ΠΑΞΟΣ
Paxos

One of the Top Twenty great escapes in the world :

Paxos can be found 14 km south of Corfu, 20 km to the east is the mainland of Greece. Paxos covers an area of 19 km^2 and is one of a cluster of picturesque small islands set in the Ionian Sea.

2004 is a very special year for Paxos as it has been declared the "*Cultural Village of Europe*".

Paxos tour map

- Motivation, context
- Paxos algorithm
- Dramatic reenactment ☺
- Paxos dissected
- Putting it into context
- Byzantine Paxos

Motivation: State machine replication

- Replicas act as a single server; accept & execute client requests (a.k.a. active server replication)
  - Each replica is a deterministic state machine
  - Each request is executed at each replica
  - Fault-tolerance through replication

- Challenge is to ensure that each replica receives same input sequence in the presence of faults
  - Replicas should *agree* on the order they accept client requests
  - Faults: crash/recover & byzantine failures,
    message losses, asynchrony

Paxos algorithm

Slides courtesy of Gregory Chockler

Formal model

- Asynchronous message passing
- Crash failures
  - can tolerate crash/recovery
- Reliable links
  - can be modified to work with eventually reliable links
- Ω – Leader Oracle
  - outputs one trusted process
  - from some point, all correct processes trust the same correct process
  - the weakest failure detector for Consensus

Synod consensus algorithm

- Leader based
  - each process has an estimate of who is the current leader
  - processes decide on the value proposed by a leader

- A completely asynchronous algorithm
  - never violates safety
  - never blocks
  - termination is guaranteed once Ω stops making mistakes

Algorithm structure

- Two phases: prepare + accept [+ decide]
  - Leader contacts a majority in each phase

- Problem: there may be multiple concurrent leaders
- Solution: *ballots* distinguish among values proposed by different leaders
  - unique, locally monotonically increasing
  - processes respond only to leader with highest ballot

Ballot numbers

- Pairs <num, process id>
- <b1, p1> **>** <b2, p2>, if
  - b1 > b2, or
  - b1=b2 and p1 > p2
- Leader p chooses a unique ballot locally
- Monotonically increasing ballot number
  - if latest known ballot is b, q
  - p chooses b+1, p

Two phases of Paxos

1. Prepare
   - leader chooses a new ballot number
   - leader learns outcome of all smaller ballots from majority

2. Accept
   - leader proposes a value with his ballot number
   - leader gets majority to accept his proposal
   - a value <u>accepted by a majority</u> can be <u>decided</u>

Prepare phase

```
Until decision is reached, try:
if leader (by Ω) then
    BallotNum:=<BallotNum.num+1, my proc id>;
    send (“prepare”, BallotNum) to all;


Upon receive (“prepare”, b) from i:
if b > BallotNum then
    BallotNum := b;
    send (“ack”, b, AcceptNum, AcceptVal) to i
else send (“abort”,b) to i
```

Accept phase

```
Upon receive ("ack", BallotNum, b, val) or ("abort",b)  from > n/2 :
    if ("abort",b) is received then start over;
    if all vals = ⊥ then myVal := initial value
    else myVal := received val with highest b;
    send ("accept", BallotNum, myVal) to all;

Upon receive ("accept", b, v) from i :
    if b ≥ BallotNum then
        BallotNum := b;   AcceptNum := b;   AcceptVal := v;
        send ("accept", b) to i;
    else  send ("reject", b) to i;
```

Decide phase

```
Upon receive ("accept",b) /("reject",b)   from > n/2 :
    if ("reject",b) is received then start over;
    send ("decide", v) to all

Upon receive ("decide", v) :
    decide v
```

Correctness: Agreement

- Follows from the following Lemma 1:

  If a leader sends (“decide”, v) after receiving (“accept”,b,v) from n-t processes, then v’=v for every proposal (*b′, v′*) with *b′>b.*

- Proved by induction, starting from the smallest balloted proposal accepted by a majority
  - Decided value v is sent by a leader after receiving (“accept”,b,v) from n-t processes
  - Proposal for b’ is selected by a leader only after a majority voted
  - These two majorities have a member in common that will repeat v in b
  - Ballot b being most recent v will be proposed by leader at ballot b’

Correctness: Termination

- Once there is one correct leader :
  - It eventually chooses the highest rank
  - No other process becomes a leader with a higher rank
  - All correct processes “ack” its prepare message, “accept” its propose message
  - The leader sends a “decide” message
  - All correct processes decide

Dramatic reenactment

Brought to you by MIT improv club

Rules of the MIT improv club

- 7 people
  - Everyone is his own messenger
  - Client requests are letters of alphabet

- Allowed mischiefs:
  - Lose / duplicate a message
  - Kill a member / resurrect a member in middle
  - Two simultaneous leaders
    - ➢ Kill the leader in middle, recover the leader later

Paxos dissected

Slides courtesy of Keith Marzullo

Paxos dissected

Follows the presentation in “Paxos made simple”

