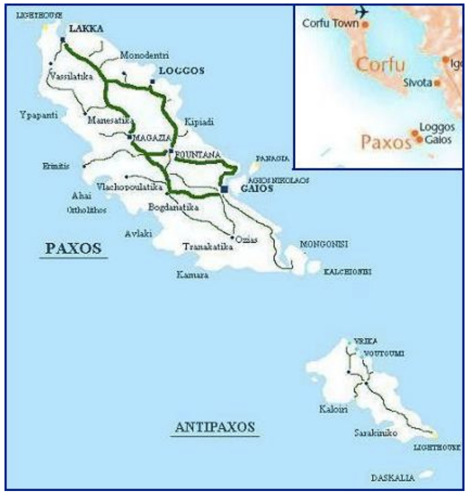
Replicated State Machines and Paxos



SLR210, P4, 2019

Quiz 1

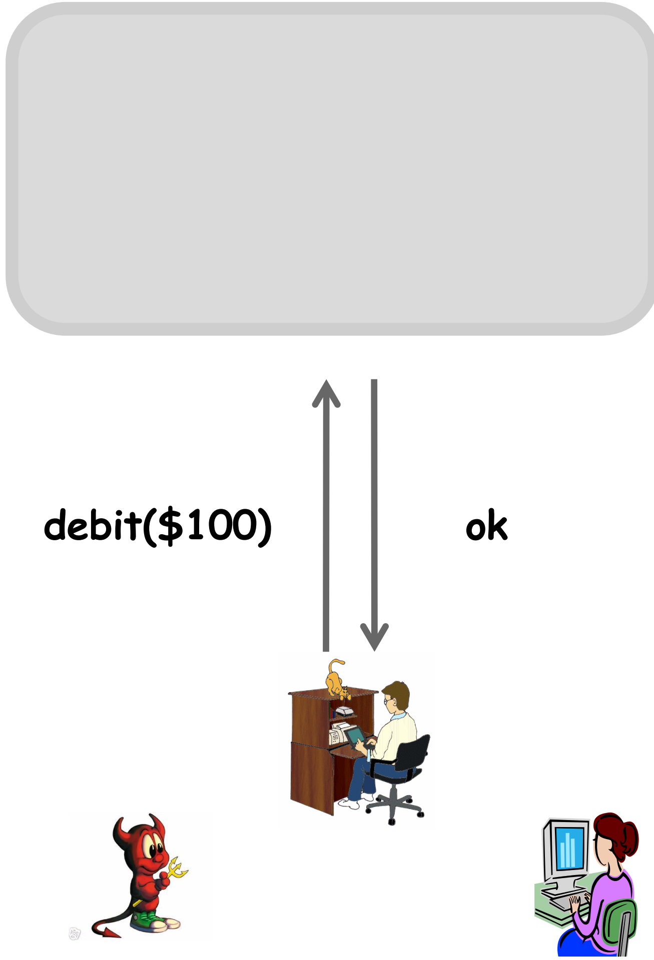
- Argue that the GLA algorithm (prev. lecture) is live
- Show that a set in which updates return boolean responses (depending on the operation’s success) cannot be wait-free implemented from GLA

How to build a consistent and reliable system?

Service accepts requests from *clients* and returns responses

- Liveness: every persistent client receives a response
- Safety: responses constitute a total order w.r.t. the service's *sequential specification*

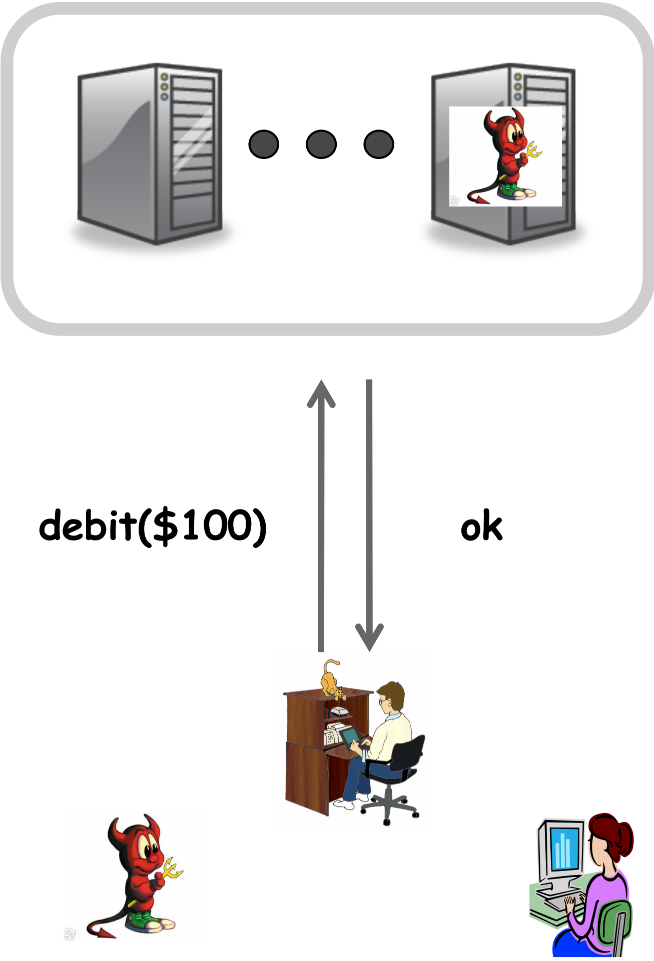
(recall universal construction)



