Paxos

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

The central part of Paxos is, like the ◊S protocol shown before, a crash-consensus protocol that may not terminate but 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.

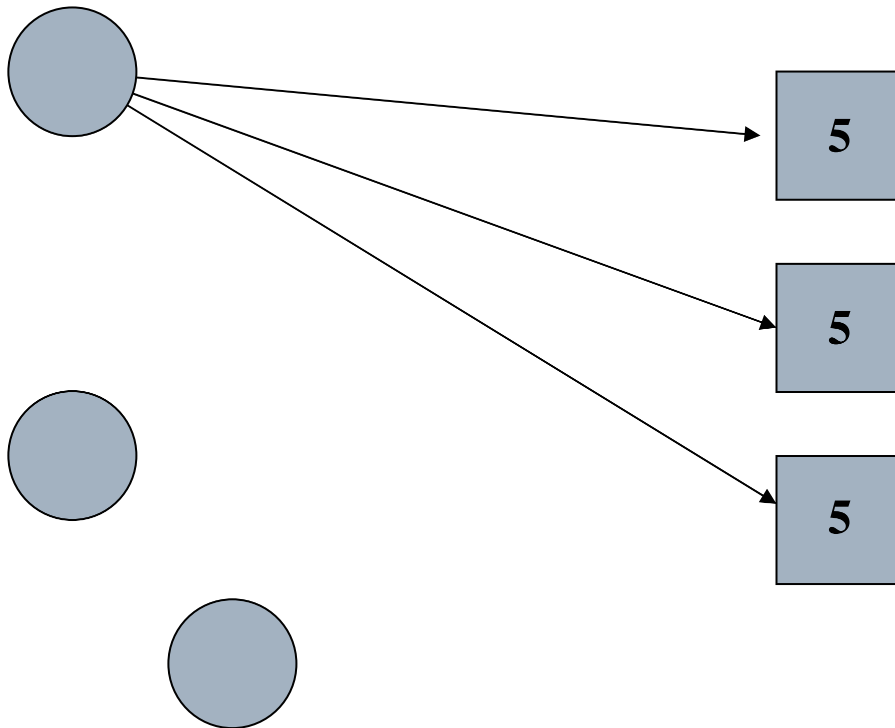
Failure Model

- 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 accepted it.
- If an acceptor can accept at most one value, then a "large enough set" is a majority of acceptors.

Choosing a Value



5 is chosen

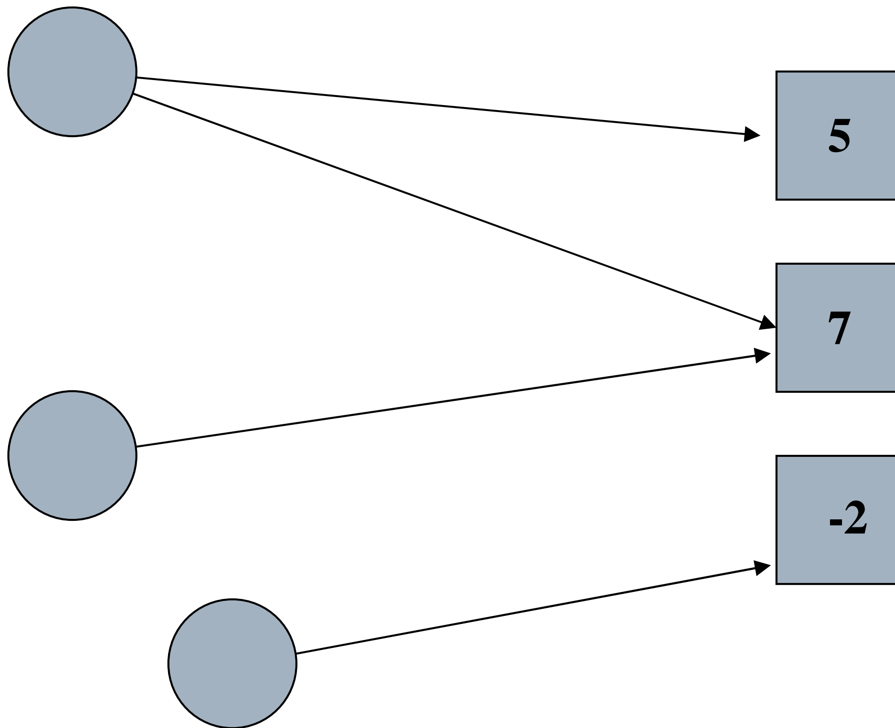
Choosing a Value (II)