— Give a constructive argument for the consensus protocol

There are three basic roles that take place in consensus

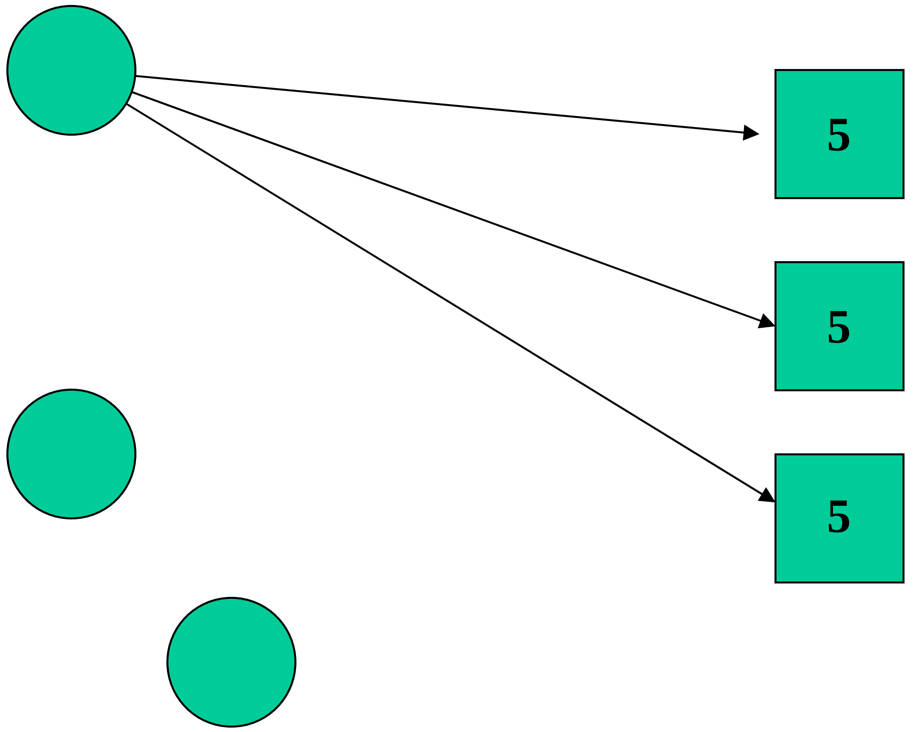
- *Proposers* that propose a value for consensus
- *Acceptors* that choose the consensus value
- *Learners* that learn the consensus value

A single process may take on multiple roles – we ignore this for now

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

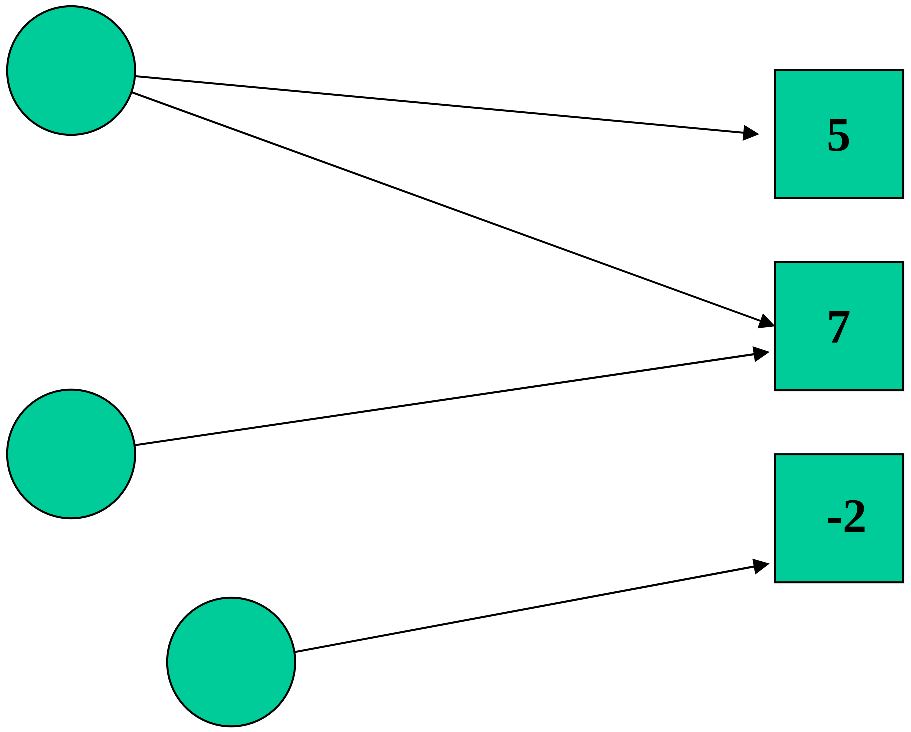
Choosing a value (II)

- In the absence of failures, we want a value to be chosen even if only one value is proposed by a single proposer. 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)