How to build a fault-tolerant system?

Replication:

- Service = collection of *servers*
- Some servers may *fail*



“CAP theorem” [Brewer 2000]

No system can combine:

- Consistency: all servers observe the same evolution of the system state
- Availability: every client’s request is eventually served
- Partition-tolerance: the system operates despite a partial failure or loss of communication

Strongly consistent replicated state machine

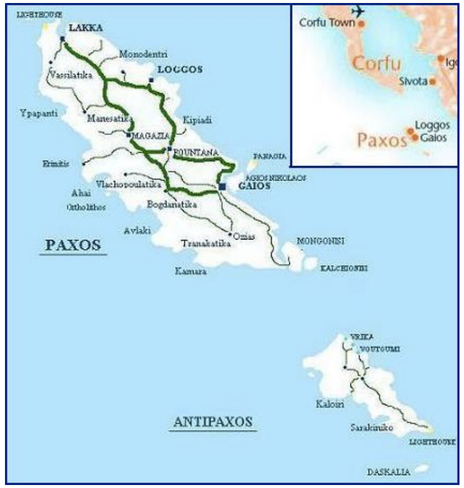
Universal construction in message-passing:

- Clients access the service via a standard interface
- Servers run replicas of the (sequential) service
- (A subset of) faulty servers do not affect consistency and availability

Leslie Lamport: The Part-Time Parliament. ACM Trans. Comput. Syst. 16(2): 133-169 (1998)

Paxos: some history

- Late 80s: a three-phase consensus algorithm
  - ✓A Greek parliament reaching agreement
- 1989: a Paxos-based fault-tolerant distributed database
- 1990: rejected from TOCS




```
"All three referees said that the
 paper was mildly interesting, though
 not very important, but that all the
 Paxos stuff had to be removed."
```

This submission was recently discovered behind a filing cabinet in the TOCS editorial office. Despite its age, the editor-in-chief felt that it was worth publishing. Because the author is currently doing field work in the Greek isles and cannot be reached, I was asked to prepare it for publication.

The author appears to be an archeologist with only a passing interest in computer science. This is unfortunate; even though the obscure ancient Paxon civilization he describes is of little interest to most computer scientists, its legislative system is an excellent model for how to implement a distributed computer system in an asynchronous environment.

…

Keith Marzullo
University of California, San Diego
(preface for the TOCS 1998 paper)

Paxos today

- Underlies a large number of practical system when strong consistency is needed
  - ✓Google Megastore, Google Spanner
  - ✓Yahoo Zookeeper
  - ✓Microsoft Azure
  - ✓….
- ACM SIGOPS Hall of Fame Award in 2012
- Turing award 2019

Consensus: recall the definition

A process *proposes* an *input* value in V (|V|≥2) and tries to *decide* on an *output* value in V

- *Agreement:* No two process decide on different values
- *Validity:* Every decided value is a proposed value
- *Termination:* No process takes infinitely many steps without deciding

  (Every correct process decides)

Cannot be solved in an asynchronous read-write shared-memory system with at least one faulty process

(extends to 1-resilient message-passing systems)

Circumventing impossibility: commit-adopt

A variant of consensus with weaker safety (relaxed agreement)

Can be used for solving consensus with an oracle

A process p_i *proposes* an *input* value in V ($|V| \geq 2$) and *decides* on a tuple (c,v) where c is a boolean and v is in V

- ✓We say p_i *adopts* v
- ✓If c=true, we say p_i *commits* on v

Commit-adopt: properties

- *Validity:* Every adopted value is an input value of some process
- *Termination:* Every correct process decides
- *CA-Agreement:*
  - ✓If a process commits on a value v, then no process can adopt a value v'≠v
  - ✓If all inputs are the same, then no process decides on (false,*)

    (every process that decides commits on a value)

Commit-adopt : protocol

Shared objects:

N atomic registers A[0,…,N-1], initially T
N atomic registers B[0,…,N-1], initially T

Upon propose(v) by process p_i:

```
v_i :=  v
A[i] := v_i
V := read A[0,…,N-1]
if all non-T values in V are v then
        B[i] := (true,vi)
else
        B[i] := (false,vi)
V := read B[0,…,N-1]
if all non-T values in V are (true,*) then
        return (true,v_i)
else if V contains (true,v) then
        v_i := v
return (false,v_i)
```

Commit-adopt: proof

Validity and Termination: immediate

CA-Agreement:

Claim 1 B[0,…,N-1] never contains (true,v) and (true,v') where v≠v'

Suppose not: p_i wrote (true,v) in B[i] and pj wrote (true,v') in B[j], v≠v'

Previously, p_i wrote v in A[i] and pj wrote v' in A[j] (let pi be the first to write)

But p_j should have seen A[i] ≠v' - a contradiction!