- 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 must accept the first proposal that it receives.

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

Choosing a Value



no value is chosen

Choosing a Value (III)

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

Keep track of the different proposals that an acceptor may accept 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 accepted by a majority of the acceptors.

Choosing a Value (IV)

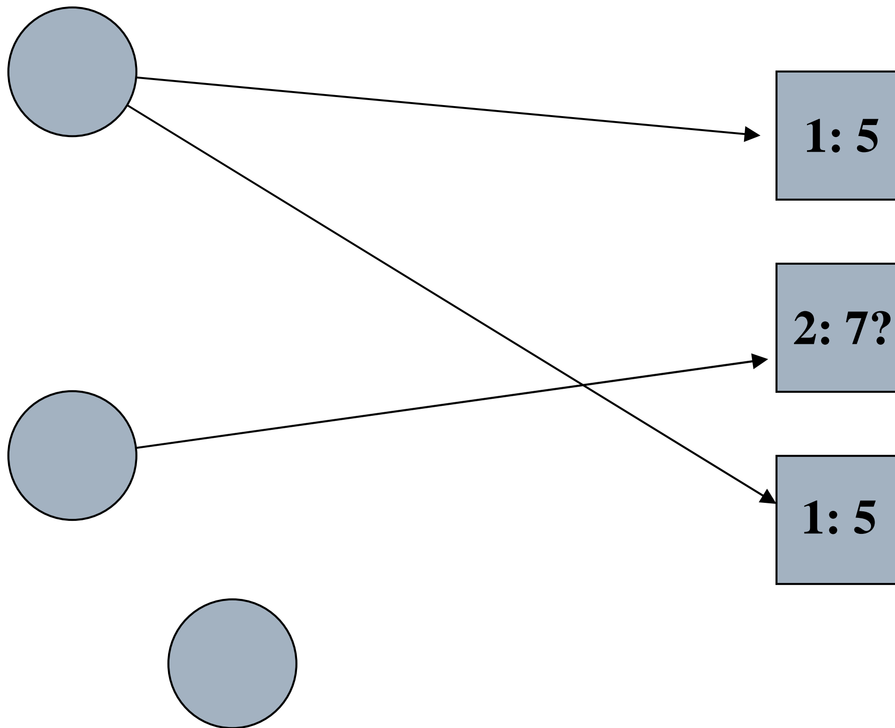
- We need to guarantee that all chosen proposals have the same value. It suffices to guarantee:

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 accepted 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.