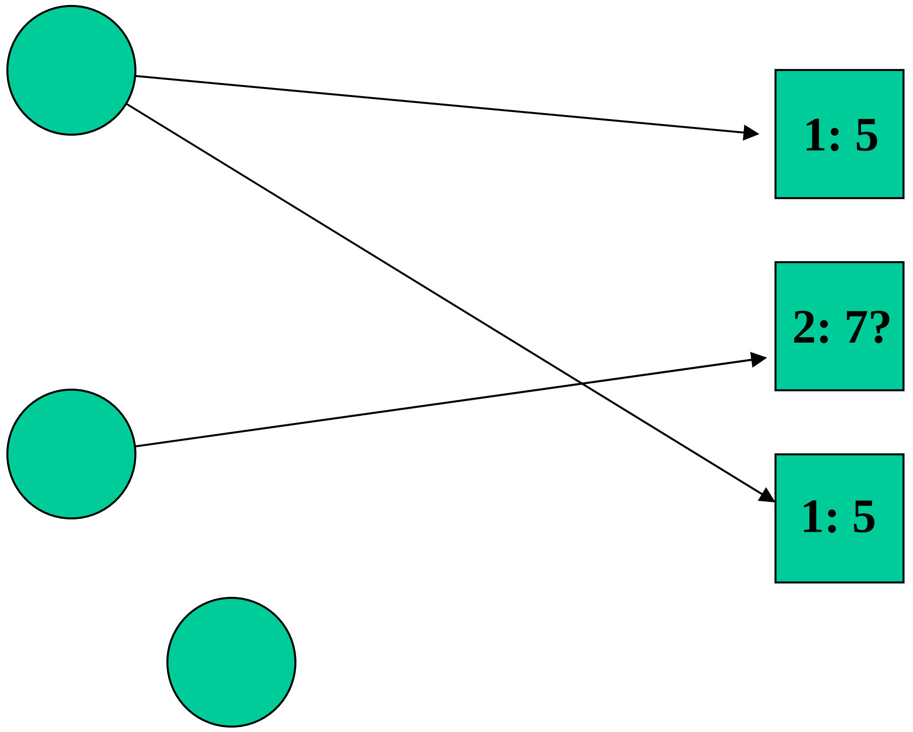
- We need to guarantee that all chosen proposals have the same value. It suffices to guarantee:

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)

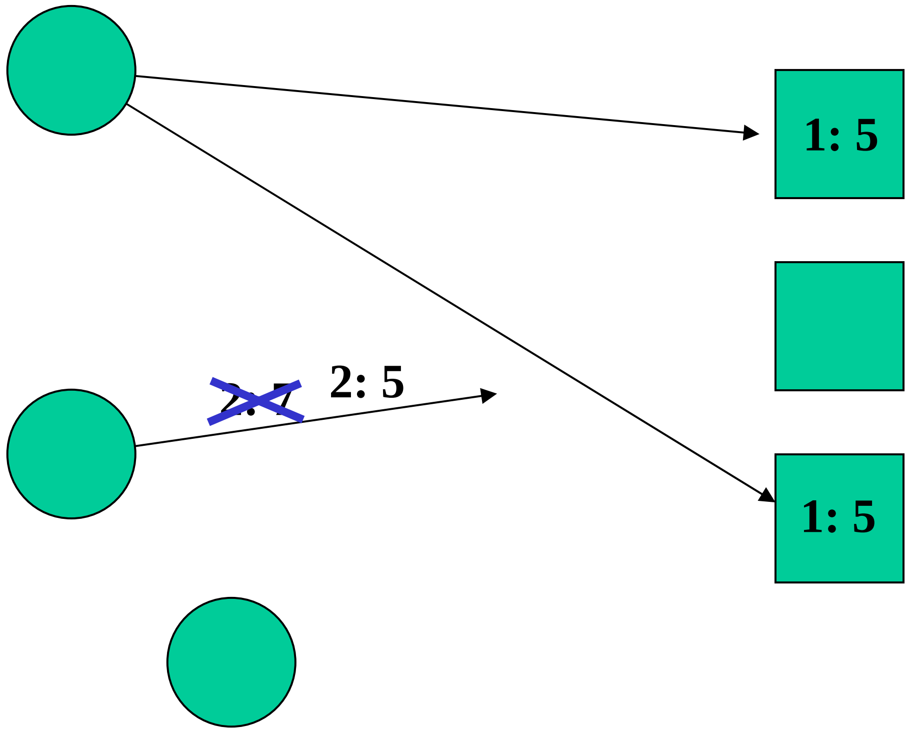
- P1 is still needed to ensure that *some* proposal is accepted.

... 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

- 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



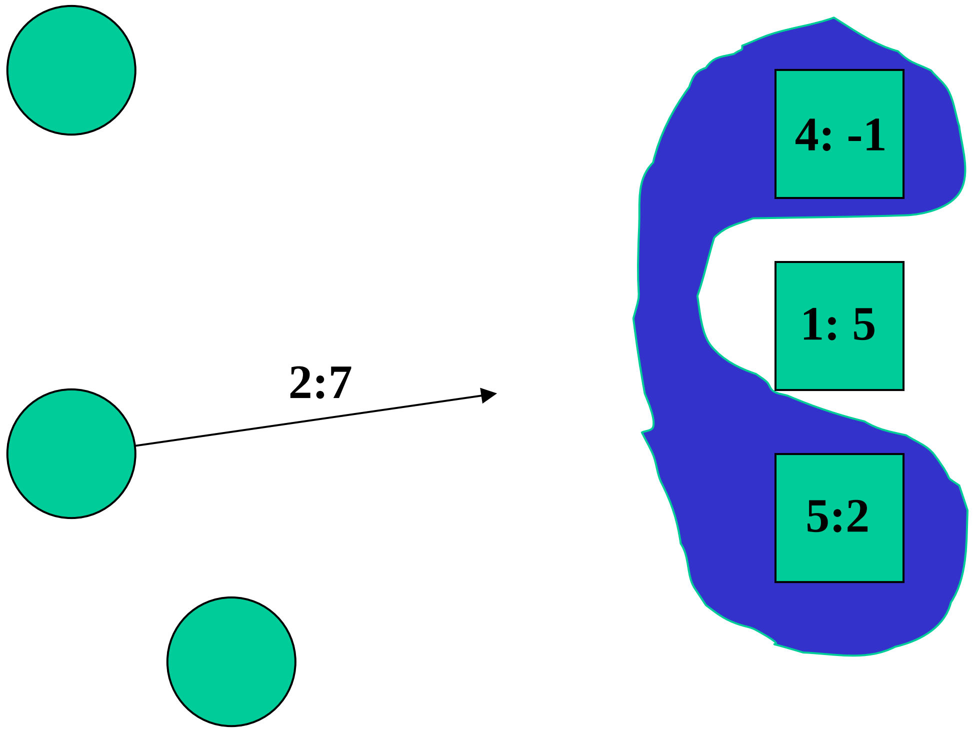
Choosing a value (VI)