Commit-adopt: proof (contd.)

Claim 2 If p_i returns (true,v) then no process pj returns (c,v') where v≠v'

Suppose not: let p_j return (c,v') where v≠v'.

By Claim 1, p_j has previously written some (false,v'') in B[j]

Since p_j hasn't adopted v, it hasn't found (true,v) in B[1,…,N]

But then p_i should have read (false,v'') in B[j] – a contradiction!

Commit-adopt: proof (contd.)

Claim 3 If all inputs are the same then no process returns (false,*)

Immediate: both "if" conditions are true, i.e., the non-T values in A and B are the same

Ω: an oracle

- Eventual leader failure detector
- Produces (at every process) events:
  - ✓‹Ω, leader, p›
  - ✓We also write p=leader()
- Eventually, all correct processes output the same correct process as the leader

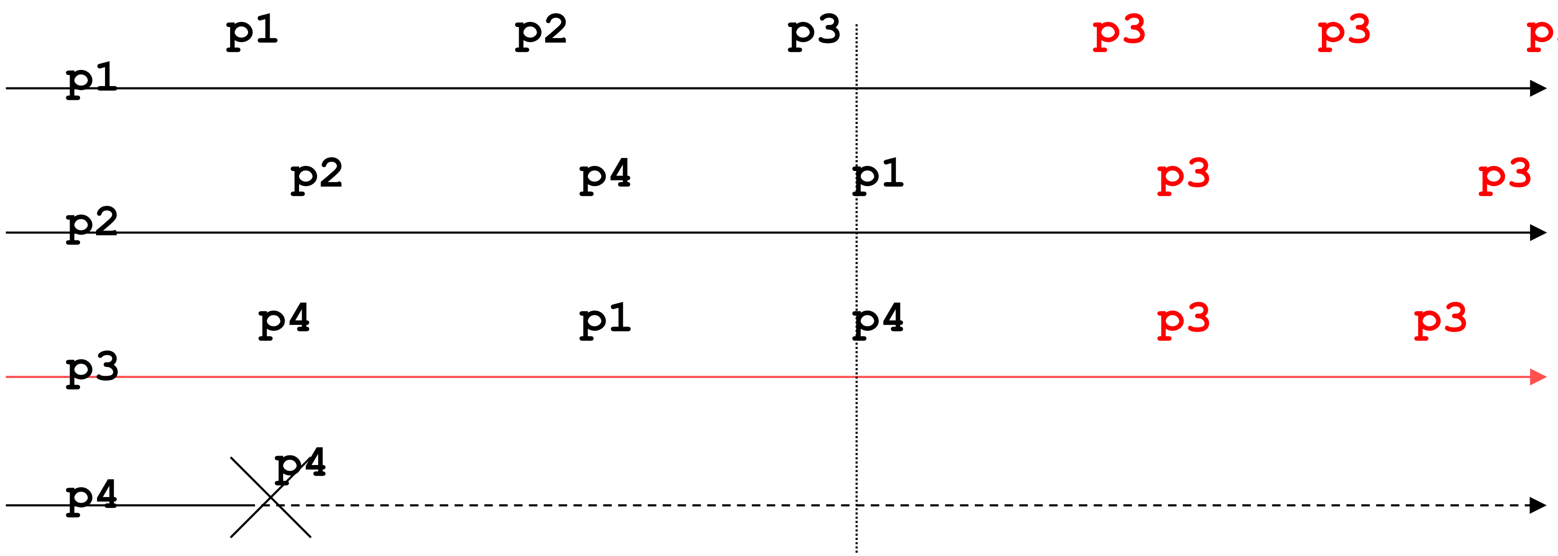
Can be implemented in eventually synchronous system:

- ✓There is a bound on communication delays and processing that holds only eventually
- ✓There is an a priori unknown bound in every run

Leader election Ω: example

There is a time after which the same correct process is considered leader by everyone.

