

... however, simultaneous proposals may lead to no majority of acceptors voting for the same value.

Choosing a Value



no value is chosen

Choosing a Value (III)

... so, acceptors need to be able to vote for more than one proposal.

We keep track of the different proposals that an acceptor votes for by assigning a natural number to each proposal.

- A proposal thus consists of a proposal number and a value.
- Different proposals have different numbers.
- A value is chosen when a single proposal with that value has been voted for by a majority of acceptors.

Choosing a Value (IV)

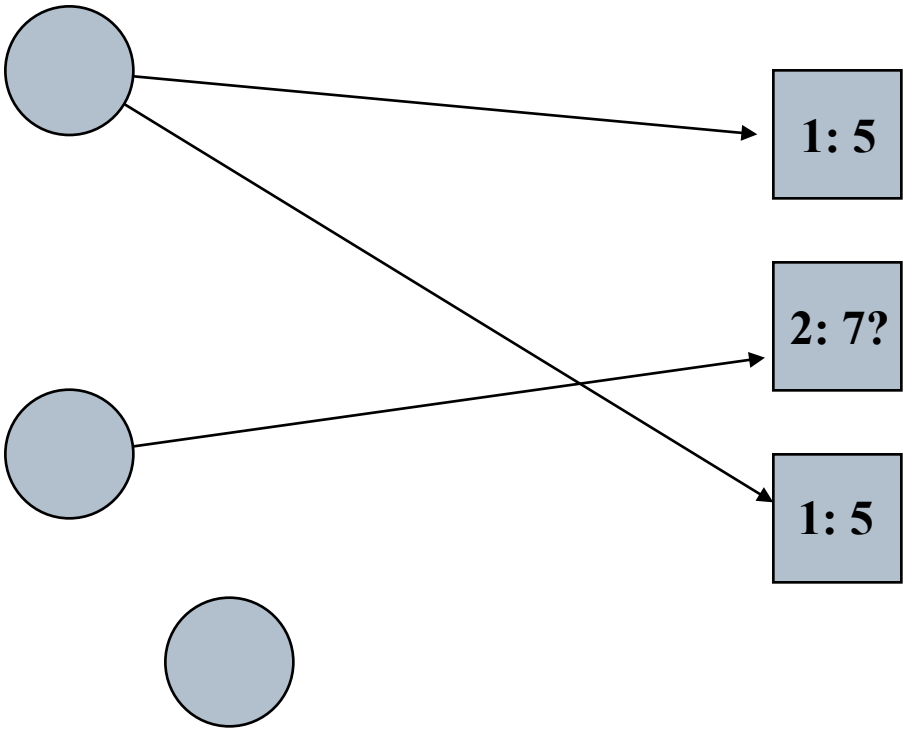
- We need to guarantee that all chosen proposals have the same value. It suffices to guarantee:

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 voted for by any acceptor has value *v*.

Choosing a Value



5 is chosen

Choosing a Value (V)

- P1 is still needed to ensure that *some* proposal is accepted.

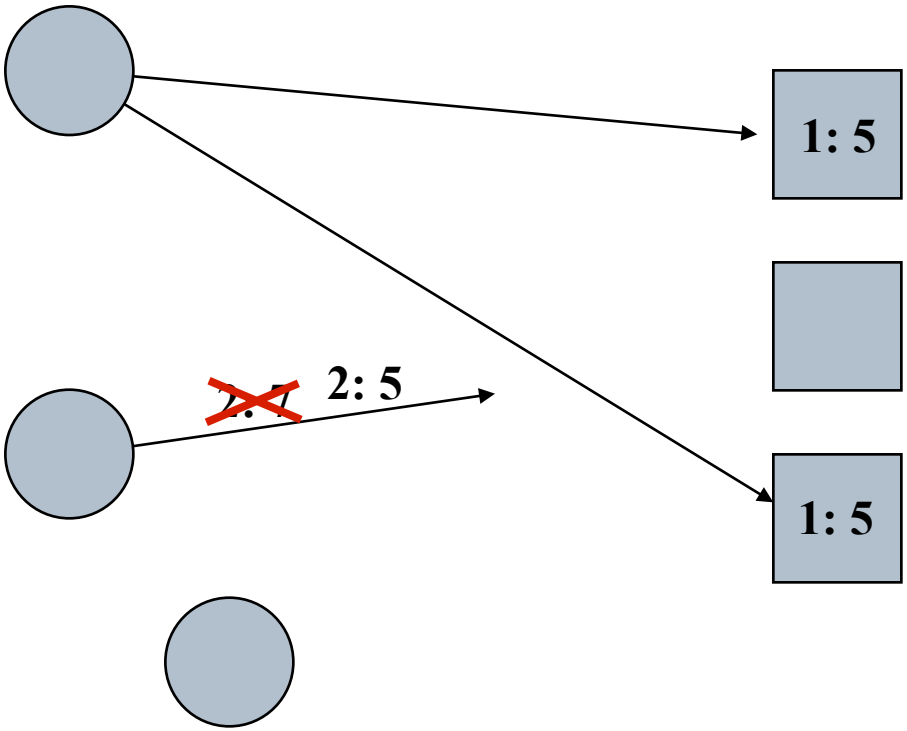
... asynchronism adds a difficulty: there can be an acceptor *a* that never receives any proposals for a long time. Then, a new proposer issues a higher-numbered proposal with a different value. If *a* receives this proposal, then P2a will be violated.