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

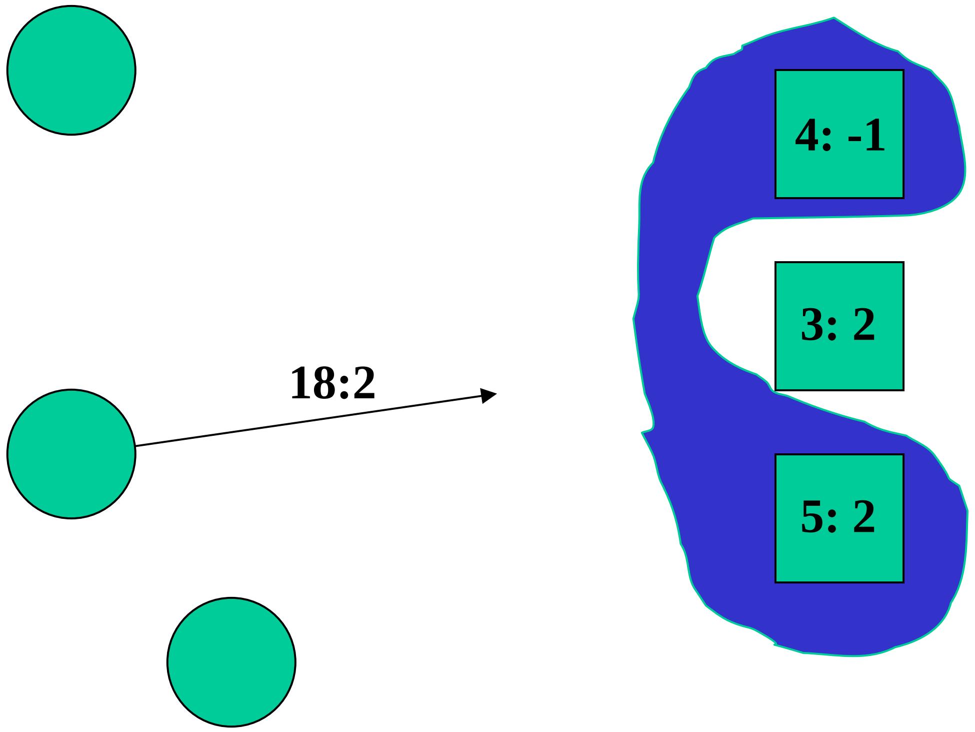
- no acceptor in *S* has accepted any proposal numbered less than *n*, or
- *v* is the value of the highest-numbered proposal among all proposals numbered less than *n* and accepted by the acceptors in *S*

Choosing a value



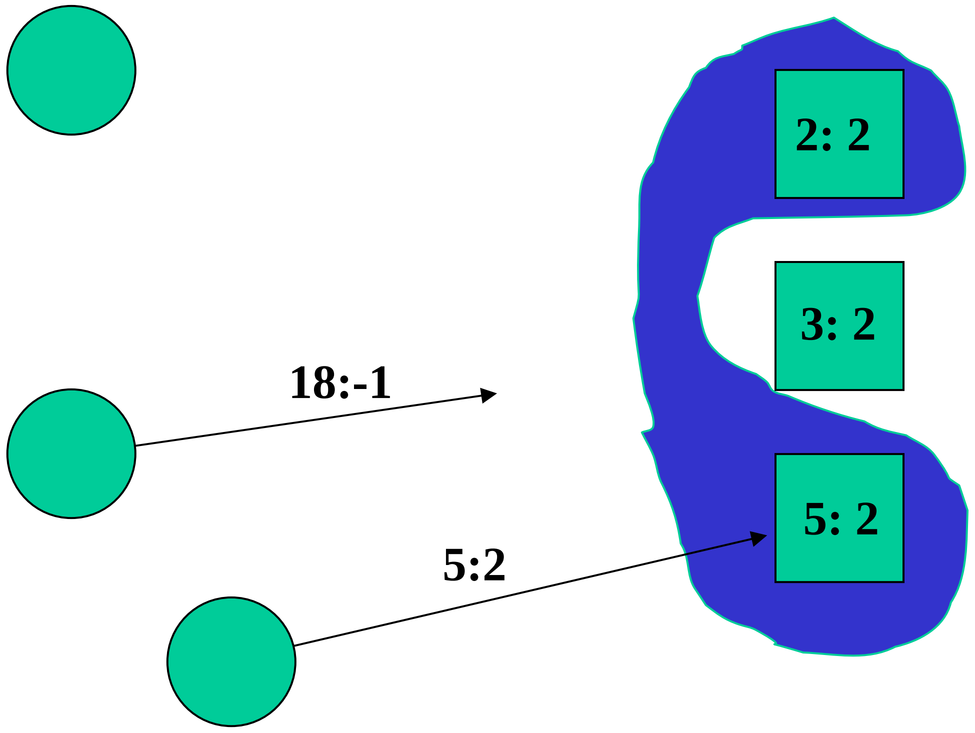
no acceptor in *S* has accepted any proposal numbered less than *n*.

