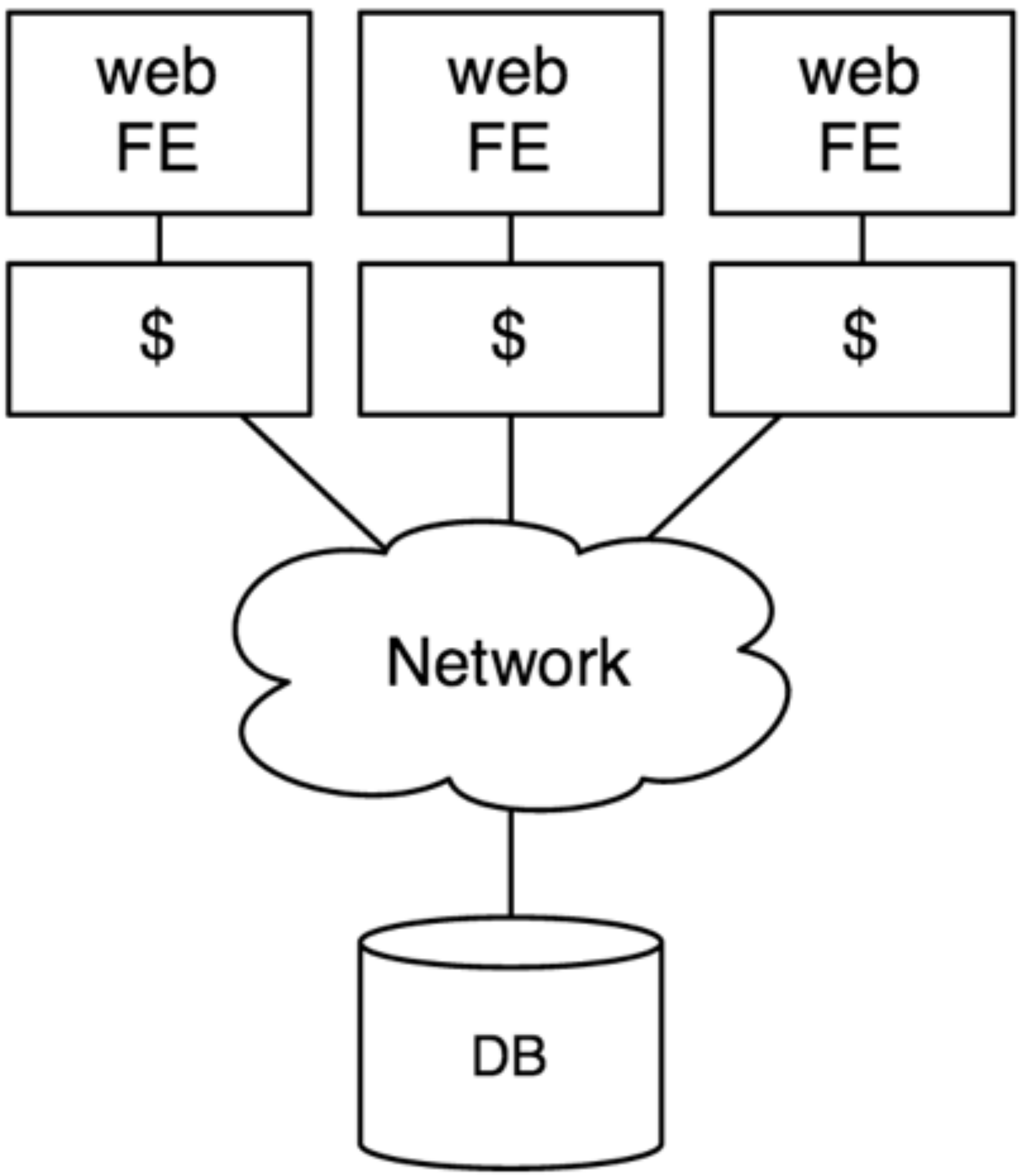
Paxos and Replication

Dan Ports, CSEP 552

Today: achieving consensus with Paxos

and how to use this to build a replicated system

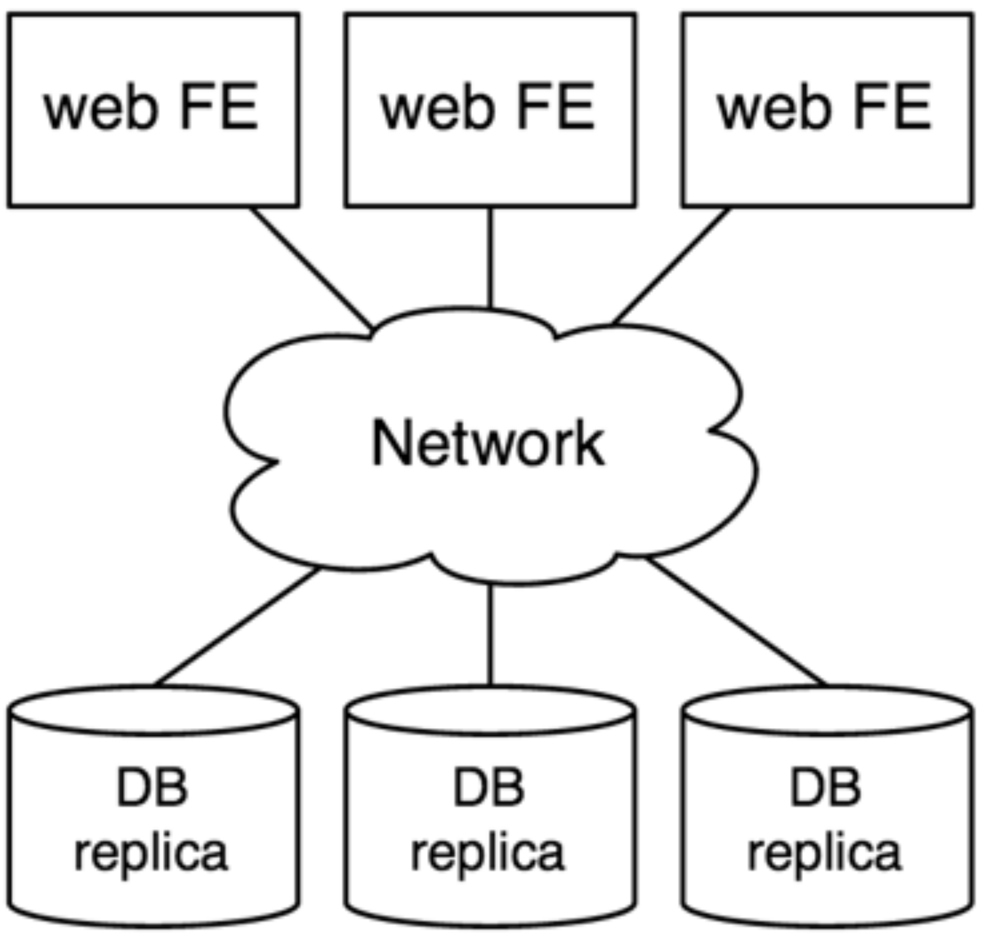
Last week



Scaling a web service using front-end caching

…but what about the database?

