# Practical Byzantine Fault Tolerance

# From Paxos to

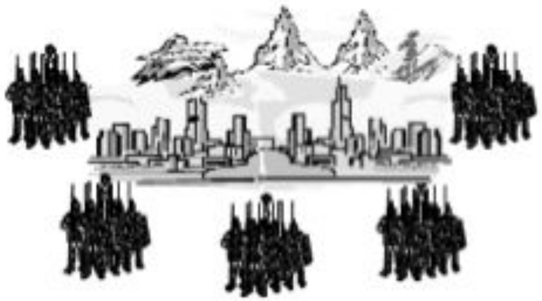
# Practical Byzantine Fault Tolerance

# From Paxos to

# Practical Byzantine Fault Tolerance

# via View-Stamped Replication

# Historical Motivation*



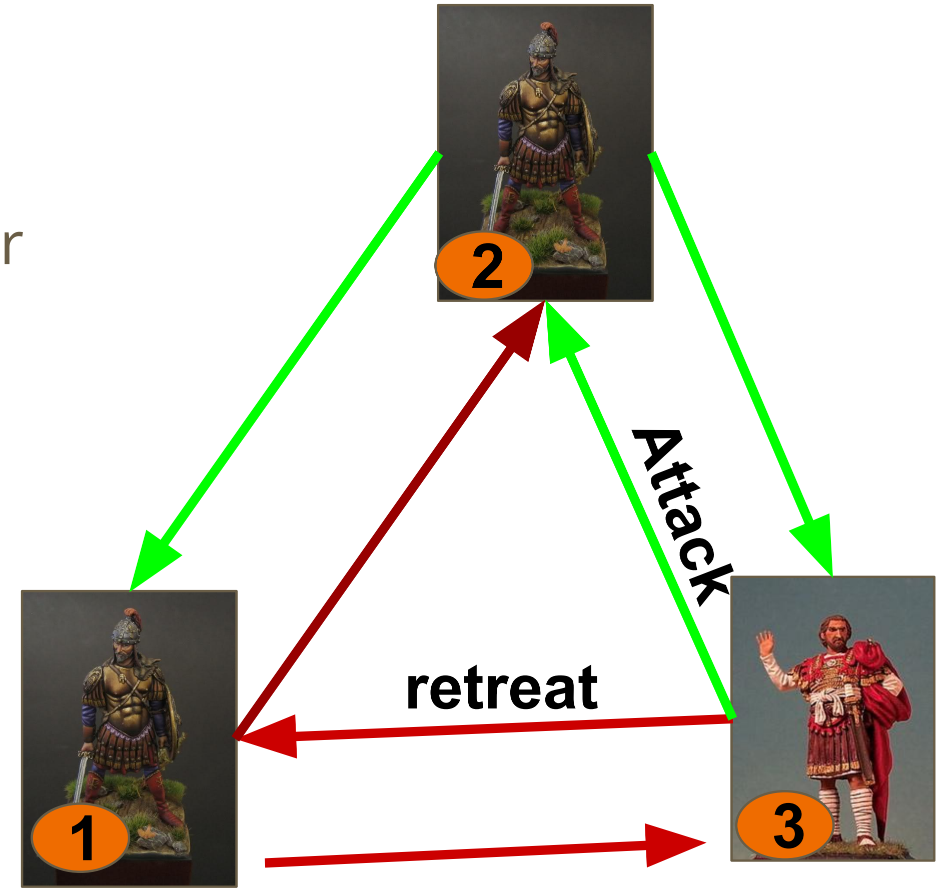
- A Byzantine army decides to attack/ retreat
  - **N** generals, **f** of them are traitors (can collude)
  - Generals camp outside the castle
    - Decide individually based on their field information
  - Exchange their plans by messengers
    - Can be killed, can be late, etc
- Requirements
  - All loyal generals agree on the same plan of action
  - 

A BFT protocol helps loyal generals decide correctly

*http://research.microsoft.com/en-us/um/people/lamport/pubs/byz.pdf

# Why is it hard?

- Simple scenario
  - 3 generals, third general is traitor
  - Traitor sends different plans
  - If decision is based on majority
    - (1) and (2) decide differently
    - (2) attacks and gets defeated
- More complicated scenarios
  - Messengers get killed, spoofed
  - Traitors confuse others:
    - (3) tells (1) that (2) retreats, etc