Choosing a Value (VI)

- We solve this problem by strengthening P2a to:

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

Choosing a Value



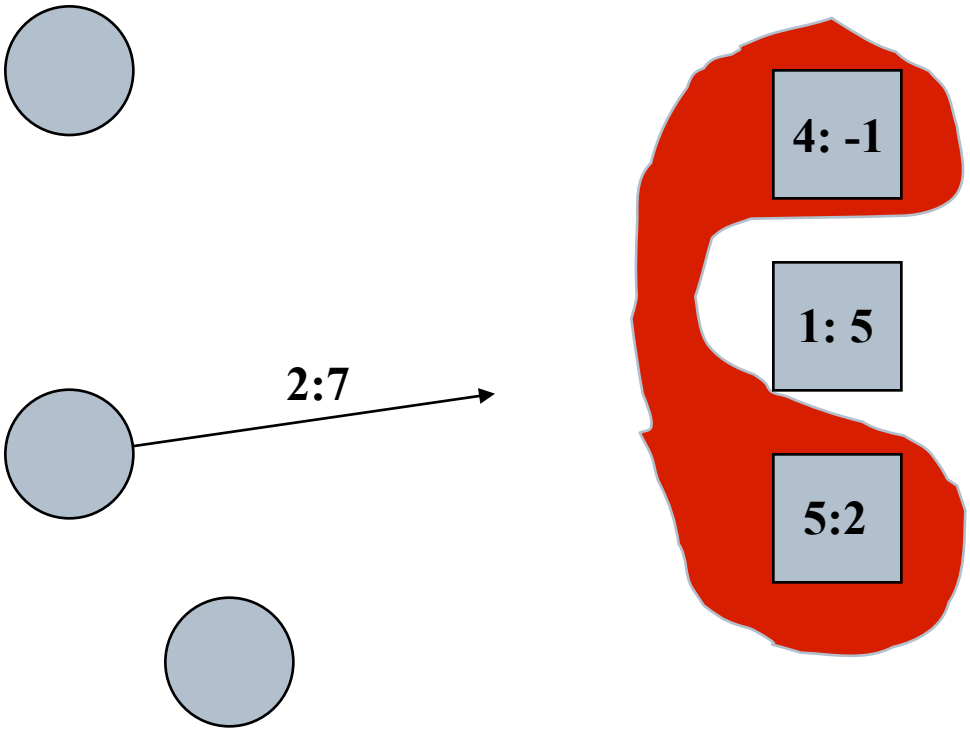
Choosing a Value (VII)

We can satisfy P2b by maintaining the following invariant:

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

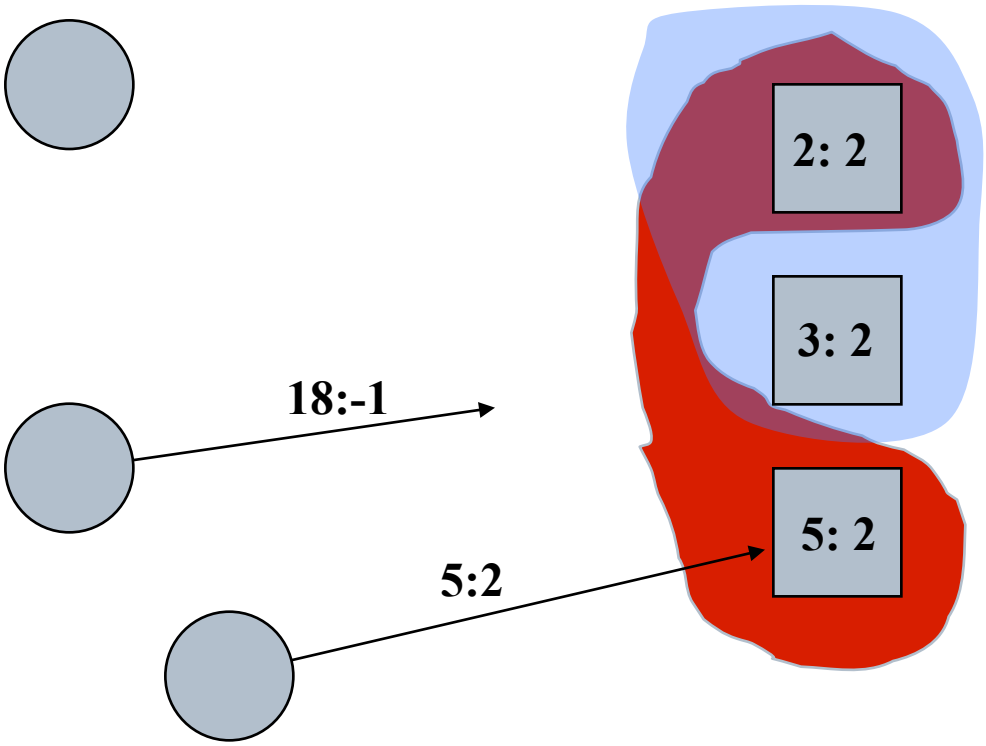
- no acceptor in *S* has voted for any proposal numbered less than *n*, or
- *v* is the value of the highest-numbered proposal among all proposals numbered less than *n* and voted for by the acceptors in *S*.