(Sufficient to output a binary flag leader/not leader)



Consensus = Ω + CA

Shared:

D[1,…,∞], regular registers, initially T

CA_1, CA_2,… a series of commit-adopt instances

Upon propose(v) by process p_i:

```
vi := v
r := 0
repeat forever
    r++
    (c,vi):=CAr(vi)                       // r-th instance of commit-adopt
    if c=true then
            D[r]:=vi                      // let the others learn your value
            return vi
    repeat
            if Ω outputs pi then
                D[r]:= vi     // advertise your value if leader
    until D[r]=v' where v'≠T                 //wait until the leader writes its value
    vi := v'                              //adopt the leader's value
```

Quiz 2: commit-adopt

- Would the CA algorithm is correct if regular registers were used?
- Show that Ω + CA indeed solve consensus
- Give an obstruction-free consensus algorithm using CA
  - ✓Obstruction-freedom: every process that runs solo from some point on eventually decides

Back to message-passing

- Asynchronous system
- Reliable communication channels
- Processes fail by crashing
- A majority of correct processes

But we proved that 1-resilient consensus is impossible even with shared memory!

“CAP theorem” is violated!

Where is the trick?

Paxos/Synod algorithm

- Let's try to decouple liveness (termination) from safety (agreement)

- Synod made out of two components:
  - ✓Ω - the eventual leader oracle
  - ✓(ofcons) obstruction-free consensus

Obstruction-free Consensus (ofcons)

- Similar to consensus
  - ✓except for Termination
  - ✓ability to abort and propose again

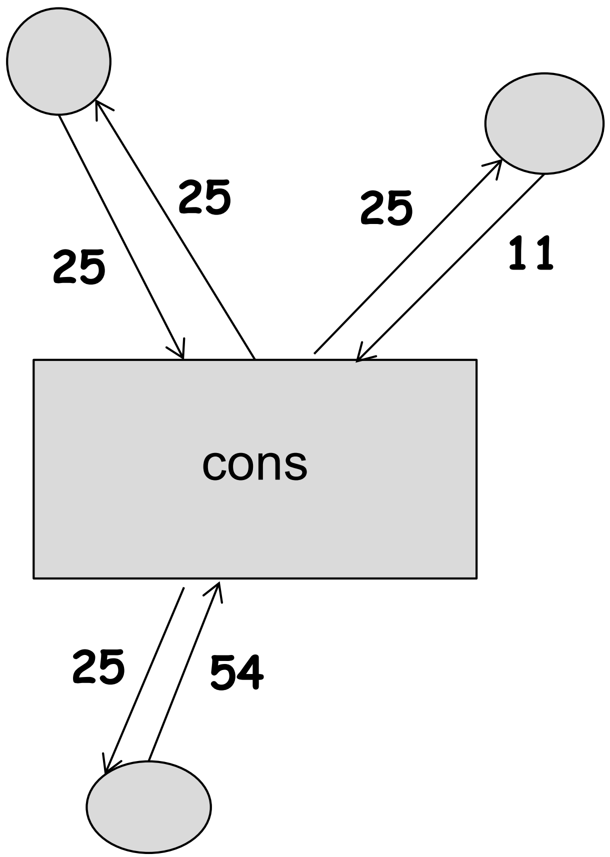
- Requests:
  - ✓‹ofcons, propose, v› (propose v)
- Responses:
  - ✓‹ofcons,decide, v’› (decide v’)
  - ✓‹ofcons,abort› (aborts)