Choosing a value



v is the value of the highest-numbered proposal among all proposals numbered less than *n* and 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 (VII)

- 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*

Choosing a value (VIII)

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 (IX)

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 (X)

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 (XI)

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.

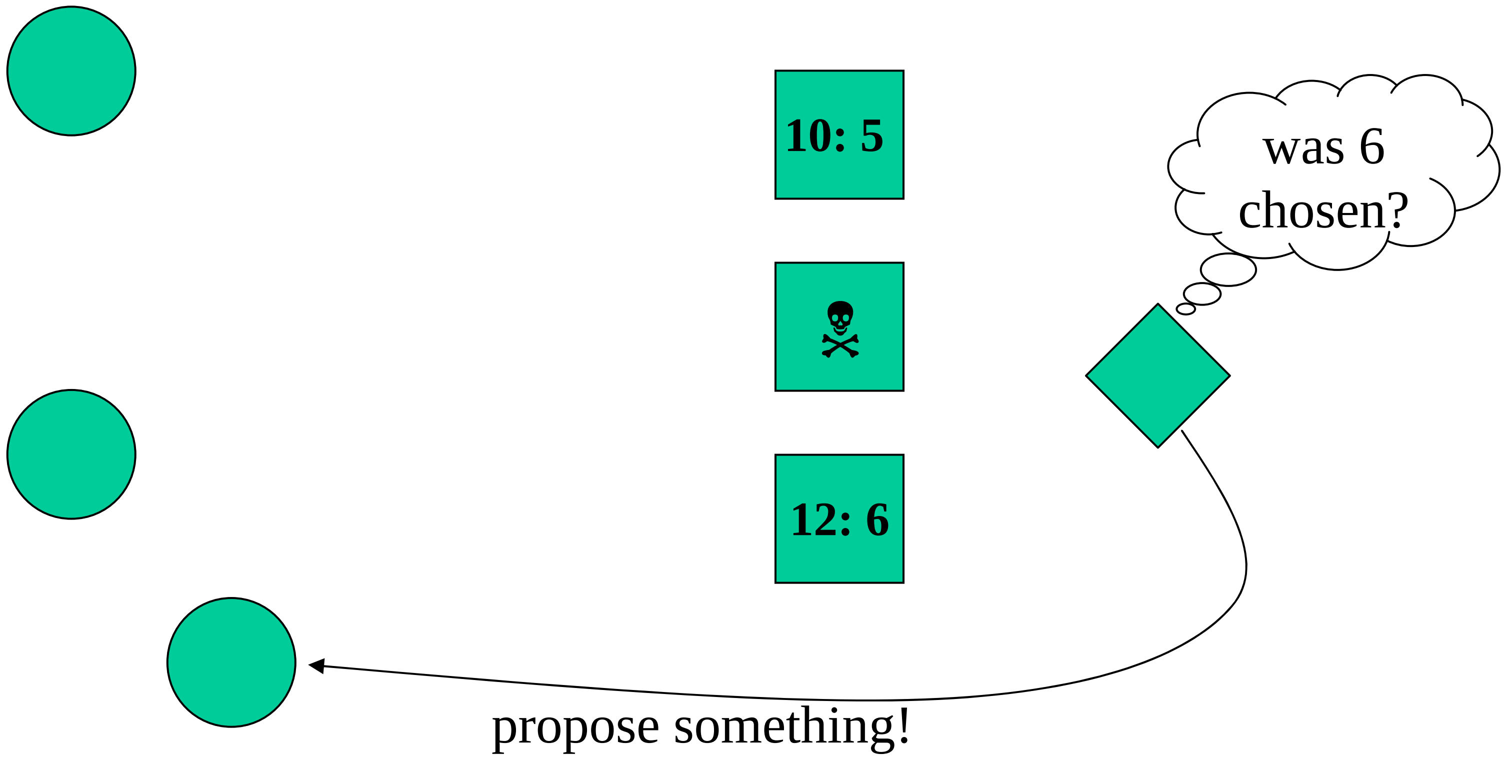
b) If an acceptor receives an *accept* request for a proposal numbered *n*, it accepts the proposal unless it has already responded to a *prepare* request having a number greater than *n*.

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.



Optimization: Fast Paxos

- Allow process 1 (only!) to skip the Prepare phase
  - initiate BallotNum to <1,1>
  - propose its own initial value

- Terminates in 3 communication steps
  - Can be further optimized by processes broadcasting their accept messages
  - Hence, achieve 2 communication steps complexity, which is optimal

Putting it into context

Slides courtesy of Keith Marzullo

Implementing state machines (I)

- We 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.
- 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 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. It 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, it 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.