# Computer Science Setting

- A general ⇔ a program component/ processor/ replica
    - Replicas communicate via messages/rpc calls
    - Traitors ⇔ Failed replicas
- Byzantine army ⇔ A deterministic replicated service
    - The service has states and some operations
    - The service should cope with failures
        - State should be consistent across replicas
    - Seen in many applications
        - replicated file systems, backup , Distributed servers
        - Shared ledger between banks

# Byzantine Fault Tolerance Problem

- Distributed computing with faulty replicas
  - **N** replicas
  - **f** of them maybe faulty (crashed/ compromised)
  - Replicas initially start with the same state
- Given a request/ operation, the goal is:
  - Guarantee that all non-faulty replicas agree on the next state
  - Provide system consistency even when some replicas may be inconsistent

# Properties

- Safety
  - *Agreement:* All non-faulty replicas agree on the same state
  - *Validity*: The chosen state is valid
- Liveness
  - Some state is eventually agreed
  - If a state has been chosen, all replicas eventually arrive at the state

# 1000+ Models of BFT Problem

- Network: synchronous, asynchronous, in between, etc
- Failure types: fail-stop (crash), Byzantine,  etc
- Adversarial model
  - Computationally bounded
  - Universal adversary: can see everything, private channels
  - Static, dynamic adversary
- Communication types
  - Message passing, broadcast, shared registers
- Identities of replicas
  - Pre-established/ unknown?

An algorithm that works for one model may not work for others!

- Sparse network, full (complete) network

# Previous Work

- The “celebrated” Impossibility Result
    - Only one faulty replica makes (*deterministic*) agreement impossible in the asynchronous model
    - Intuition
        - A faulty replica may just be slow, and vice versa.
        - E.g. cannot make progress if don’t receive enough messages
    - Most protocols
        - Require synchrony assumption to achieve safety and liveness
        - Have some *randomization*: terminate with high prob., agreement can be altered with non-zero prob., etc.

# Previous Work(2)

- Paxos
  - Model
    - Network is asynchronous (messages are delayed arbitrarily, but eventually delivered)
    - Tolerate crashed failure
  - Guarantee safety, but not  liveness
    - The protocol may not terminate
    - Terminate if the network is synchronous eventually
  - One of the main results
    - Require at least **2f+1** replicas to tolerate **f** faulty replicas