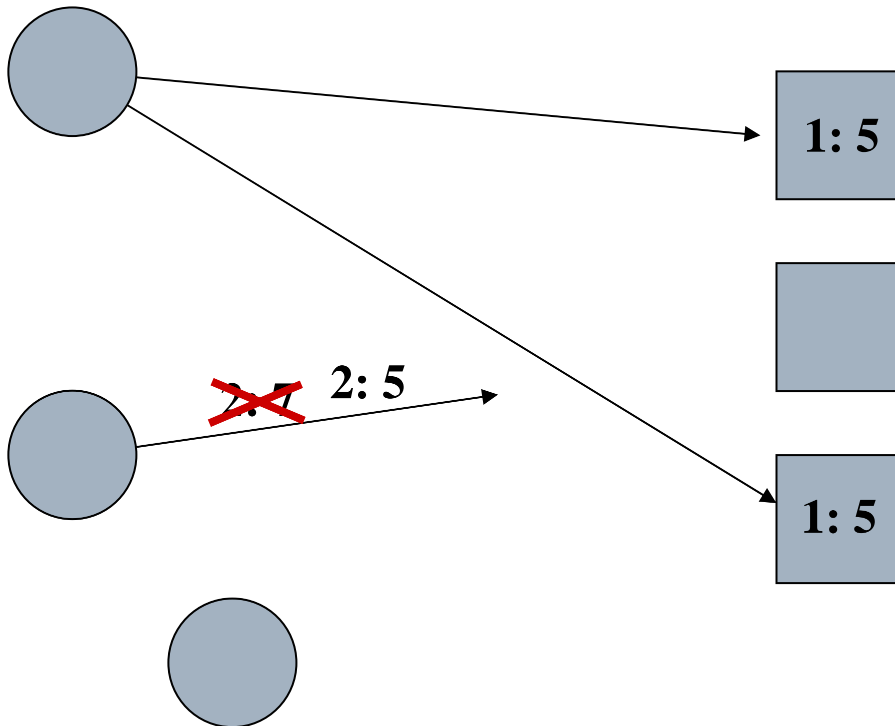
... asynchronism adds a difficulty: there can be an acceptor *a* that never receives any proposals for a long time. Then, a new proposer issues a higher-numbered proposal with a different value. If *a* receives this proposal, then P2a would be violated.

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



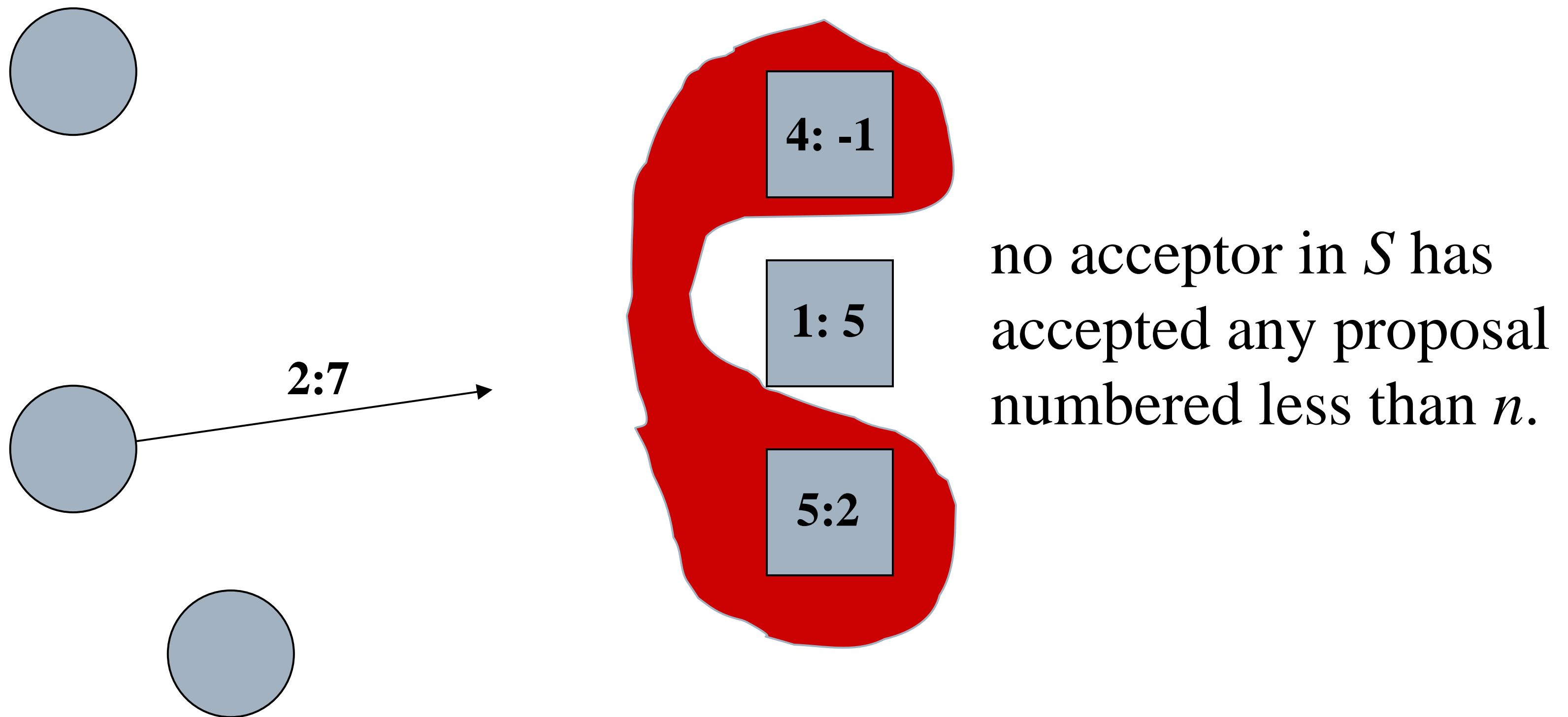
Choosing a Value (VII)