Byzantine Paxos

Slides courtesy of Idith Keidar

Byzantine Paxos Setting

- Motivation: State machine replication

- Structured like Paxos:
  - Updates are sent to the current leader
  - Leader uses a consensus algorithm to have all replicas agree on the order of updates

- Used to implement BFS – Byzantine Fault Tolerant NFS
  - Only 3% slower than un-replicated NFS

Overcoming byzantine failures with 3t+1 processes

- For crash failures –
  - We gather “votes” from a majority in every ballot.
  - Since every two majorities intersect, for every two ballots, at least one process votes in both.

- But now, a faulty process can lie about what it did in the other ballot.
  - We want a *correct* process in the intersection.
  - Since $n\text{-}t \geq 2t+1$, two sets of size $n\text{-}t$ intersect by at least one correct process.
  - Gather $n\text{-}t$ votes in a ballot, to ensure that for every two ballots, at least one correct process votes in both.

Model

- Universe: n processes: {0,...n-1}.
- Up to t Byzantine failures, $t < n/3$.
  - Assume $n = 3t+1$
- Authentication
- Reliable links, no recovery (for now).

Safety problems: Leader can lie

- Leader can choose a value different than the highest accepted by *n-t* processes
  - Solution: Should "prove" he's not lying by sending the signed "ack" (phase 1) messages to *all* processes
- If no previous ballot was accepted, leader can send different new values to different processes
  - Solution: Before accepting a value proposed by the leader, verify that the value was proposed to "enough" processes
  - Phase 1: Prepare
  - Phase 2: Accept – echo leader's proposal
  - Phase 3: Decide – now only if *n-t* proposed

Safety problems: Others can lie

- Faulty users can send invalid values with higher AcceptNums in “ack” messages.
  - Solution: Should “prove” value is valid by forwarding signed “accept” messages.
  - Add new variable: Proof, initially empty set.

- Faulty users can send invalid “decide” messages.
  - Solution: Wait for $n\text{-}t=2t+1$ “decide” messages.

Liveness problems

- Faulty leader can deadlock algorithm.
  - Solution: Propose a new leader when the current does not deliver.
  - Use rotating coordinator until one is correct.
    Leader will be BallotNum mod n.

- Faulty processes may keep selecting new leaders all the time (livelock).
  - Solution: Accept a new ballot only if *t+1* processes propose a new leader.

Byzantine Paxos Phase I: Prepare

```
Until decision is reached, try:

if leader then
    BallotNum ← BallotNum +1
    send ("prepare", BallotNum) to all


Upon receive ("prepare", b) from t+1 :

      if (b < BallotNum) then return;
      if (b > BallotNum) then
          BallotNum ← b
          send ("prepare", BallotNum) to all
      send ("ack", b, AcceptNum, AcceptVal, Proof) to Leader
```

Byzantine Paxos Phase II: Accept

Upon receive ("ack", BallotNum, b, val, proof) from n-t :

```
    S = {received (signed) "ack" messages}
    if (all vals that have valid proofs in S are $\perp$) then myVal $\leftarrow$ init value
    else myVal $\leftarrow$ val that has valid proof with highest b in S
    send ("accept", BallotNum, myVal, S) to all
```

Upon receive ("accept", b, v, S) :