Instead:



How do we replicate the database?

How do we make sure that all replicas have the same state?

…even when some replicas aren't available?

Two weeks ago (and ongoing!)

- Two related answers:
  - Chain Replication
  - Lab 2 - Primary/backup replication

- Limitations of this approach
  - Lab 2 - can only tolerate one replica failure (sometimes not even that!)
  - Both: need to have a fault-tolerant view service
  - How would we make *that* fault-tolerant?

Last week: Consensus

- The consensus problem:
  - multiple processes start w/ an input value
  - processes run a consensus protocol,
    then output chosen value
  - all non-faulty processes choose the same value

Paxos

- Algorithm for solving consensus in an asynchronous network
- Can be used to implement a state machine (VR, Lab 3, upcoming readings!)
- Guarantees safety w/ any number of replica failures
- Makes progrèss when a majority of replicas online

Paxos History

1989 Viewstamped Replication – Liskov & Oki

1990 Paxos – Leslie Lamport, “The Part-Time Parliament”

1998 Paxos paper published

~2005 First practical deployments

2010s Widespread use!

2014 Lamport wins Turing Award

Why such a long gap?

- Before its time?
- Paxos is just hard?
- Original paper is intentionally obscure:
  - “Recent archaeological discoveries on the island of Paxos reveal that the parliament functioned despite the peripatetic propensity of its part-time legislators. The legislators maintained consistent copies of the parliamentary record, despite their frequent forays from the chamber and the forgetfulness of their messengers.”

Meanwhile, at MIT

- Barbara Liskov & group develop
  Viewstamped Replication: essentially same protocol

- Original paper entangled with distributed transaction system & language

- VR Revisited paper tries to separate out replication (similar: RAFT project at Stanford)

- Liskov: 2008 Turing Award, for programming w/ abstract data types, i.e. object-oriented programming

Paxos History

1989 Viewstamped Replication – Liskov & Oki

1990 Paxos – Leslie Lamport, “The Part-Time Parliament”

1998 Paxos paper published

The ABCDs of Paxos [2001]
Paxos Made Simple [2001]
Paxos Made Practical [2007]
Paxos Made Live [2007]
Paxos Made Moderately Complex [2011]

~2005 P

2010s Widespread use!

2014 Lamport wins Turing Award

Three challenges about Paxos

- How does it work?
- Why does it work?
- How do we use it to build a real system?

- (these are in increasing order of difficulty!)

Why is replication hard?

- Split brain problem:
  Primary and backup unable to communicate w/ each other, but clients can communicate w/ them

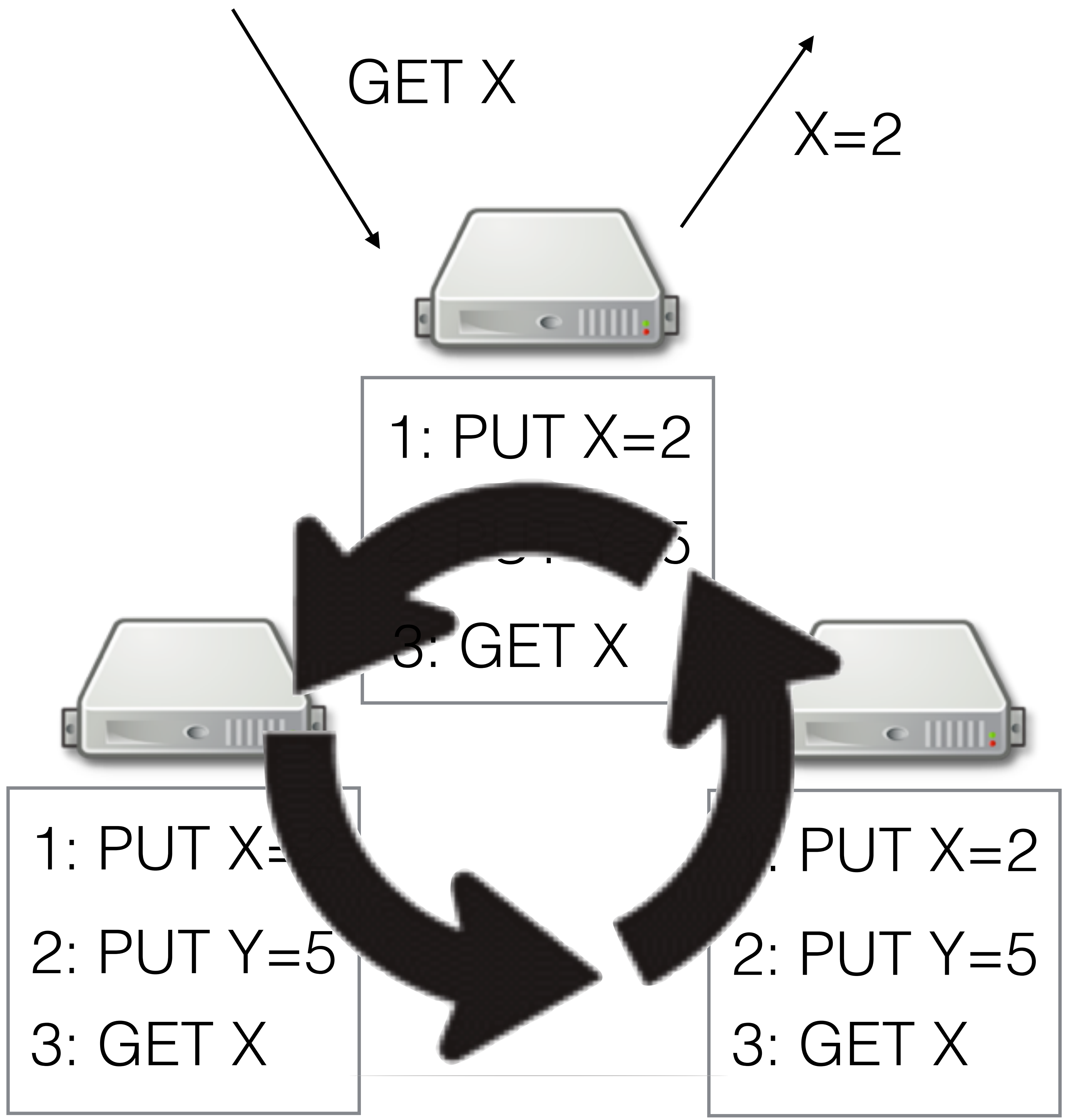
- Should backup consider primary failed and start processing requests?