Choosing a Value



no acceptor in *S* has voted for any proposal numbered less than *n*.

Choosing a Value



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

Choosing a Value (VIII)

- 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 voted for by each acceptor in some majority of acceptors.

Choosing a Value (VIII)

- Avoid predicting the future by *extracting a promise* from a majority of acceptors not to subsequently vote for any proposals numbered less than *n*.

Choosing a Value (IX)

Here is the resulting algorithm for issuing a proposal:

1) A proposer chooses a new proposal number *n* and sends a request to each member of some set of acceptors, asking it to respond with:
   a) A promise never again to vote for a proposal numbered less than *n*, and
   b) The proposal with the highest number less than *n* that it has voted for, if any.

   ... call this a *prepare* request with number *n*.

Choosing a Value (X)

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

   A proposer issues a proposal by sending, to some set of acceptors, a request that the proposal be voted for. Call this an *accept* request.

Choosing a Value (XI)

What do acceptors do?

- 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, thereby voting for the proposal, iff it has not promised not to, e.g.

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

... which implies P1: An acceptor votes for the first proposal that it receives.

Choosing a Value (XII)

- If an acceptor receives a *prepare* request *r* numbered *n* having already responded to a *prepare* request numbered greater than *n*, then the acceptor can simply ignore *r*.
- It can also ignore *prepare* requests to which it has already responded.

  ... so, an acceptor only needs to remember the maximum proposal number for which it has participated, the highest numbered proposal it has voted for, and the number of the highest-numbered *prepare* request to which it has responded.

  rnd_a: highest proposal number in which *a* has participated.
  $vrnd_a$: highest proposal number in which *a* cast a vote.
  $vval_a$: the value *a* voted for with proposal number $vrnd_a$.

  This information needs to be stored on stable storage to allow *a* to recover after a crash.

- A coordinator *c* maintains

  $crnd_c$: highest proposal number *c* has begun.
  $cval_c$: the value *c* picked for proposal number $crnd_c$.

Choosing a Value: Pseudocode

Phase 1:

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

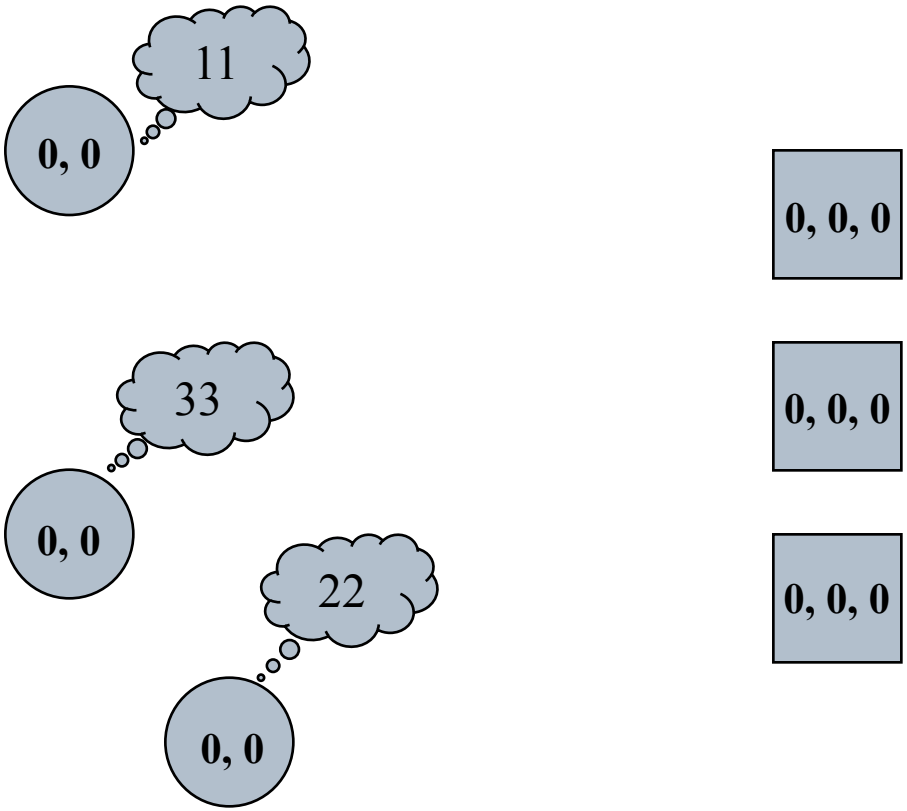
b. Acceptor a receives *prepare*(n) from c:
   i. If $n > rnd_a$ then set rnd_a to n and send *promise*(rnd_a, $vrnd_a$, $vval_a$) to c.
   ii. Else ignore request.