# 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

~2005 F

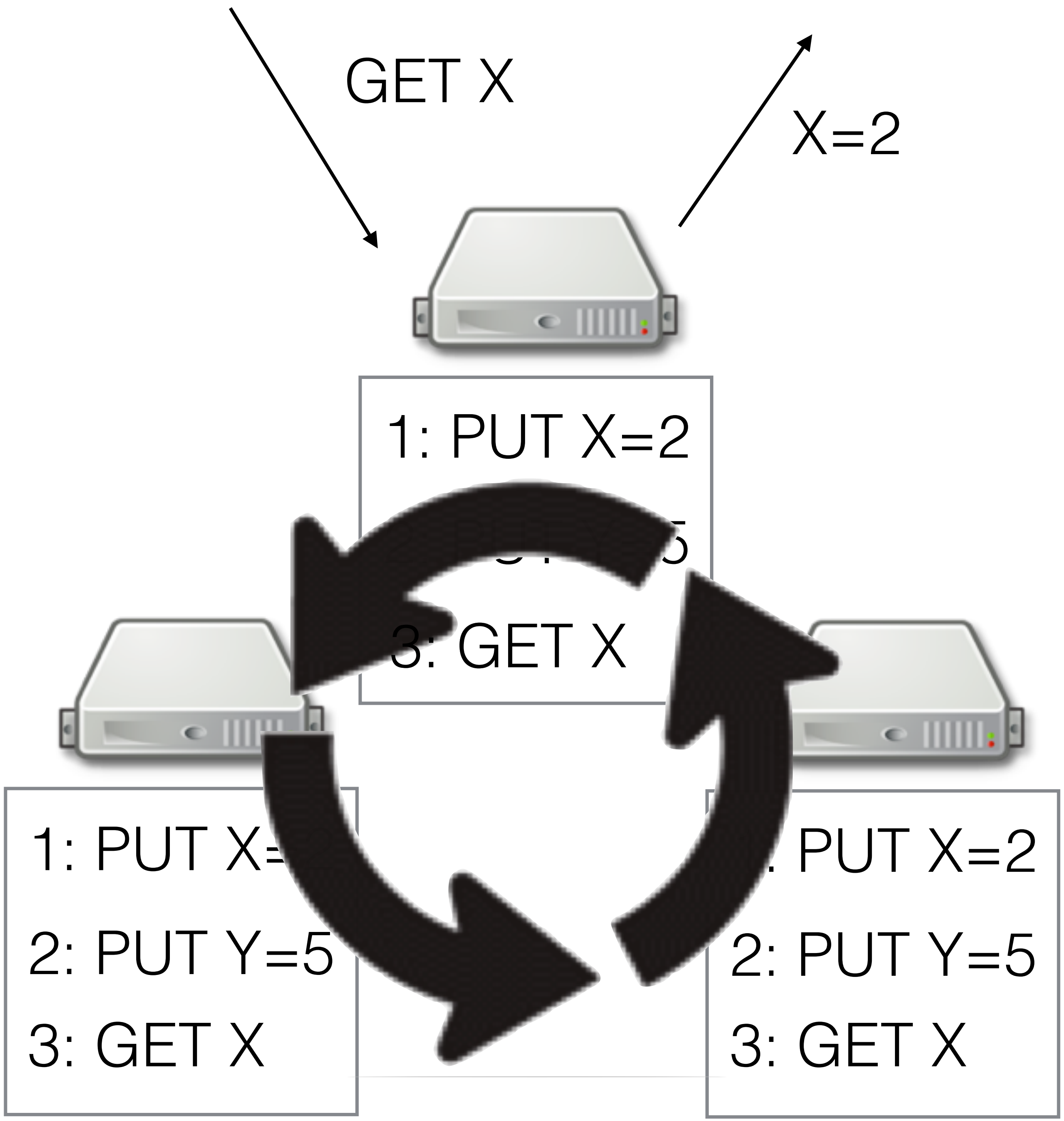
The ABCDs of Paxos [2001]
Paxos Made Simple [2001]
Paxos Made Practical [2007]
Paxos Made Live [2007]
Paxos Made Moderately Complex [2011]

2010s Widespread use!

2014 Lamport wins Turing Award

# 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

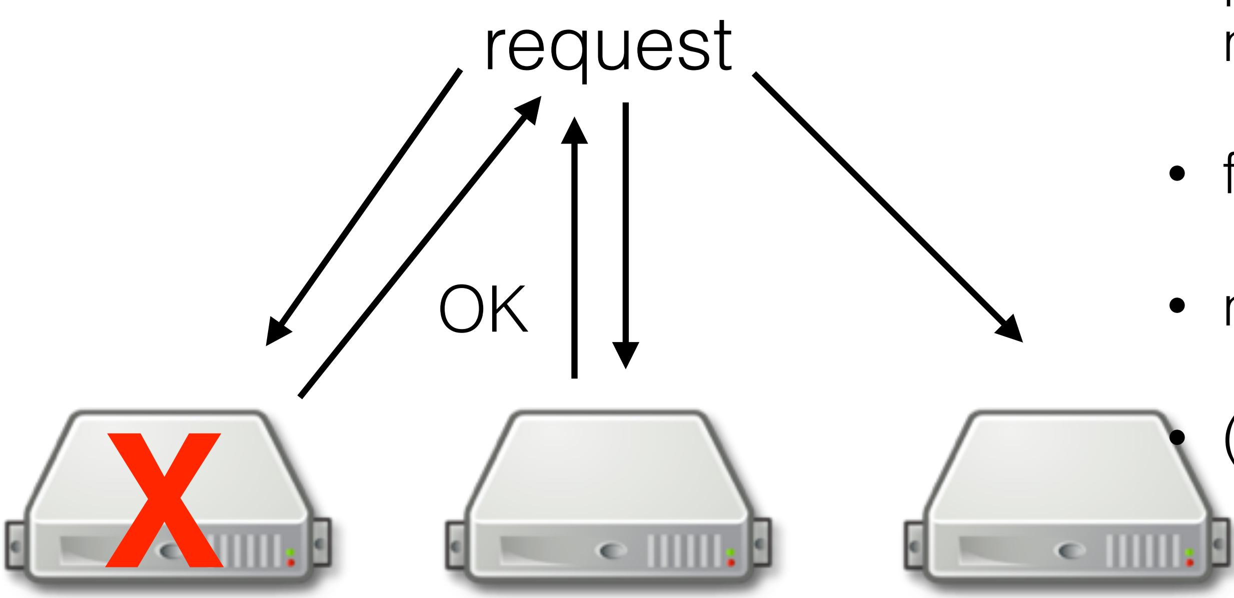
- Correctness: if agreement reached,
  all agreeing servers will agree on same value
  (once agreement reached, can't change mind!)