We can satisfy P2b by maintaining the following invariant:

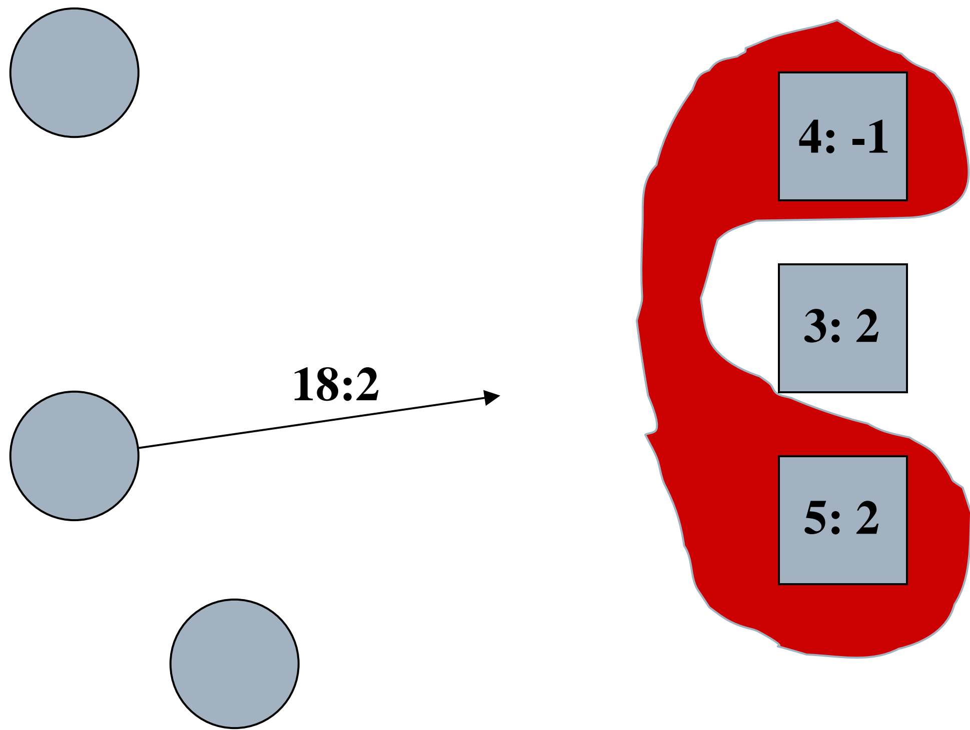
P2c: For any v and n, if a proposal with value v and number n is issued, then there is a set S consisting of a majority of acceptors such that either

- 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 and accepted by the acceptors in S.