  - What if the primary considers the backup is failed and keeps processing requests?

- How does Lab 2 (and Chain Replication) deal with this?

Using consensus for state machine replication

- 3 replicas, no designated primary, no view server
- Replicas maintain log of operations
- Clients send requests to some replica
- Replica proposes client's request as next entry in log, runs consensus
- Once consensus completes:
  execute next op in log and return to client


GET X
X=2
1: PUT X=2
3: GET X
1: PUT X=
2: PUT Y=5
3: GET X
PUT X=2
2: PUT Y=5
3: GET X

Two ways to use Paxos

- Basic approach (Lab 3)
  - run a completely separate instance of Paxos for each entry in the log
- Leader-based approach (Multi-Paxos, VR)
  - use Paxos to elect a primary (aka leader) and replace it if it fails
  - primary assigns order during its reign
- Most (but not all) real systems use leader-based Paxos

Paxos-per-operation

- Each replica maintains a log of ops
- Clients send RPC to any replica
- Replica starts Paxos proposal for latest log number
  - completely separate from all earlier Paxos runs
  - note: agreement might choose a different op!
- Once agreement reached: execute log entries & reply to client

Terminology

- *Proposers* propose a value
- *Acceptors* collectively choose one of the proposed values
- *Learners* find out which value has been chosen
- In lab3 (and pretty much everywhere!), every node plays *all three* roles!

Paxos Interface

- Start(seq, v): propose v as value for instance seq
- fate, v := Status(seq):
  find the agreed value for instance seq
- Correctness: if agreement reached,
  all agreeing servers will agree on same value
  (once agreement reached, can't change mind!)

How does an individual Paxos instance work?

Note: all of the following is in the context of deciding on the value for one particular instance, i.e., what operation should be in log entry 4?

Why is agreement hard?

- Server 1 receives Put(x)=1 for op 2,
  Server 2 receives Put(x)=3 for op 2
- Each one must do *something* with the first operation it receives
- …yet clearly one must later change its decision
- So: multiple-round protocol; tentative results?
- Challenge: how do we know when a result is tentative vs permanent?

Why is agreement hard?

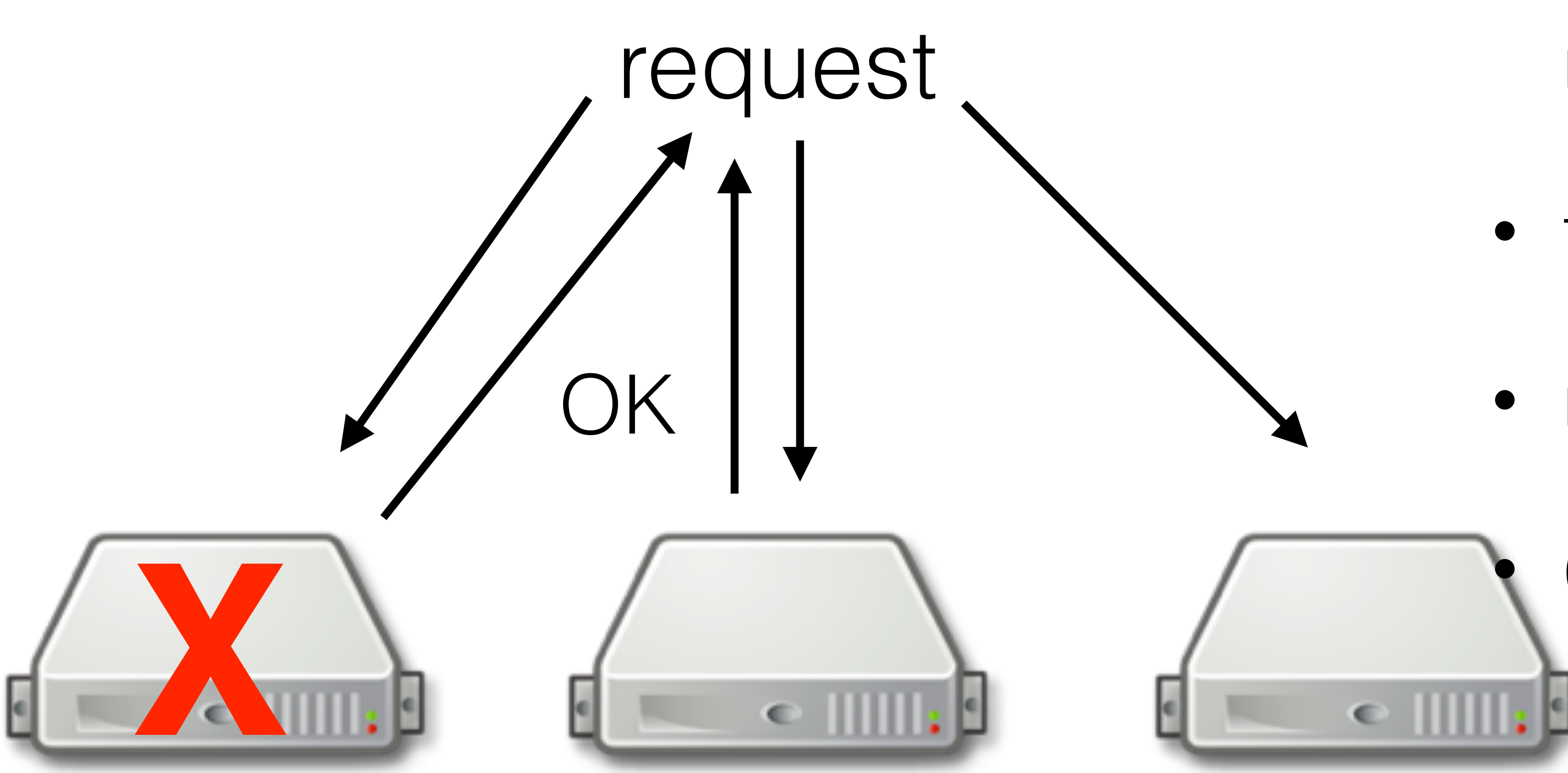
- S1 and S2 want to select Put(x)=1 as op 2, S3 and S4 don't respond
- Want to be able to complete agreement w/ failed servers — so are S3 and S4 failed?
  - or are they just partitioned, and trying to accept a different value for the same slot?
- How do we solve the split brain problem?