# 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?

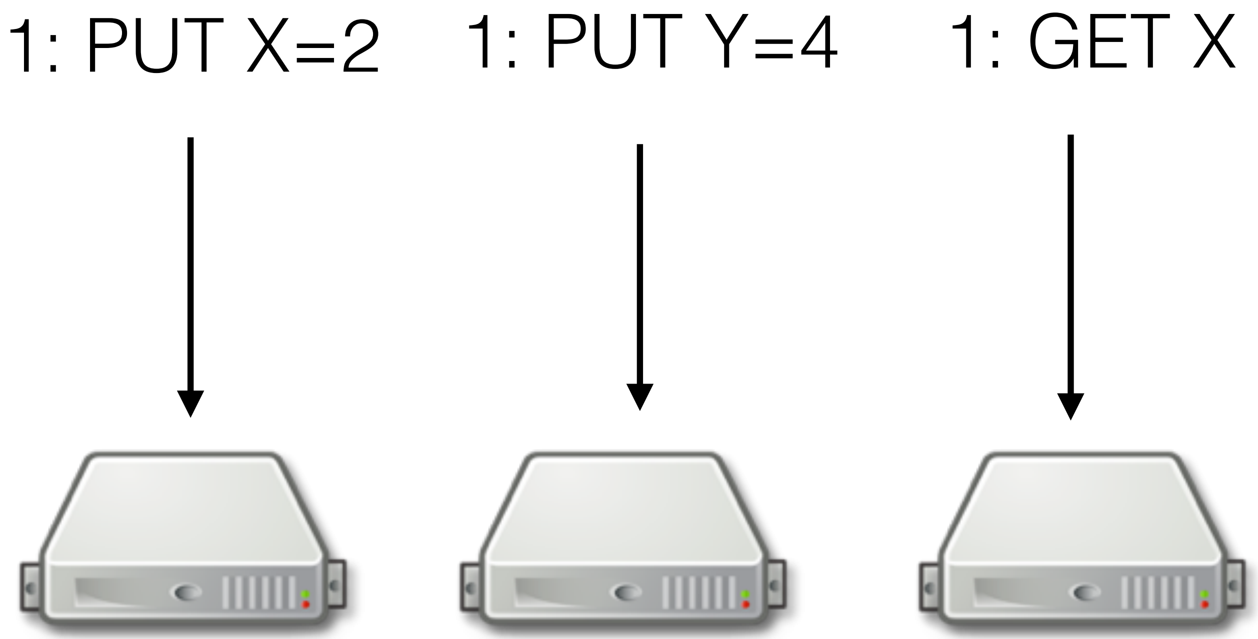
# Strawman

- 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?