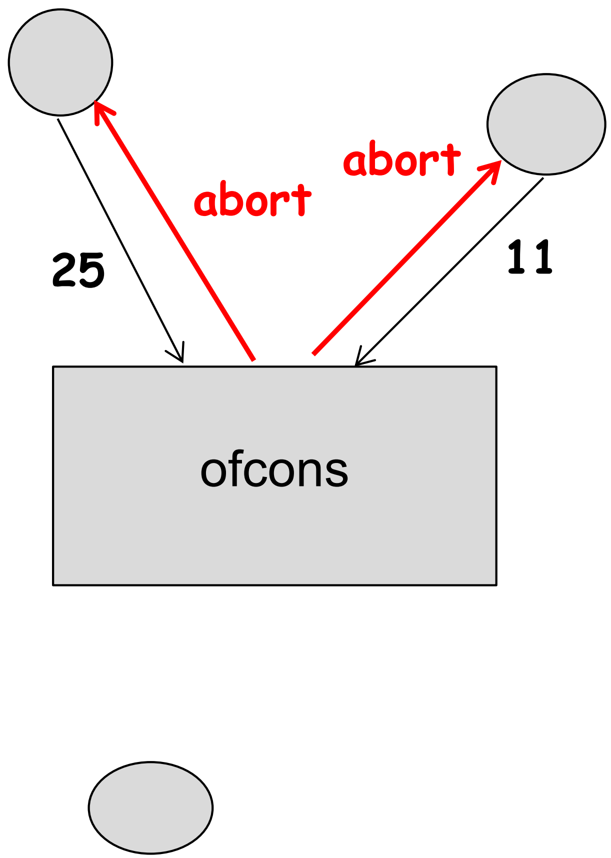
23

Obstruction-free Consensus

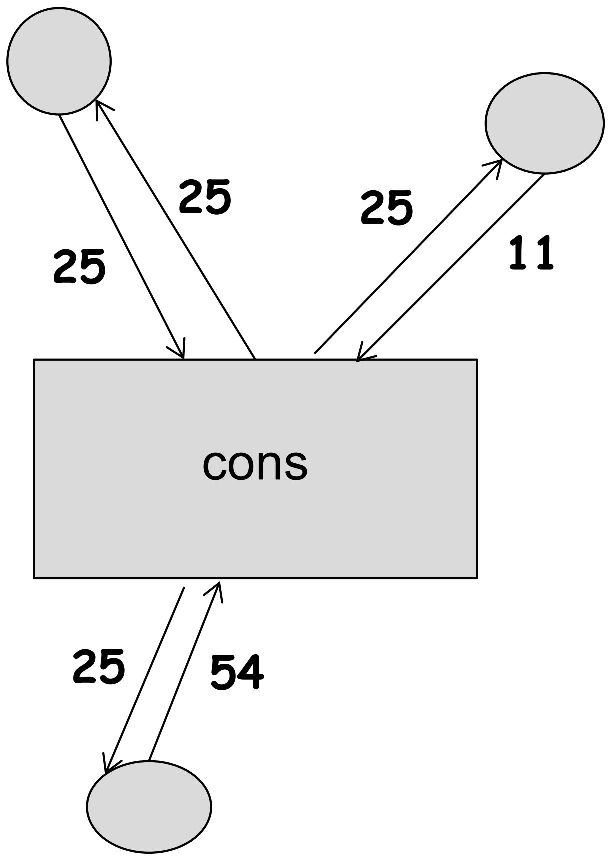
- *C1. Validity*:
  - ✓Any value decided is a value proposed
- *C2. Agreement:*
  - ✓No two correct processes decide differently
- *C3. Obstruction-Free Termination:*
  - ✓If a correct process p proposes, it eventually decides or aborts.
  - ✓If a correct process decides, no correct process aborts infinitely often.
  - ✓If there is a time after which a single correct process p proposes a value sufficiently many times, p eventually decides.

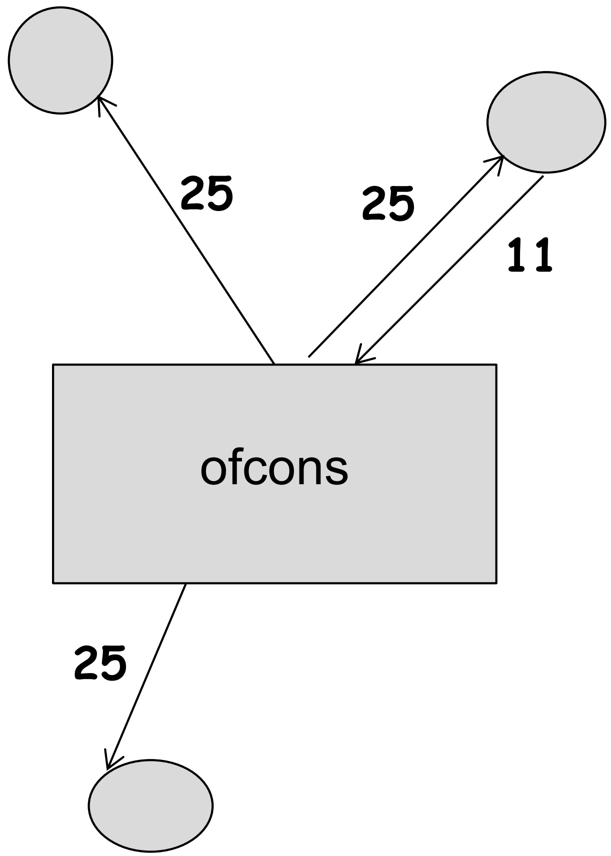
Consensus vs. OF-Consensus





Consensus vs. OF-Consensus





Consensus using Ω and ofcons

- Straightforward
  - ✓Assume that in cons everybody proposes

```
upon ‹cons, propose, v›
  while not(decided)
      if   self=leader() then
       result = ofcons.propose(v)
       if  result=(decide,v’) then
           return v’
```

Link to Paxos/Synod

- External cons.propose events come in a state machine replication algorithm as requests from clients
  - ✓As in universal construction

- Focus now on implementing OFCons

OFCons

- Not subject to FLP impossibility!
- Can be implemented in fully asynchronous system
  - ✓Using the correct-majority assumption
  - ✓Or read-write
- Synod OFCons: a 2-phase algorithm