Phase 2:

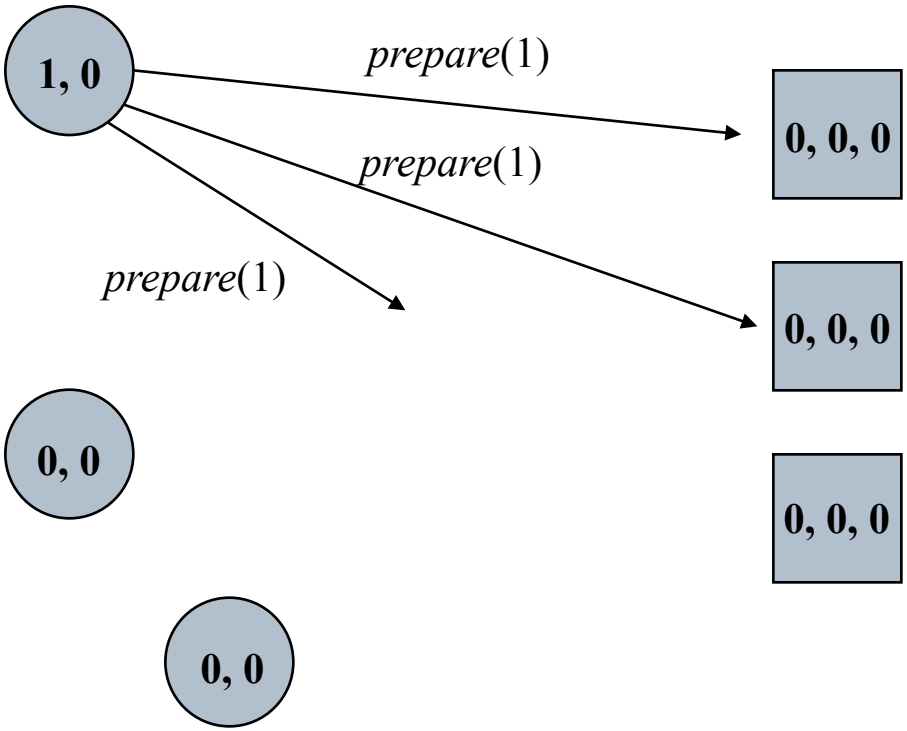
a. Proposer c receives *promise*(rnd_a, $vrnd_a$, $vval_a$) from a majority of acceptors with $rnd_a = crnd_c$:
   i. If all reply with $vrnd_a = 0$, then set $cval_c$ to any proposed value
   Else set $cval_c$ to $vval_a$ associated with largest received value of $vrnd_a$.
   ii. Send *accept*($crnd_c$,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