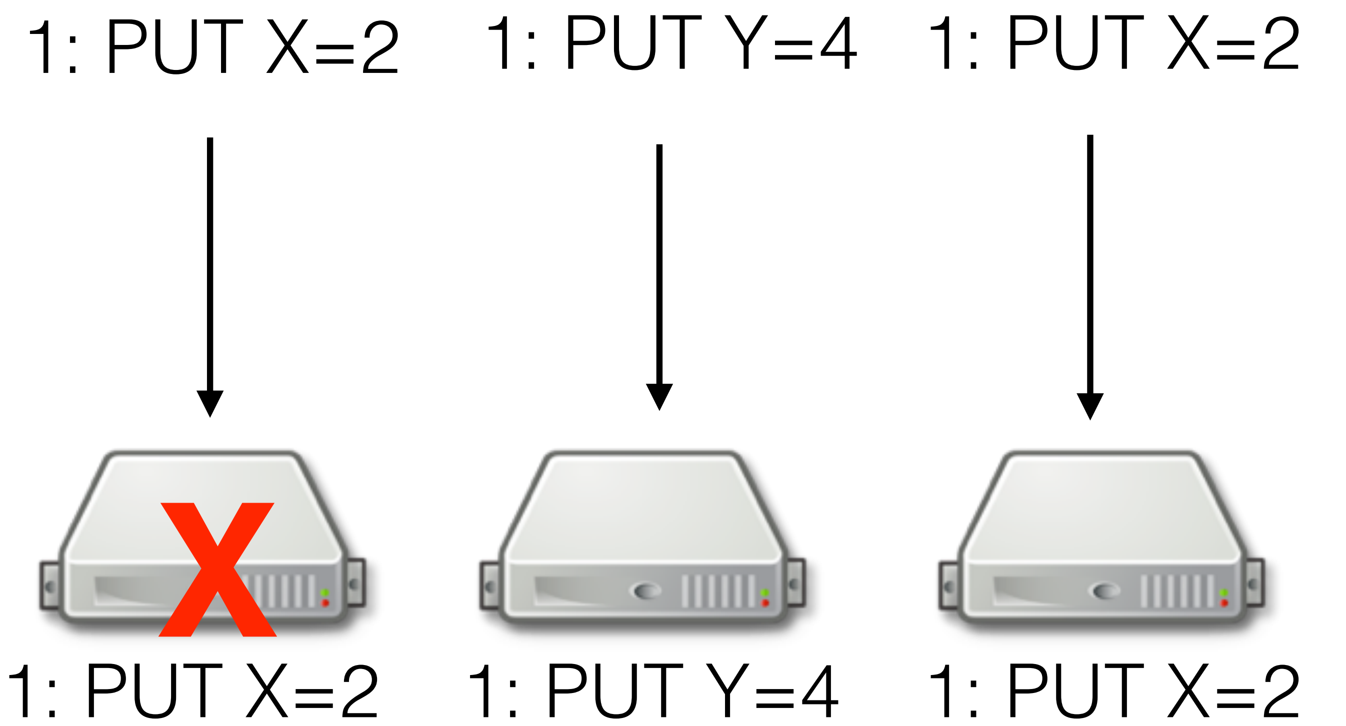
# Strawman

- What if no request gets a majority?