Synod OFCons I

```
Code of every process pi:

Initially:
   ballot:=i-n; proposal:=nil; readballot:=0; imposeballot:=i-n;
   estimate:= nil; states:=[nil,0]^n

upon ‹ofcons, propose, v›
   proposal := v; ballot:=ballot + n; states:=[nil,0]^n
   send [READ, ballot] to all

upon receive [READ,ballot’] from pj
   if readballot ≥ ballot’ or  imposeballot ≥ ballot’ then
          send [ABORT, ballot’] to pj
   else
          readballot:=ballot’
          send [GATHER, ballot’, imposeballot, estimate] to pj

upon receive [ABORT, ballot] from some process
   return abort
```

Synod OFCons II

```
upon receive [GATHER, ballot, estballot, est] from pj
  states[pj]:=[est,estballot]

upon  #states ≥ majority //collected a majority of responses
  if ∃ states[pk]=[est,estballot] with estballot>0  then
     select states[pk]=(est,estballot) with highest
  estballot
     proposal:=est;
  states:=[nil,0]^n
  send [IMPOSE, ballot, proposal] to all

upon receive [IMPOSE,ballot',v] from p_j
   if readballot > ballot' or  imposeballot > ballot' then
          send [ABORT, ballot'] to p_j
  else
         estimate := v; imposeballot:=ballot'
         send [ACK, ballot'] to p_j
```

Synod OFCons III

```
upon received [ACK, ballot] from majority
  send [DECIDE, proposal] to all

upon receive [DECIDE, v]
   send [DECIDE, v] to all
   return [decide, v]
```

Correctness

- Validity
  - ✓Immediate
- Agreement
  - ✓When is the decided value determined?
- OF Termination
  - ✓Show that a correct process that proposes either decides or aborts
  - ✓If a single process keeps proposing?

Time Complexity

- Fault-free time complexity: 4 message delays

  + 1 communication step for decision relaibible broadcast
- Optimizations
  - ✓Getting rid of the first READ phase
- Allow a single process (presumed leader, say p1) to skip the READ phase in its 1st ballot
  - ✓Reduces fault-free/sync time complexity to 2

34

From Synod to Paxos

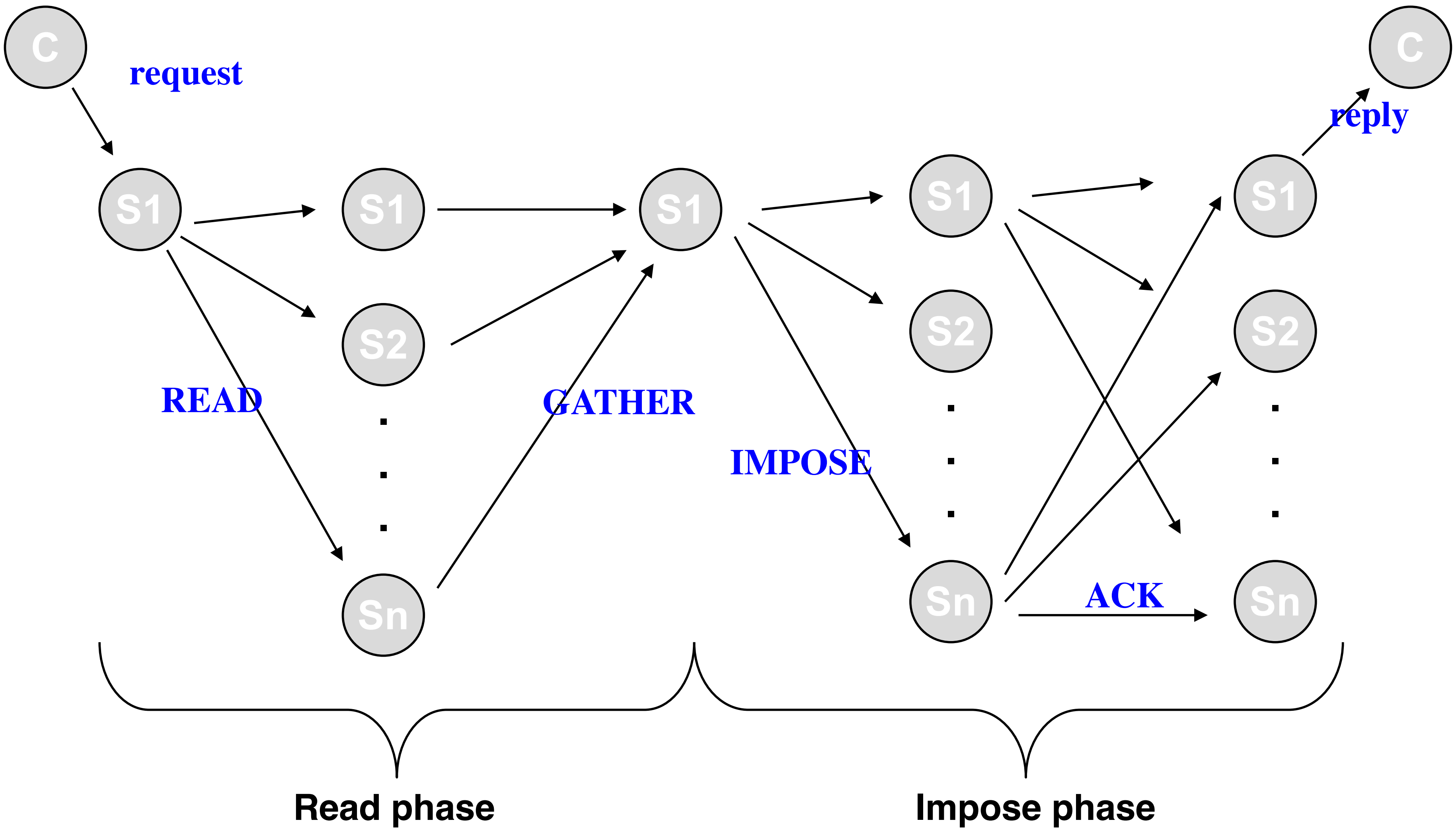
- Paxos is a state-machine replication (SMR) protocol
  - ✓i.e., a universal construction given a sequential object