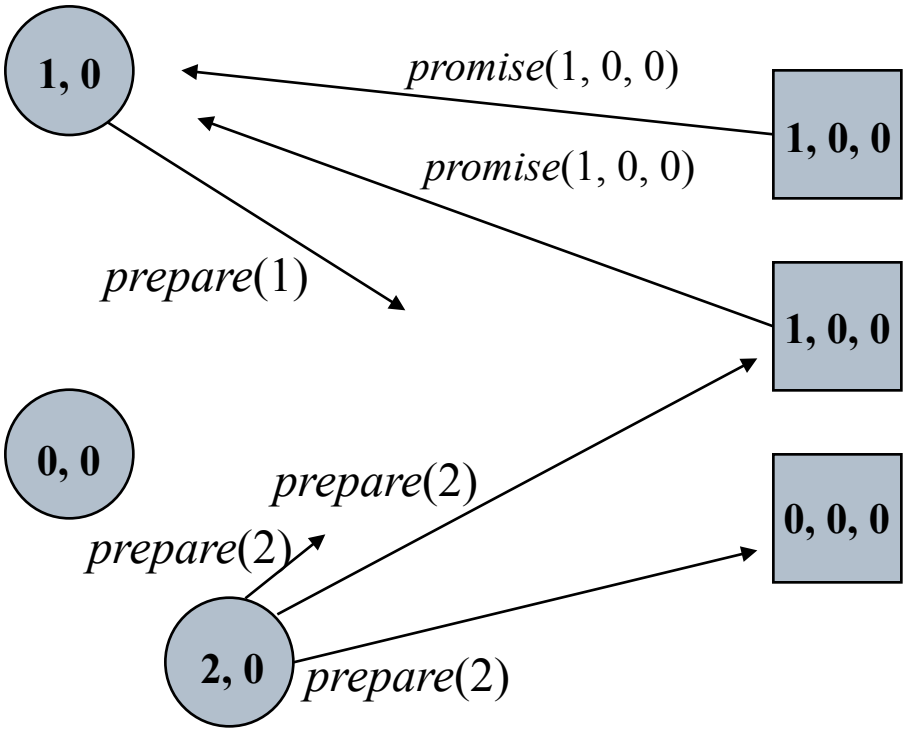
Paxos in Action: I



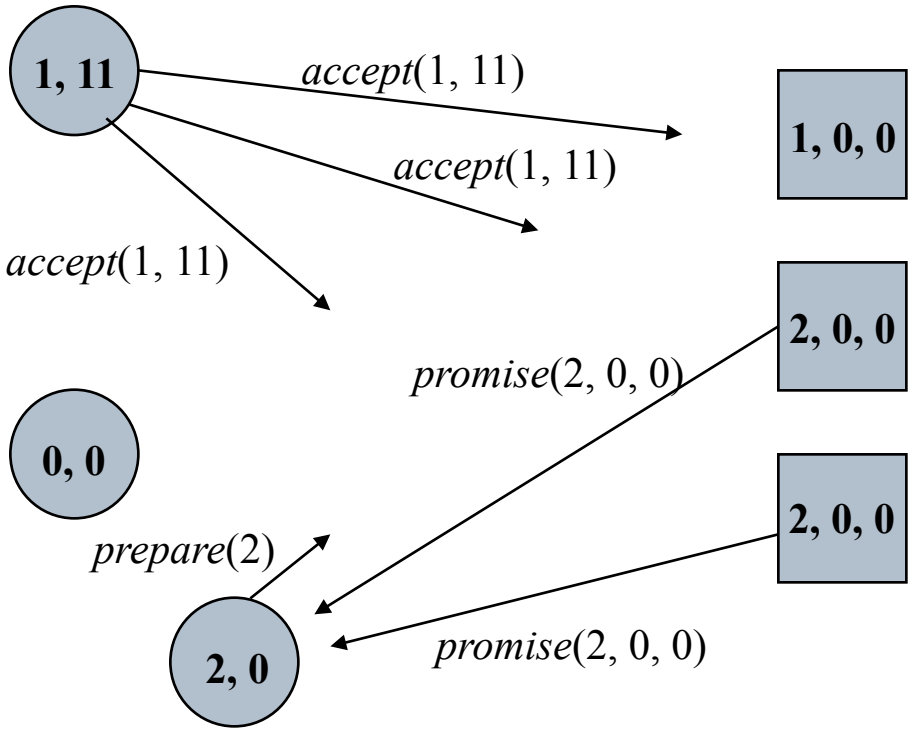
Paxos in Action: II



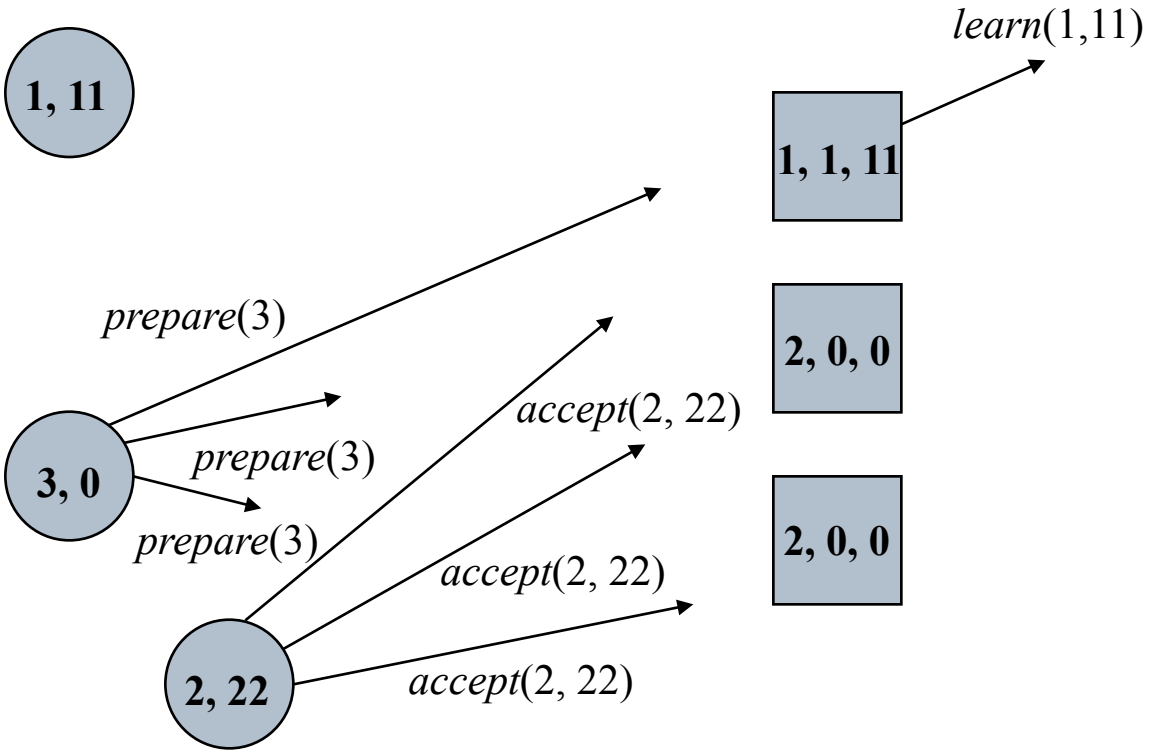
Paxos in Action: III



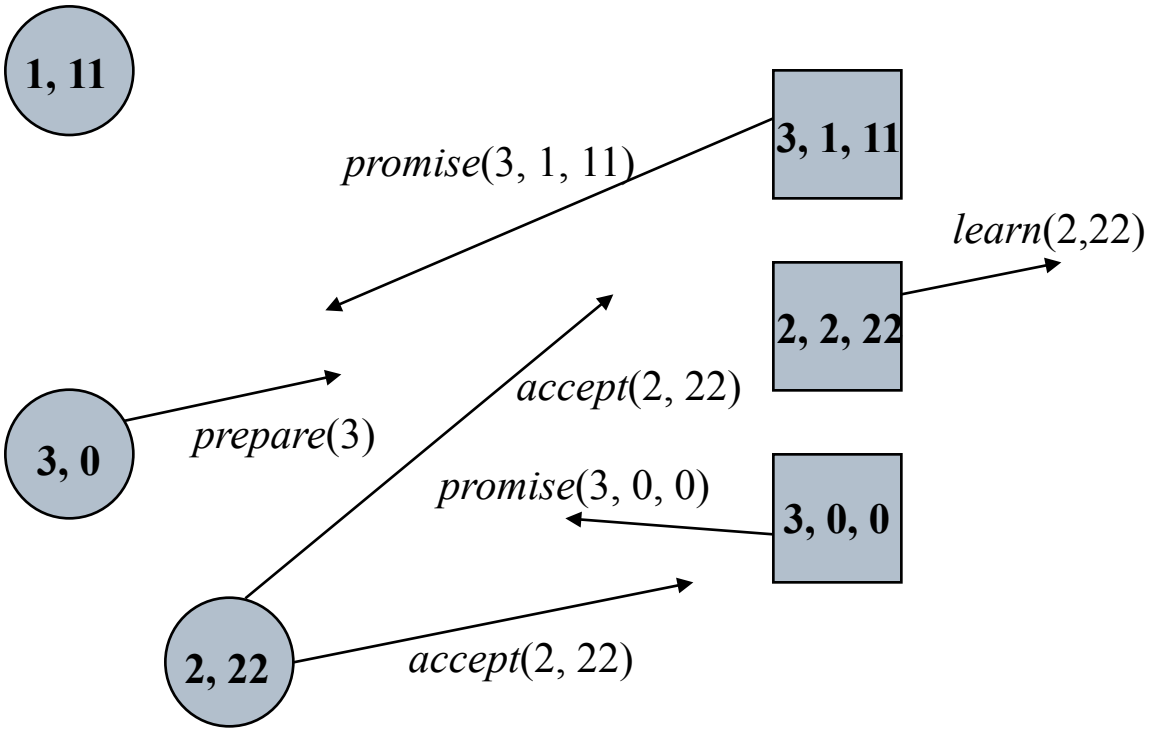
Paxos in Action: IV



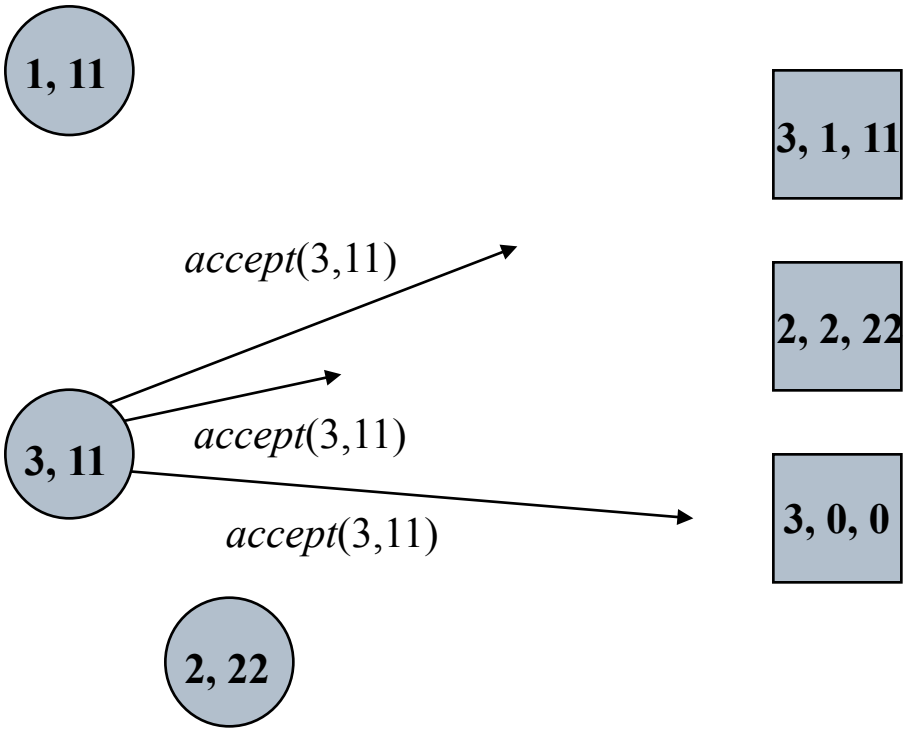