Key ideas in Paxos

- Need multiple protocol rounds that converge on same value
- Rely on **majority quorums** for agreement to prevent the split brain problem

Majority Quorums

- Why do we need 2f+1 replicas to tolerate f failures?
- Every operation needs to talk w/ a majority (f+1)
- Why?



- Have to be able to proceed w/ n-f responses
- f of those might fail
- need one left
- $(n-f)-f \geq 1 \Rightarrow n \geq 2f+1$

Another reason for quorums

- Majority quorums solve the split brain problem
- Suppose request N talks to a majority
- All previous requests also talked to a majority
- Key property: any two majority quorums intersect at at least one replica!
- So request N is guaranteed to see all previous operations
- What if the system is partitioned & no one can get a majority?

The mysterious *f*

- *f* is the number of failures we can tolerate
- For Paxos, need 2f+1 replicas
  (Chain Replication was f+1; some protocols need 3f+1)
- How do we choose *f*?
- Can we have more than 2f+1 replicas?

Paxos protocol overview

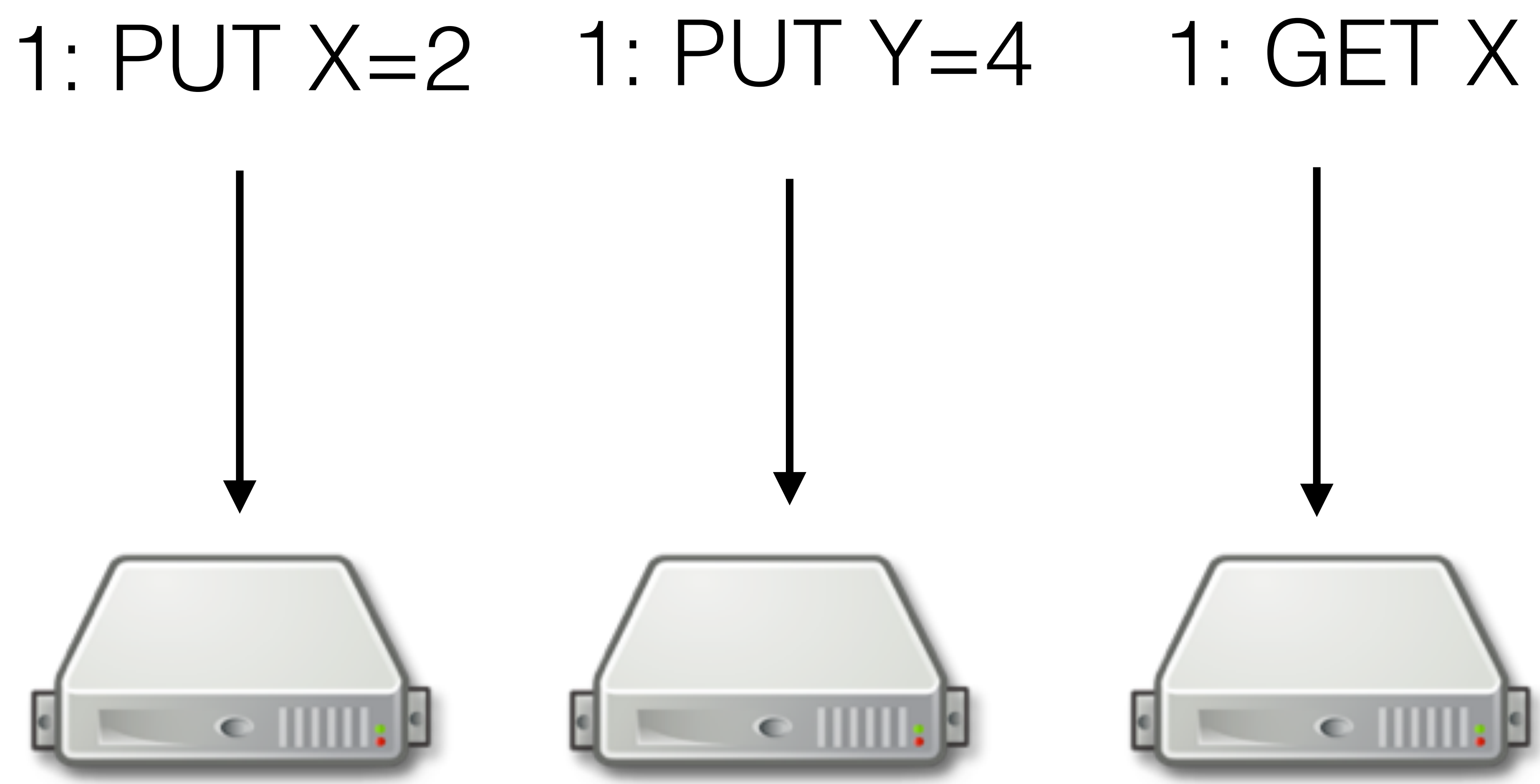
- Proposers select a value
- Proposers submit proposal to acceptors,
  try to assemble a majority of responses
  - might be concurrent proposers,
    e.g., multiple clients submitting different ops
  - acceptors must choose which requests they
    accept to ensure that algorithm converges

Strawman

- Proposer sends propose(v) to all acceptors
- Acceptor accepts first proposal it hears
- Proposer declares success if its value is accepted by a majority of acceptors
- What can go wrong here?