- Implemented as totally-ordered broadcast: exports one operation toBroadcast(m) and issues toDeliver(m') notifications

Paxos SMR

- Clients initiate requests
- Servers run consensus
  - ✓Multiple instances of consensus (Synod)
  - ✓Synod instance 25 used to agree on the 25th request to be ordered
- Both clients and servers have the (unreliable) estimate of the current leader (some server)
- Clients send requests to the leader
- The leader replies to the client

Paxos failure-free/sync message flow



Observation

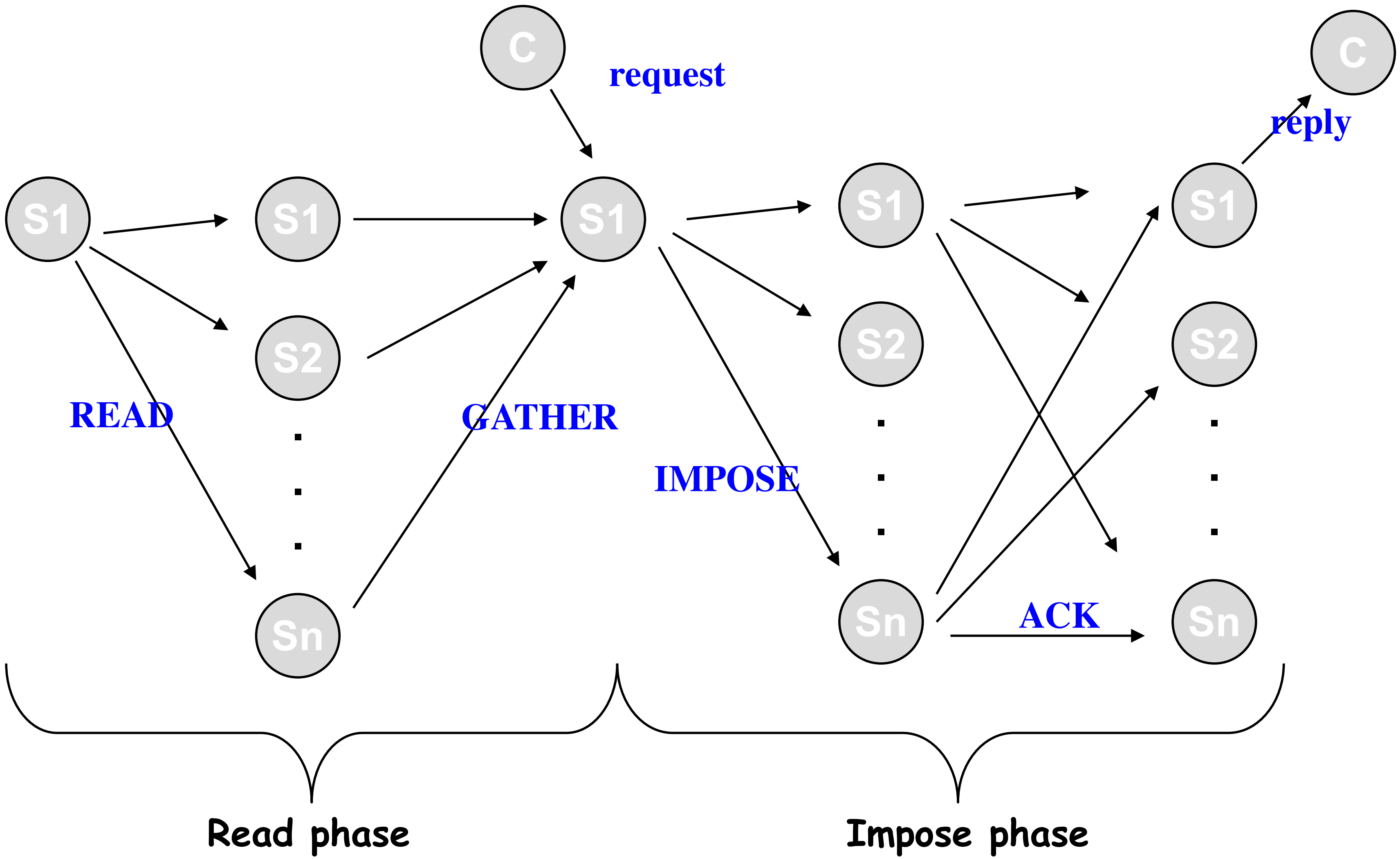
- READ phase involves no updates/new consensus proposals
  - ✓Makes the leader catch up with what happened before
- Most of the time the leader will remain the same
  - ✓+ nothing happened before (e.g., new requests)