Paxos in Action: V



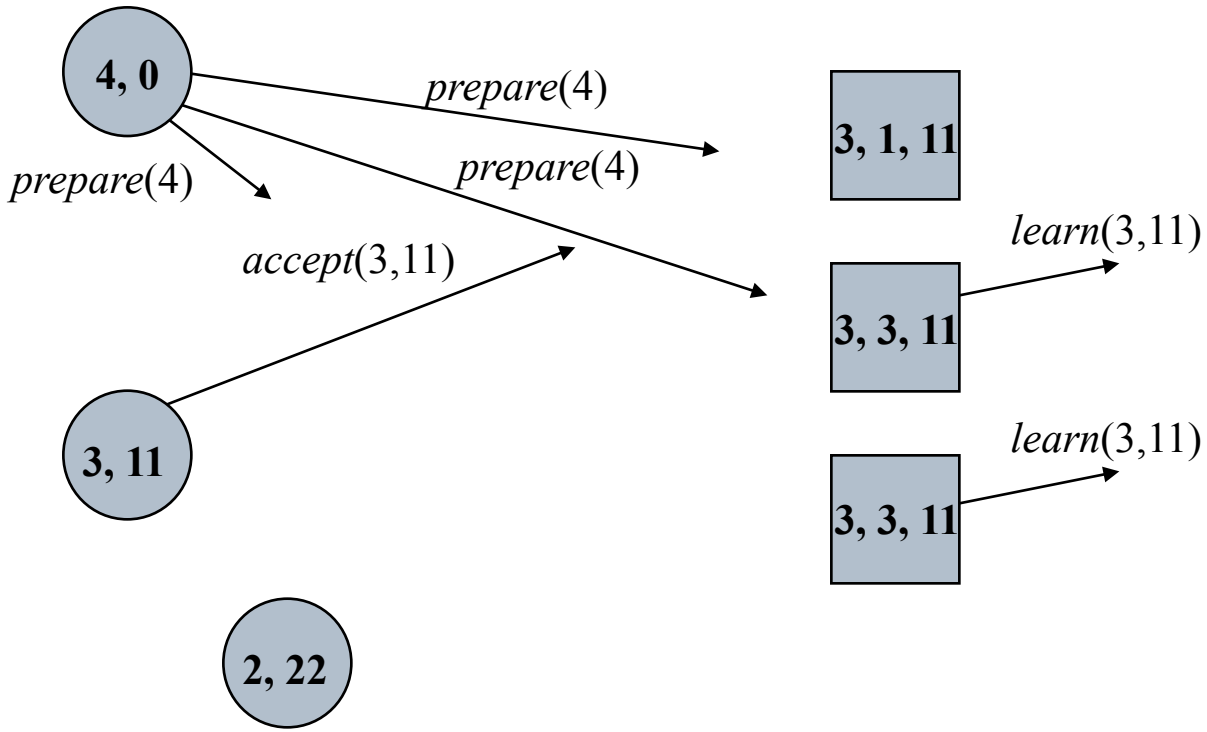
Paxos in Action: VI



Paxos in Action: VII



Paxos in Action: VIII

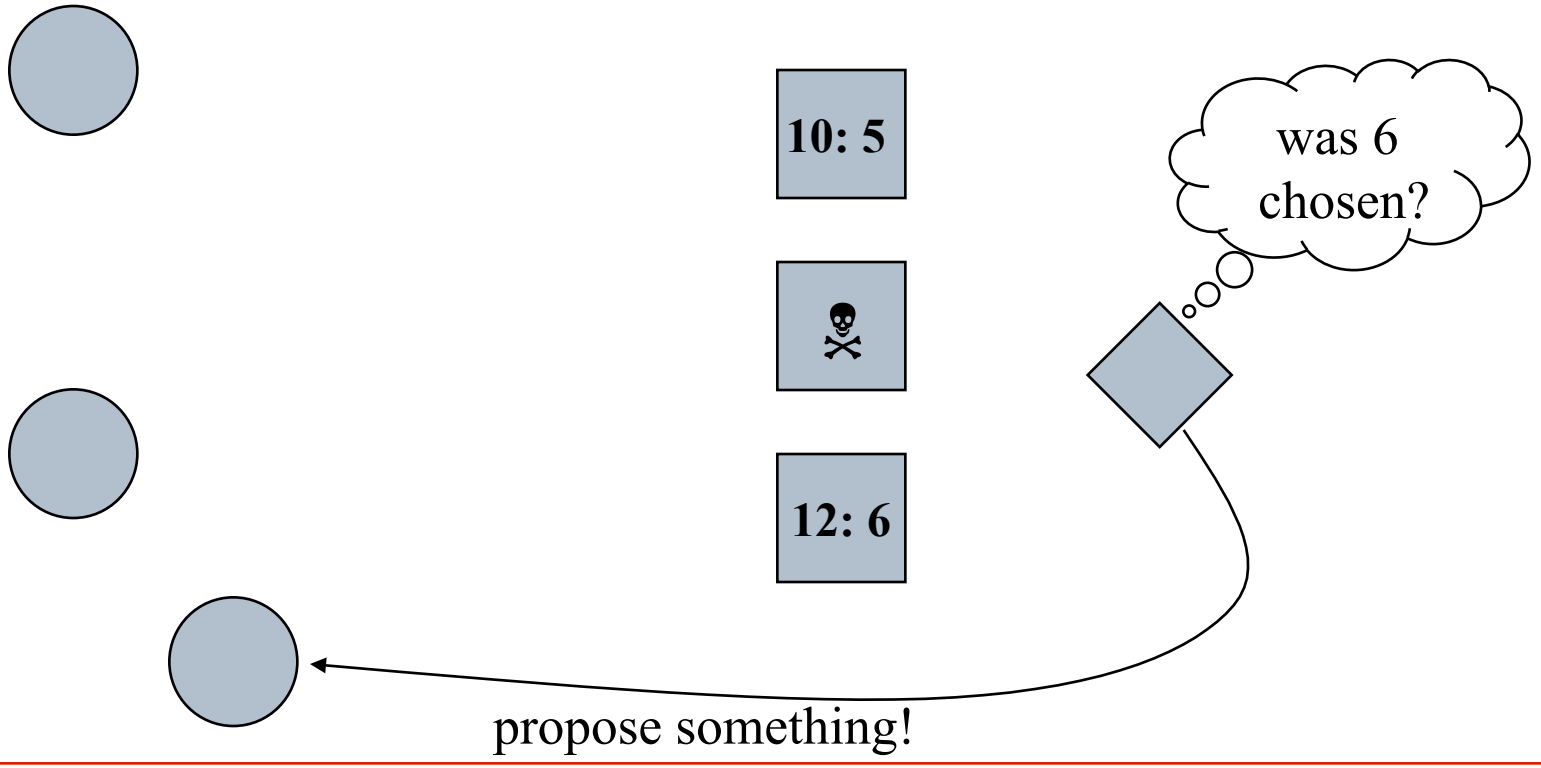


Learning a Chosen Value (I)

- 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.

◇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 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**; }
}

◇S Consensus as Paxos

- All processes are acceptors.
- Each round has a *distinguished proposer* and a *distinguished listener* ($r \bmod n$) + 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…

- $\diamondsuit$W is the weakest failure detector that solves consensus.

  It's equivalent to $\diamondsuit$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*, which in pathological 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.