Choosing a Value

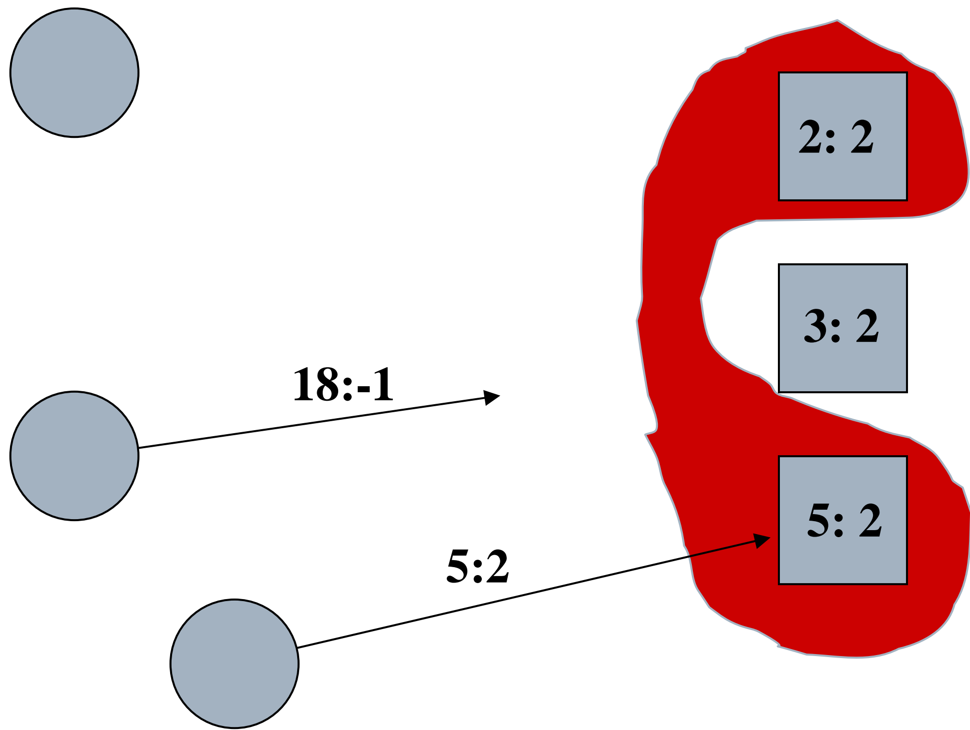


Choosing a Value



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

Choosing a Value



v is the value of the highest-numbered proposal among all proposals numbered less than *n* and accepted 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 accepted 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 accept 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 accept a proposal numbered less than n, and
   b) The proposal with the highest number less than n that it has accepted, 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 accepted. 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, 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 a number $> n$.

... which implies P1.

Choosing a Value (XII)

A small optimization:

- 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 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: Summary

Phase 1:

a) A proposer selects a proposal number *n* and sends a *prepare* request with number *n* to a majority of acceptors.

b) If the acceptor receives a *prepare* request with number *n* greater than any that of any *prepare* request to which it has already responded, then it responds to the request with a promise not to accept any more proposals numbered less than *n* and with the highest-numbered proposal (if any) that it has accepted.

Phase 2:

a) If the proposer receives a response to its *prepare* requests (numbered *n*) from a majority of acceptors, then it sends an *accept* request to each of those acceptors for a proposal numbered *n* with a value *v*, where *v* is the value of the highest-numbered proposal among the responses, or is any value if the responses report no proposals.