38

Optimization

- Run READ phase only when the leader changes
  - ✓and for multiple Synod instances simultaneously
- Use the same ballot number for all future Synod instances
  - ✓run only IMPOSE phases in future instances
  - ✓Each message includes ballot number (from the last READ phase) and ReqNum, e.g., ReqNum = 11 when we're trying to agree what the 11^{th} operation should be
- When a process increments a ballot number it also READs
  - ✓e.g., when leader changes

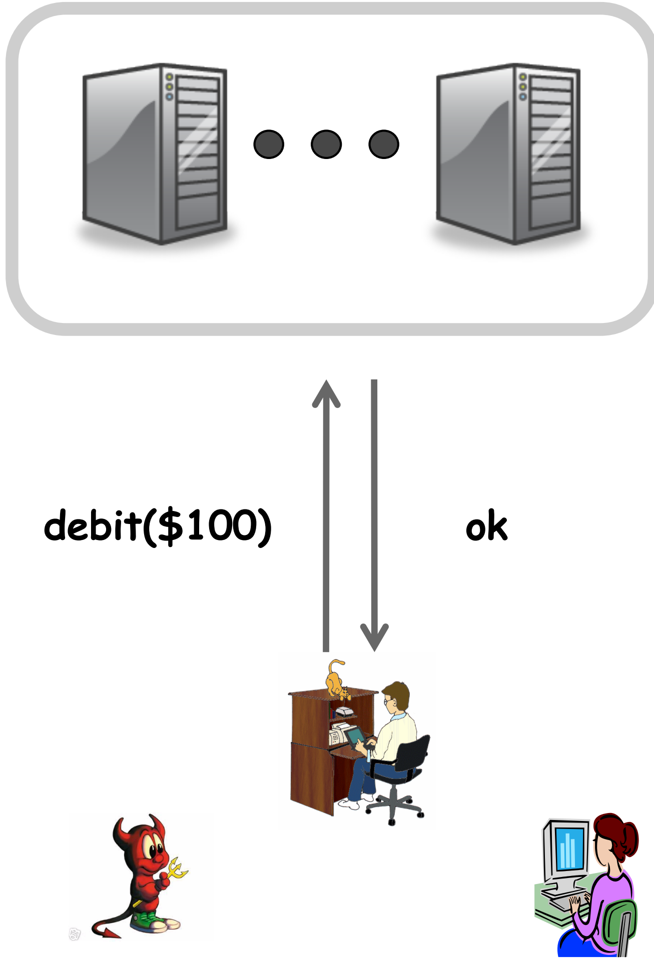
Paxos Failure-Free Message Flow



Paxos:
(universal) state machine replication

Replicated service:

- Collection of *servers*
- Some servers may *fail*



- Liveness: every persistent client receives a response
- Safety: responses constitute a total order w.r.t. the service's *sequential specification*

(linearizability)

Atomic broadcast: Abstract ordering service

Interface:

- call *broadcast(m)*
- callback *deliver(m)*

Properties:

- Validity: if a correct process invokes *broadcast(m)*, then eventually every correct process executes *deliver(m)*
- No duplication: for a given m, a process executes deliver(m) at most once
- No creation: if a process executes delivers(m), then some process previously executed broadcast(m)
- Total order: if a process delivers m and then m’, then no process delivers m’ before m

Quiz 3

- Prove that Synod satisfies Agreement and OF-termination
- Show that Atomic Broadcast is equivalent to State Machine Replication (SMR):
  - ✓Any SMR algorithm can be used to implement Atomic Broadcast
  - ✓Any Atomic Broadcast algorithm can be used to implement SMR