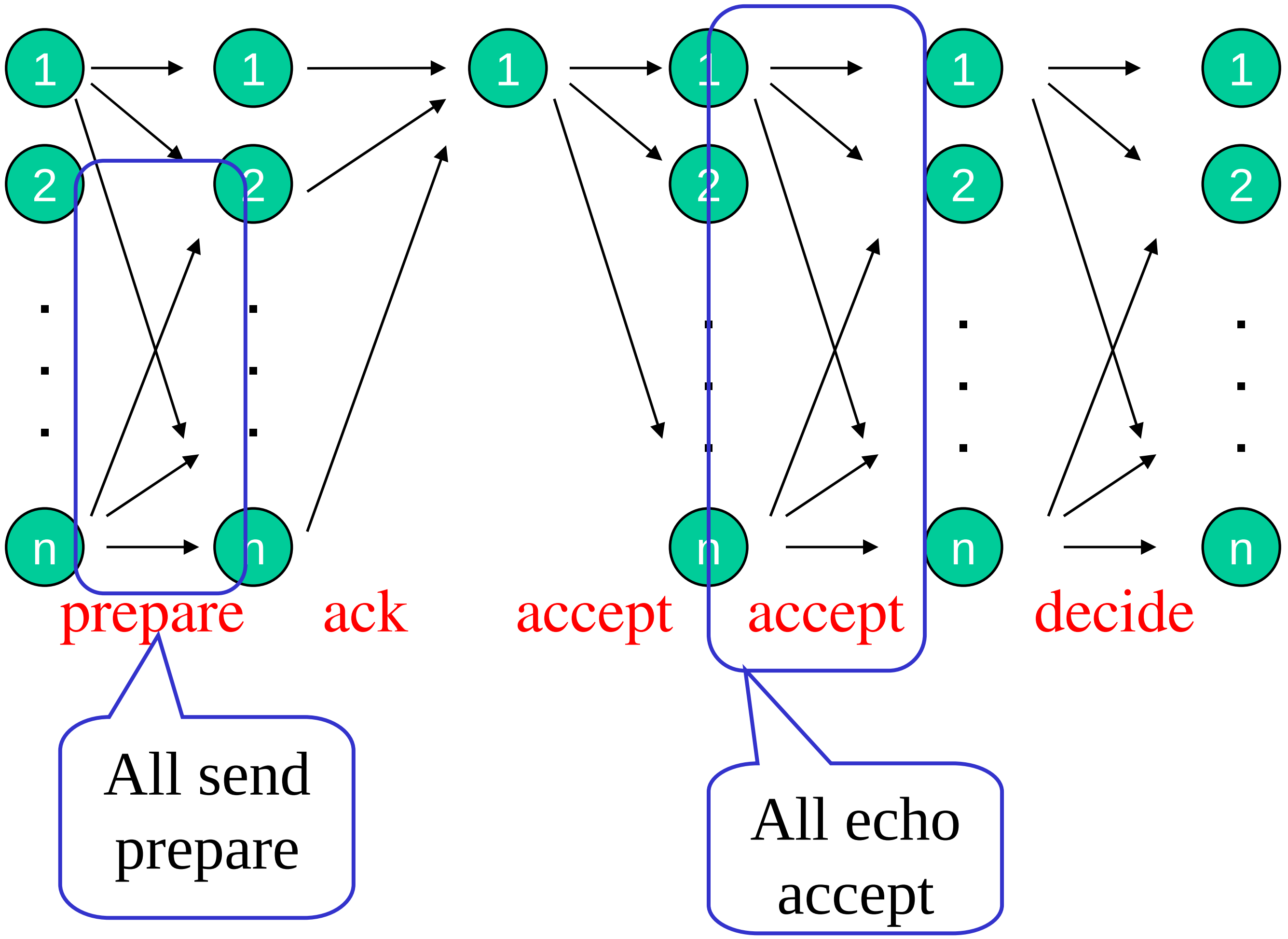
```
    if ( b $\leq$ BallotNum) then return;
    if (v is not a valid choice given S) then return;
    BallotNum $\leftarrow$ b;
    send ("accept", BallotNum, v, S) to all
```

Byzantine Paxos Phase III: Decide

```
Upon receive ("accept", b, v, S) from n-t
    if (b < BallotNum) then return
    AcceptNum ← b; AcceptVal ← v
    Proof ← set of n-t signed "propose" messages
    send ("decide", b, v) to all

Upon receive ("decide", b, v) from n-t
    decide v
```

In Failure-Free Runs



Saving Communication

- Prepare and its “ack” can be merged into one message round.
- Proofs don’t have to be sent with messages: processes can have the information to check the proofs locally because the original messages are multicast.

Invariant

- If proposals *(b,v)* and (*b, v'*) are accepted by correct processes *i* and *j,* (possibly *i = j*) then *v'=v.*

- *Proof:*
  - An accepted proposal is proposed by *n-t* processes.
  - Two sets of *n-t = 2t+1* processes have at least one correct process in common.
  - A correct process sends no more than one propose message with the same b.

Lemma 1

- If a proposal *(b,v)* is accepted by *t+1* correct processes, then for every proposal (*b′, v′*) *that is proposed by a correct process* with *b′>b, v′=v.*

- Follows from invariant…
  - Since two sets of *t+1* correct processes have at least one correct process in common.

Lemma 2

- If a proposal *(b,v)* *is proposed by a correct process*, then there is a set *S* including at least *t+1* correct processes such that either
  - no correct *p* in *S* accepts a proposal ranked less than *b;* or
  - *v* is the value of the highest-ranked proposal among proposals ranked less than *b* accepted by correct processes in *S.*