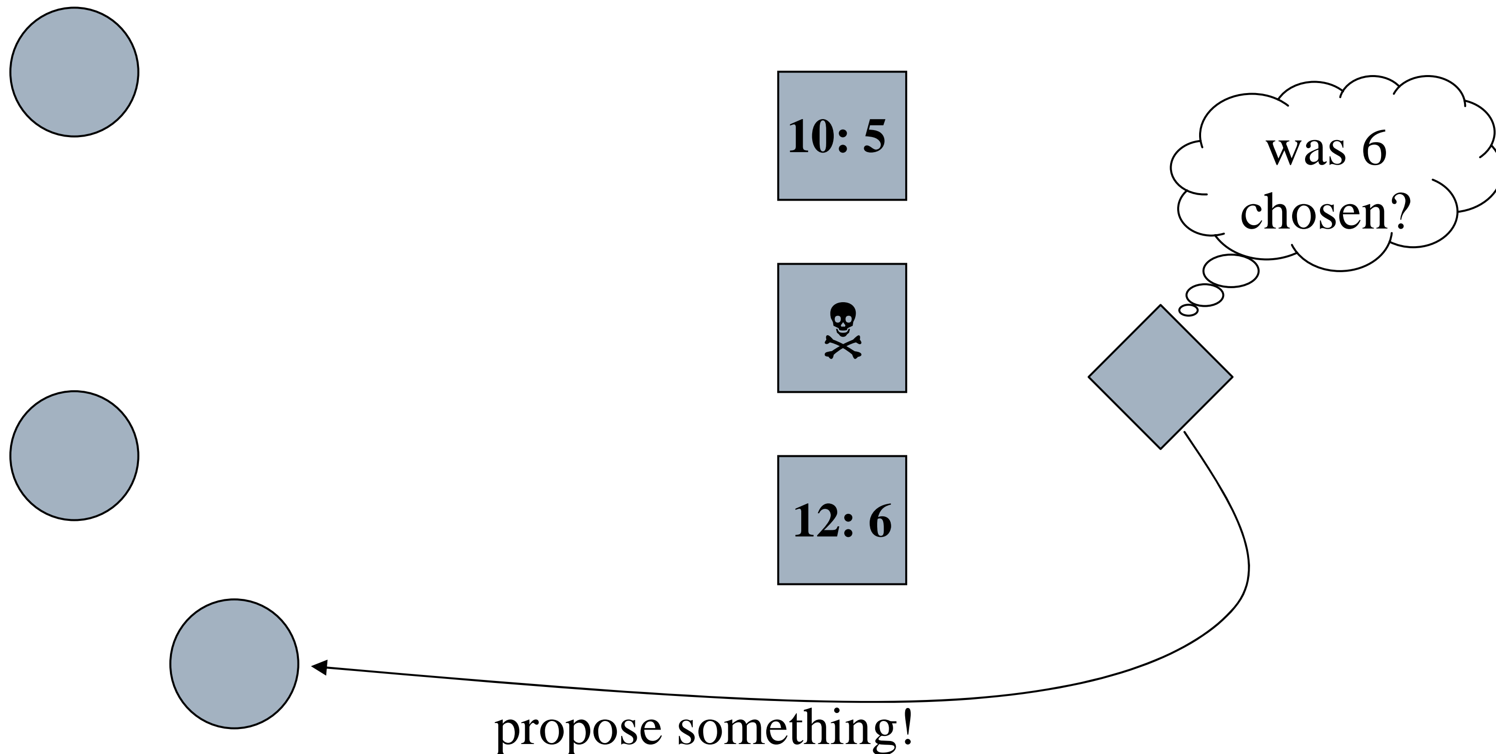
b) If an acceptor receives an *accept* request for a proposal numbered *n*, it accepts the proposal unless it has already responded to a *prepare* request having a number greater than *n*.

Learning a Chosen Value (I)

- A learner must find out that a proposal has been accepted 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.



◊S Consensus

```
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;
        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.

Implementing State Machines (I)

- We implement a sequence of separate instances of consensus, where the value chosen by the ith instance is the ith command in the sequence.
- 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 kth command, it tries to have the command chosen as the value in the kth 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 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 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)

- λ can execute 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.
- λ is free to propose (via phase 2) commands for 17 and higher. Gaps can occur (such as the missing 11 and 12) because of lost messages and asynchronous communications and then having λ fail.

Implementing State Machines (VI)

- 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, l 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 just running phase 2 of the asynchronous consensus, which is optimal in terms of delay.

Implementing State Machines (VII)

- 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 fo 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 ith command.