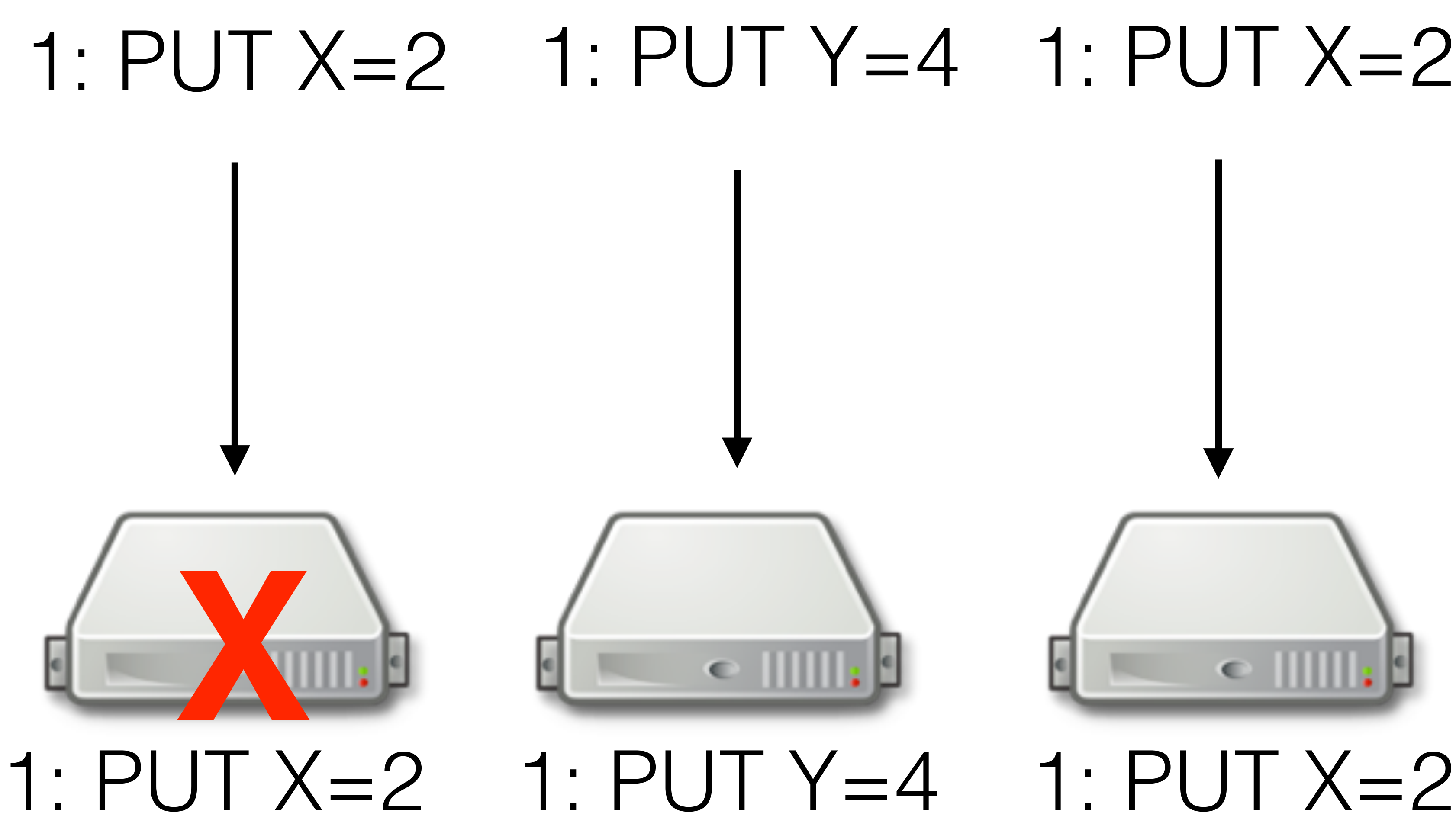
Strawman

- What if no request gets a majority?