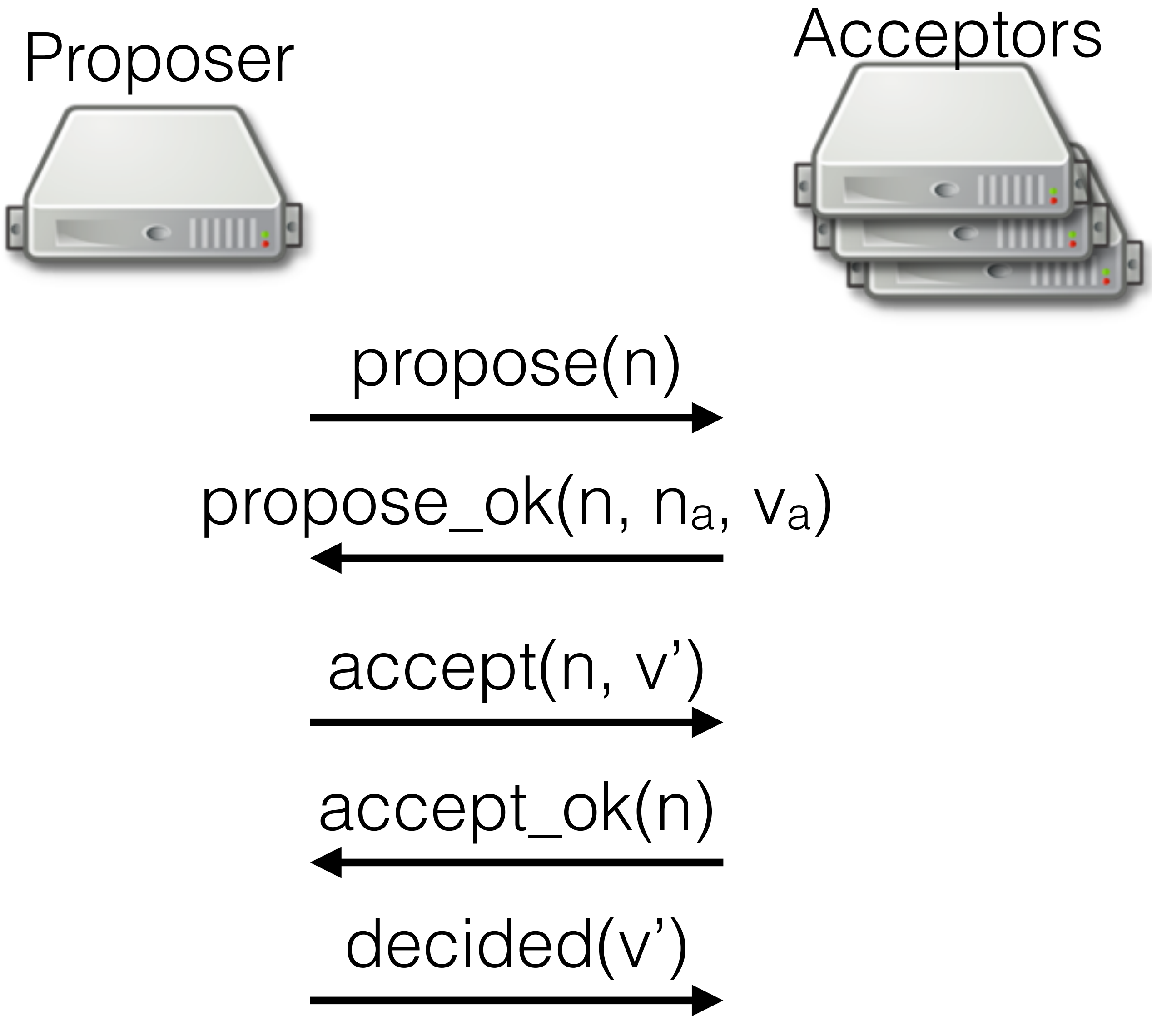
# Strawman

- 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)?

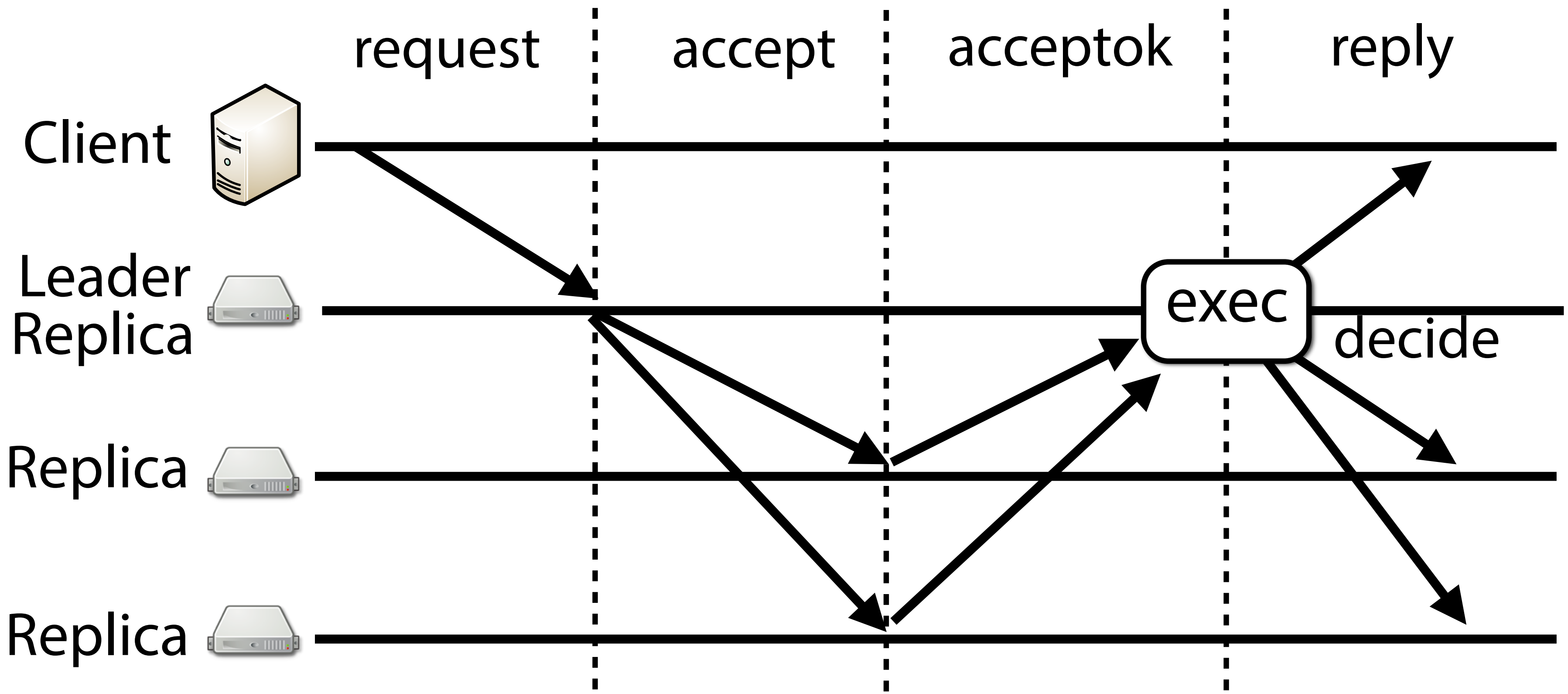
- **Why does the proposer need to choose the value $v_a$ with highest $n_a$?**