Liveness

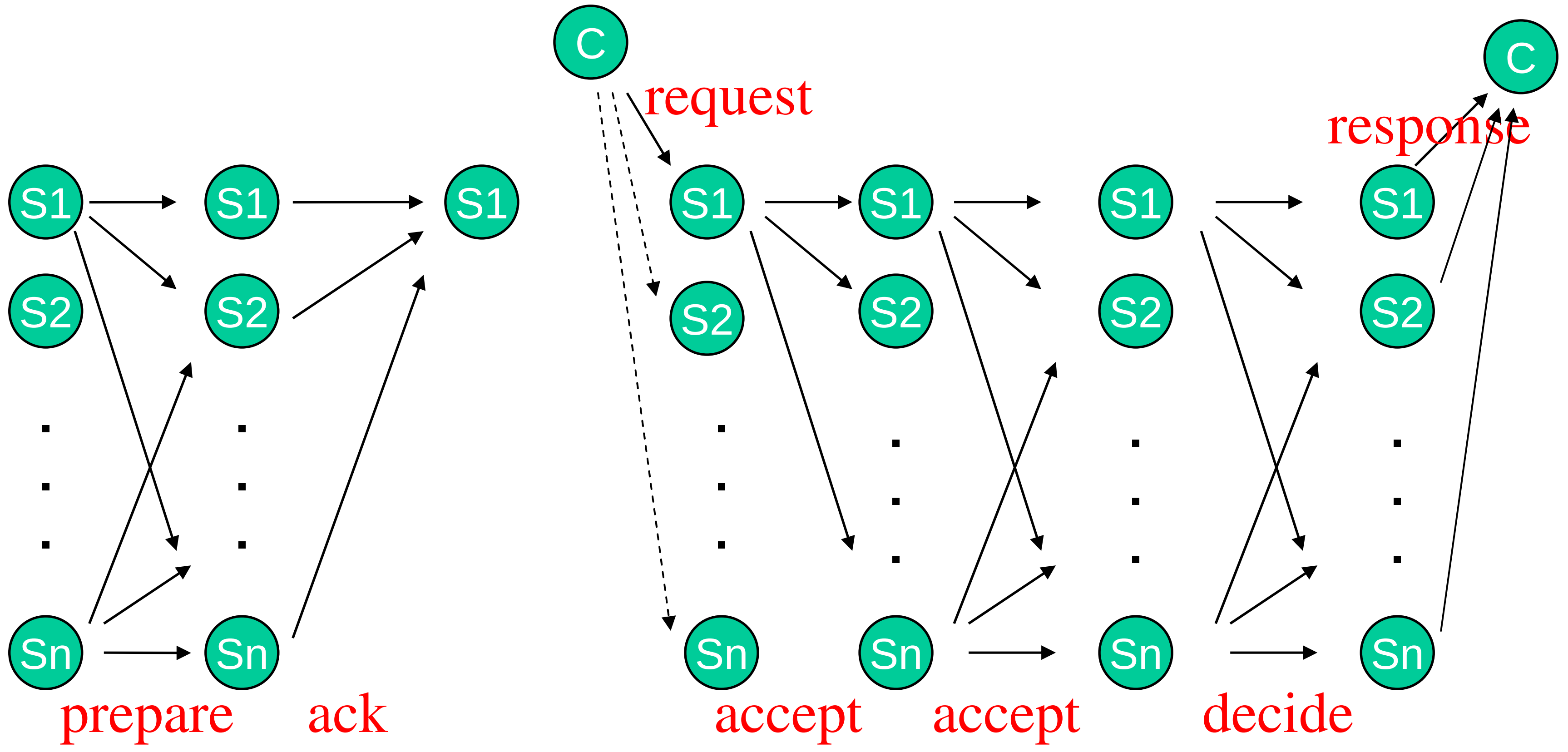
- Is the current leader making progress?
  - If yes, some correct process decides. This process can periodically forward the “proof” for its decision to others so they will decide too.
  - If not, all timeout on the leader and start a new ballot.

- Once there is a correct leader
  - The *n*-*t* correct processes will send all the needed messages
  - The *t* faulty processes will not be able to force a new ballot

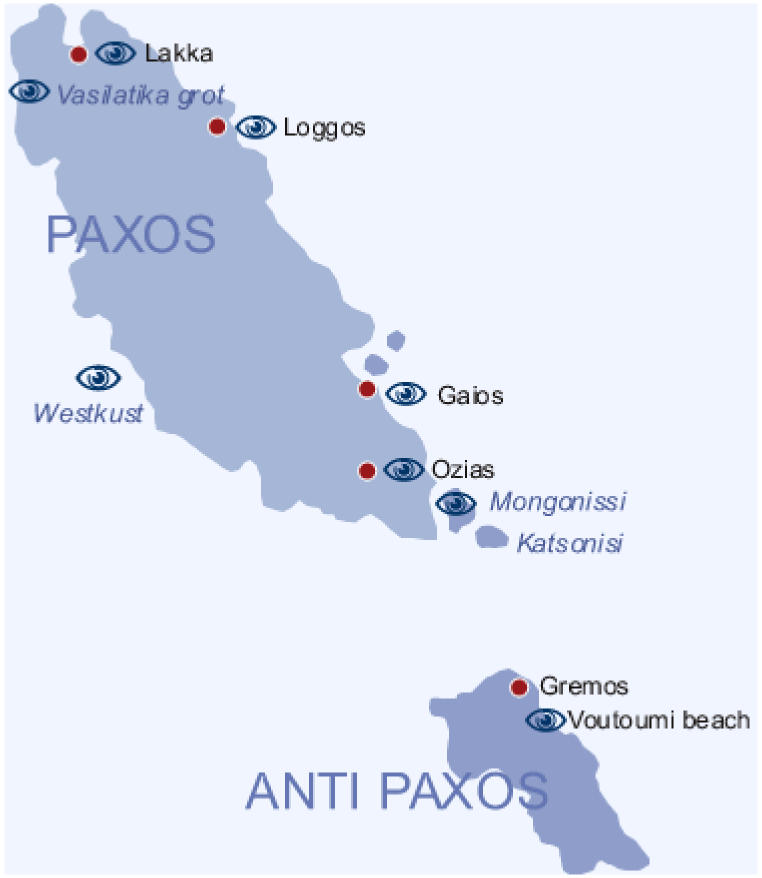
Atomic Broadcast: Issues

- Leader can propose invalid client requests.
- Leader can refrain from proposing client requests.
- Leader can lie to client about response.
- Leader can refrain from sending client responses.
- Solution: clients cannot trust a single server.

Byzantine Message Flow



End of tour


Lakka
Vasilatika grot
Loggos
PAXOS
Westkust
Gaios
Ozias
Mongonissi
Katsonisi
Gremos
Voutoumi beach
ANTI PAXOS