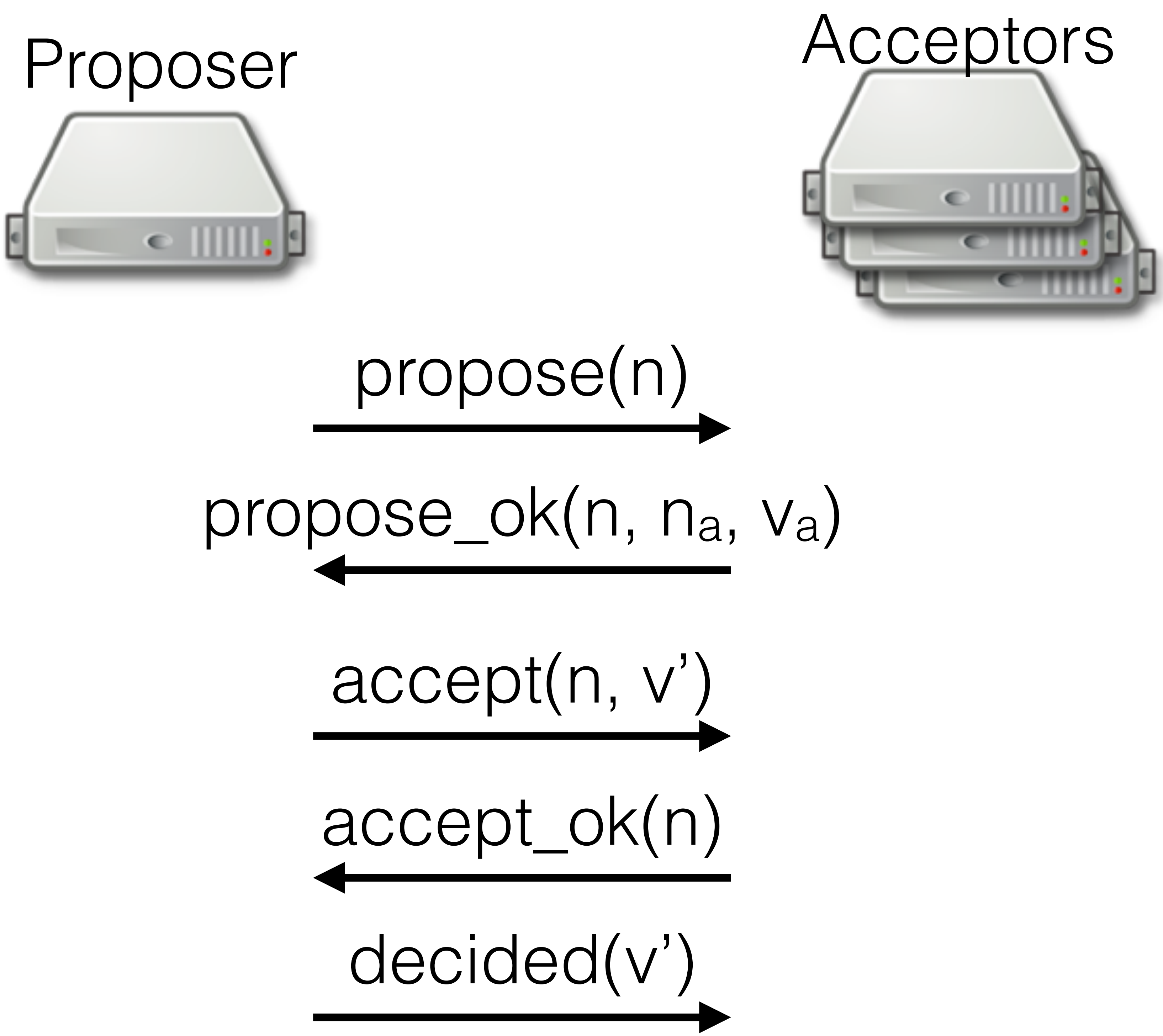
Strawman

- What if there's a failure after a majority quorum?



- How do we know which request succeeded?

Basic Paxos exchange



Definitions

- n is an id for a given proposal attempt
  not an instance — this is still all within one instance!
  e.g., n = <time, server_id>
- v is the value the proposer wants accepted
- server S *accepts* n, v
  => S sent accept_ok to accept(n, v)
- n, v is *chosen* => a majority of servers accepted n,v

Key safety property

- Once a value is chosen, no other value can be chosen!
- This is the safety property we need to respond to a client: algorithm can't change its mind!
- Trick: another proposal can still succeed, *but* it has to have the same value!
- Hard part: "chosen" is a systemwide property: no replica can tell locally that a value is chosen

Paxos protocol idea

- proposer sends propose(n) w/ proposal ID,
  but doesn't pick a value yet
- acceptors respond w/ any value already accepted and promise not to accept proposal w/ lower ID
- When proposer gets a majority of responses
  - if there was a value already accepted, propose that value
  - otherwise, propose whatever value it wanted

Paxos acceptor

- n_p = highest propose seen
  n_a, v_a = highest accept seen & value

- On propose(n)
  if $n > n_p$
    $n_p = n$
    reply propose_ok(n, n_a, v_a)
  else reply propose_reject

- On accept(n, v)
  if $n \geq n_p$
    $n_p = n$
    $n_a = n$
    $v_a = v$
    reply accept_ok(n)
  else reply accept_reject

Example: Common Case

Proposer Acceptor Acceptor Acceptor

propose(1)

propose_ok(1, nil, nil)

propose_ok(1, nil, nil)

propose_ok(1, nil, nil)

accept(1, V)

accept_ok(1)

accept_ok(1)

accept_ok(1)

decided(V)

What is the commit point?

- i.e., the point at which, regardless of what failures happen, the algorithm will always proceed to choose the same value?
- once a majority of acceptors send accept_ok(n)!
- why not when a majority of proposers send propose_ok(n)?

Acceptor

propose_ok(10)

accept_ok(10, X)

Acceptor

propose_ok(10)

propose_ok(11)

Acceptor

propose_ok(10)

propose_ok(11)

accept_ok(11, Y)

- Has a value been chosen?
- Could either X or Y be chosen?
- What happens if #2 gets accept(10, X)?
- What happens if #1 gets accept(11, Y)?

- **Why does the proposer need to choose the value v_a with highest n_a?**
- Guaranteed to see any value that has already obtained a majority of acceptors
    - can't change this value, so we need to use it!
- Will also see any value that *could subsequently* obtain a majority of acceptors
    - because the proposal prevents any lower-numbered proposal from being accepted

What about FLP?

- No determinstic algorithm for solving consensus in an asynchronous network is both safe (correct) and live (terminates eventually)
- Paxos is an algorithm for solving consensus…
- Paxos must not be guaranteed to be live
- How can it get stuck?

Worst-case for Paxos

Proposer Acceptor Acceptor Acceptor Proposer

propose(1)

prop_ok(1) prop_ok(1) prop_ok(1)

propose(2)

prop_ok(2) prop_ok(2) prop_ok(2)

accept(1)

accept_rej(1) accept_rej(1) accept_rej(1)

propose(3)

accept(2)

prop_ok(3) prop_ok(3) prop_ok(3)

accept_rej(2) accept_rej(2) accept_rej(2)

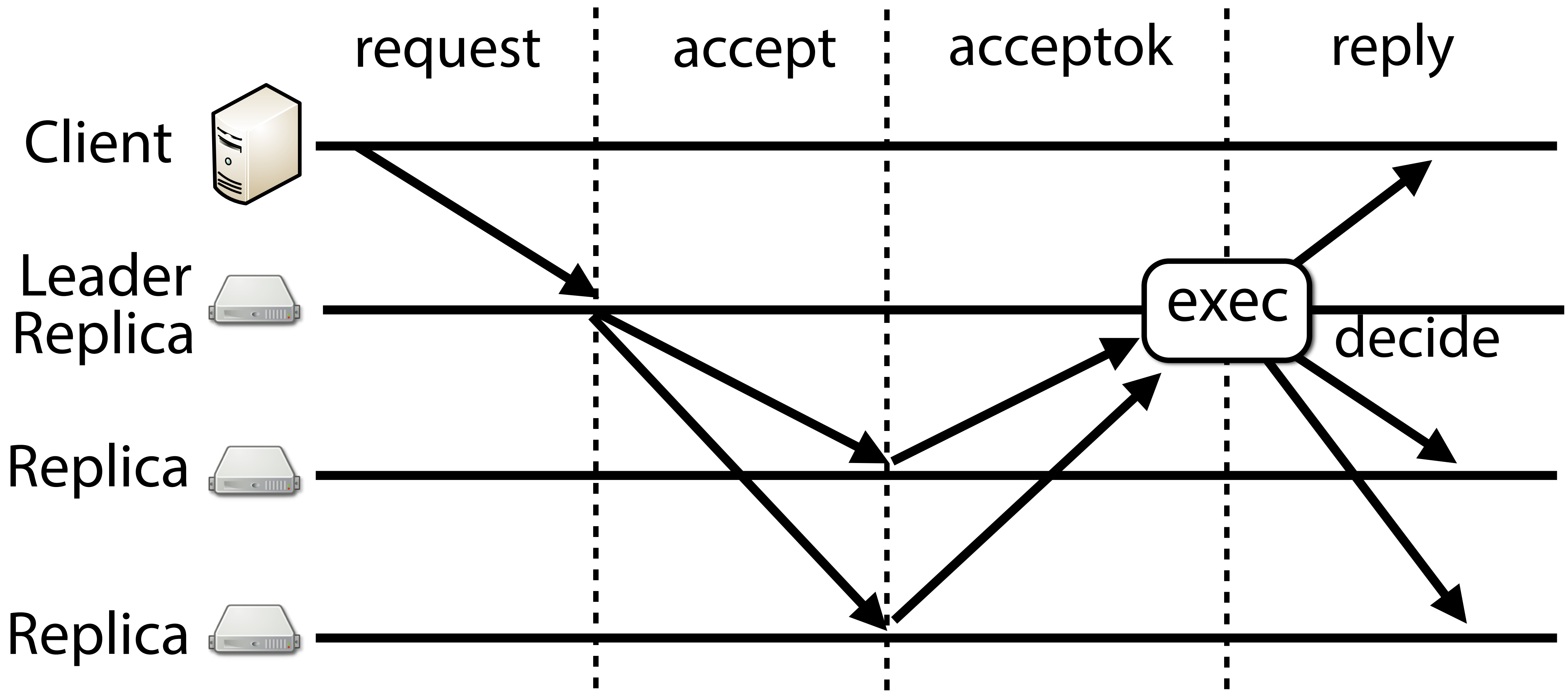
What can we do about this?