- 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

# 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

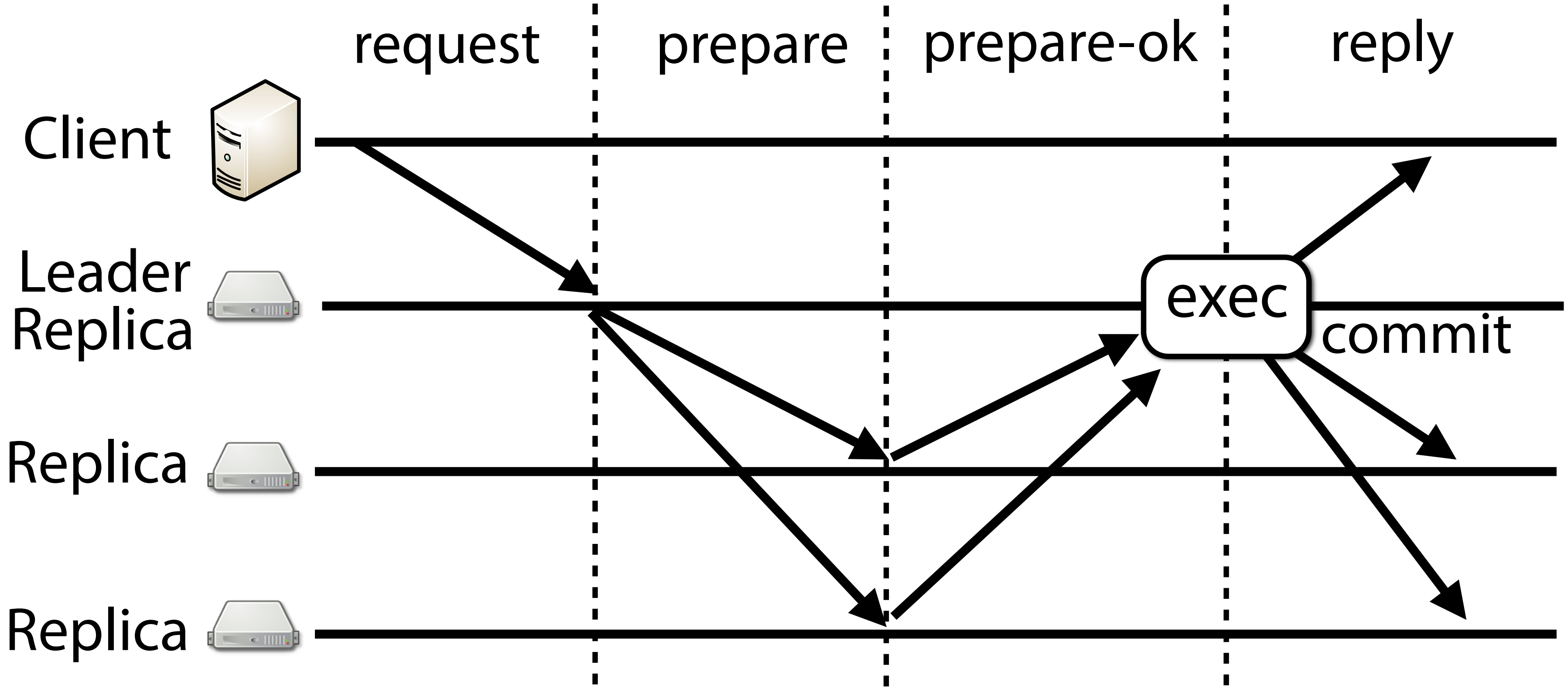
- A Paxos-like protocol presented in terms of state machine replication
- i.e, a system-builder’s view of Paxos
- see also RAFT from Stanford