- don’t retry immediately; wait random time then retry
- designate one replica as leader (aka distinguished proposer), have it make all the proposals
- what if that replica fails?
  - just an optimization, other replicas can still make proposals if they think it failed

Multi-Paxos

- All of the above was about a *single instance*, i.e., agreeing on the value for *one* log entry
- In reality: series of Paxos instances
- Optimization: if we have a leader, have it run the first phase for multiple instances at once
- propose(n): acceptor sets $n_p = n$ for this instance *and all future instances*
- Then the proposer can jump to the accept phase

Multi-Paxos


request
accept
acceptok
reply
Client
Leader Replica
Replica
Replica
exec
decide

Viewstamped Replication

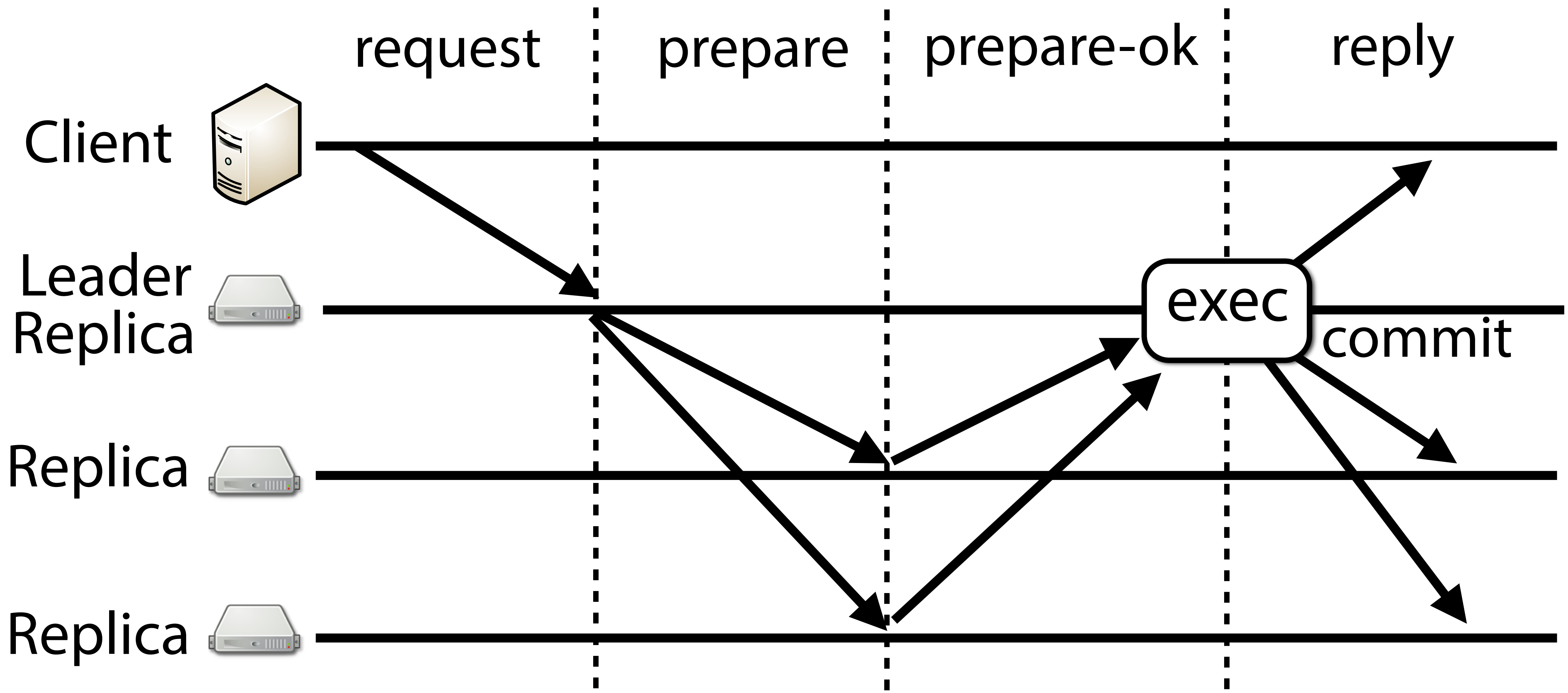
- A Paxos-like protocol presented in terms of state machine replication
- i.e, a system-builder’s view of Paxos
- see also RAFT from Stanford

Viewstamped Replication is *exactly* Multi-Paxos!

Starting point

- 2f+1 replicas, one of them is the primary
- each one maintains a numbered log of operations either PREPARED or COMMITTED
- clients send all requests to primary
- primary runs a two-phase commit over replicas

2-phase commit



Beyond 2PC

- 2PC does not remain available with failures
- So let's try requiring a majority quorum:
  f+1 PREPARE-OKs, including the primary
- can tolerate f backup failures (no primary failure)
- Minor detail: what if backup receives op n+1 without seeing op n
  - need state transfer mechanism

The hard part

- need to detect that the primary has failed (timeout?)
- need to replace it with a new primary
  - need to make sure that the new primary knows about all operations committed by the primary
  - need to keep the old primary from completing new operations
  - need to make sure that there are no race conditions!

Replacing the primary

- Each replica maintains a view number,
  view number determines the primary,
  process PREPARE-OK only if view number matches

- When primary suspected faulty: send
  <START-VIEW-CHANGE, new v> to all

- On receiving START-VIEW-CHANGE:
  increment view number, stop processing reqs
  send <DO-VIEW-CHANGE, v, log> to new primary

- When primary receives DO-VIEW-CHANGE from majority:
  take log with highest seen (not necessarily committed) op
  install that log, send <START-VIEW, v, log> to all

Why is this correct?

Why is this correct?

- New primary sees every operation that could possibly have completed in old view
  - every completed operation was processed by majority of replicas, and we have DO-VIEW-CHANGE logs from a majority
- Can the old primary commit new operations?
  - no - once a replica sends DO-VIEW-CHANGE it stops listening to the old primary!

Why is this correct?

- Because it's Paxos!
- View change = propose a new primary
  - a two-phase protocol involving majorities
  - other replicas promise not to accept ops in old view
  - and proposer finds out all ops accepted in old view and must propose them in new view

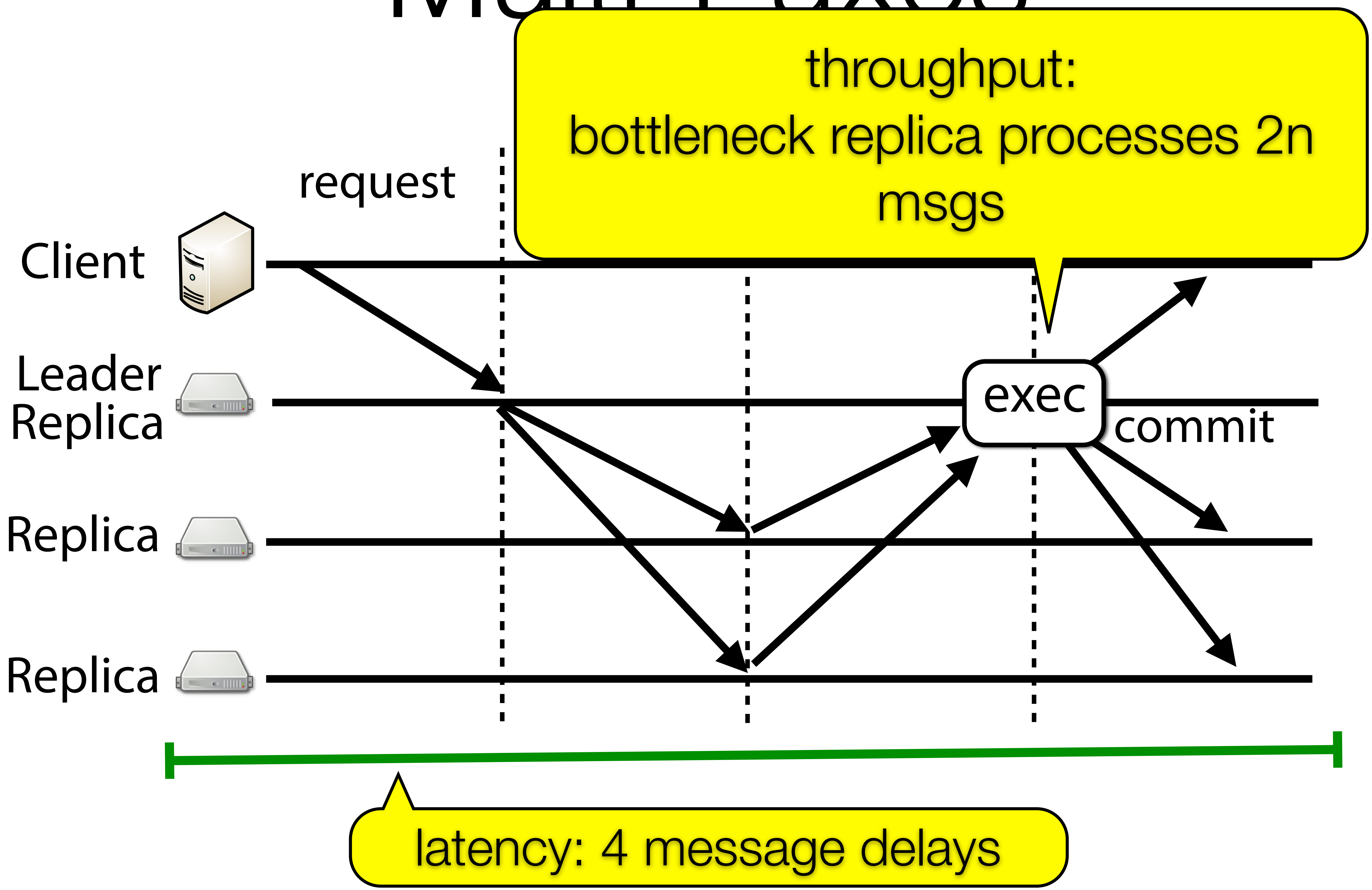
VR = (Multi-)Paxos

- view number = proposal number
- start-view-change(v) = propose(v)
- do-view-change(v) = propose_ok(v)
- start-view(v, log) = accept(v, op) for appropriate instance
- prepare(v, opnum, op) = accept(v, op) for instance opnum
- prepare_ok(v, opnum) = accept_ok(v, op) for instance opnum
- commit(opnum, op) = decided(opnum, op)

Paxos performance

- What determines Paxos performance?
- We'll consider Multi-Paxos / VR
  since it's the most common way to use Paxos

Multi-Paxos


throughput:
bottleneck replica processes 2n msgs
request
Client
Leader Replica
Replica
Replica
exec
commit
latency: 4 message delays

Batching

- Have leader accumulate requests from many clients
- Run one round of Paxos in parallel to add them all to the log
- Much higher throughput
- Potentially higher latency (can get it about even)

Partitioning

- One idea: run multiple Paxos groups
  - each replica will be a leader in some, follower in others - spreads load around
  - very common in practice
- Separate idea: partition instances, different leaders for each instance
  - some protocols do this for higher throughput
  - more complicated, easy to get wrong