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

**Discuss how this is exactly Paxos phases**

# 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!

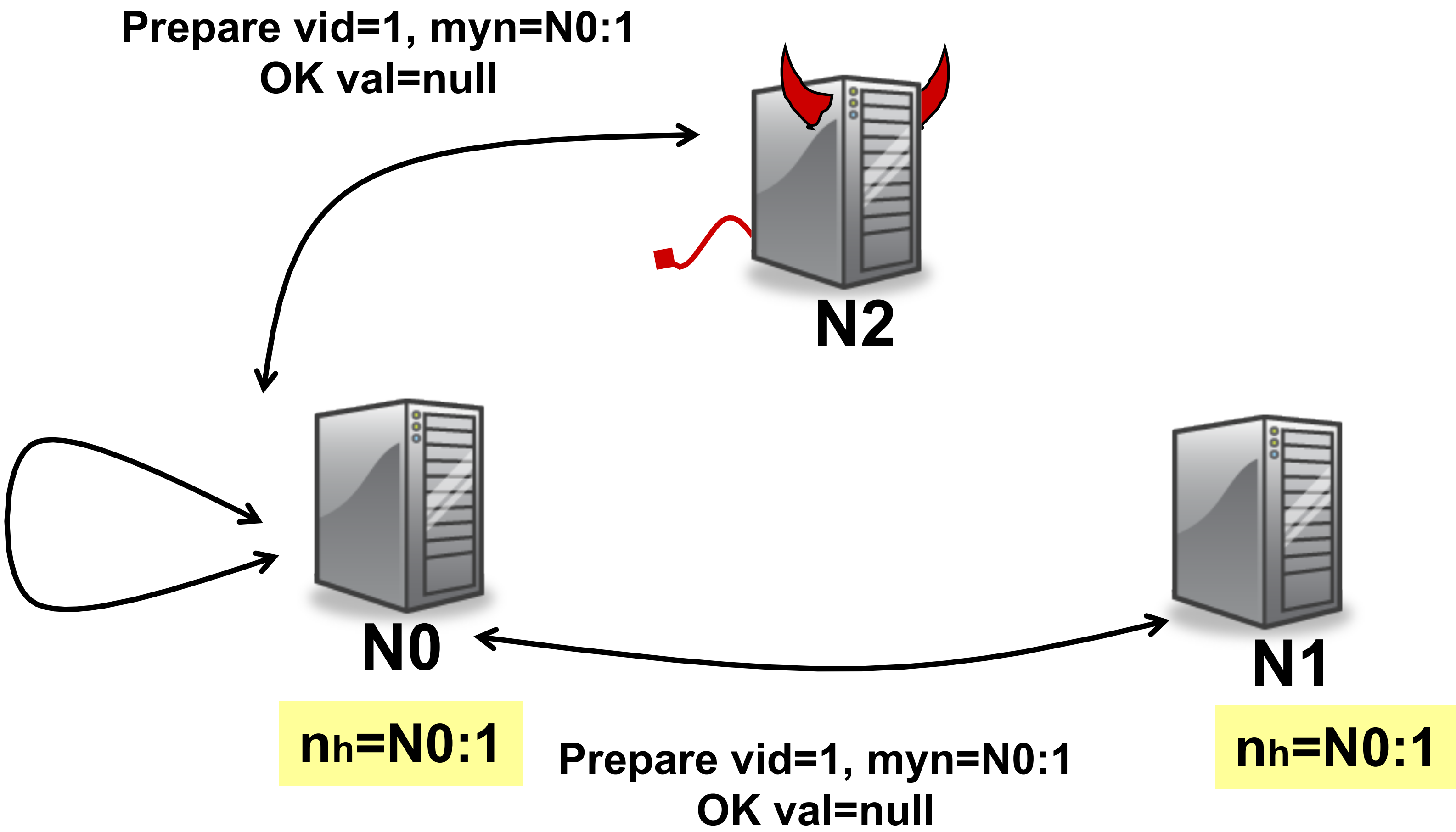
# Why is this correct?

- 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

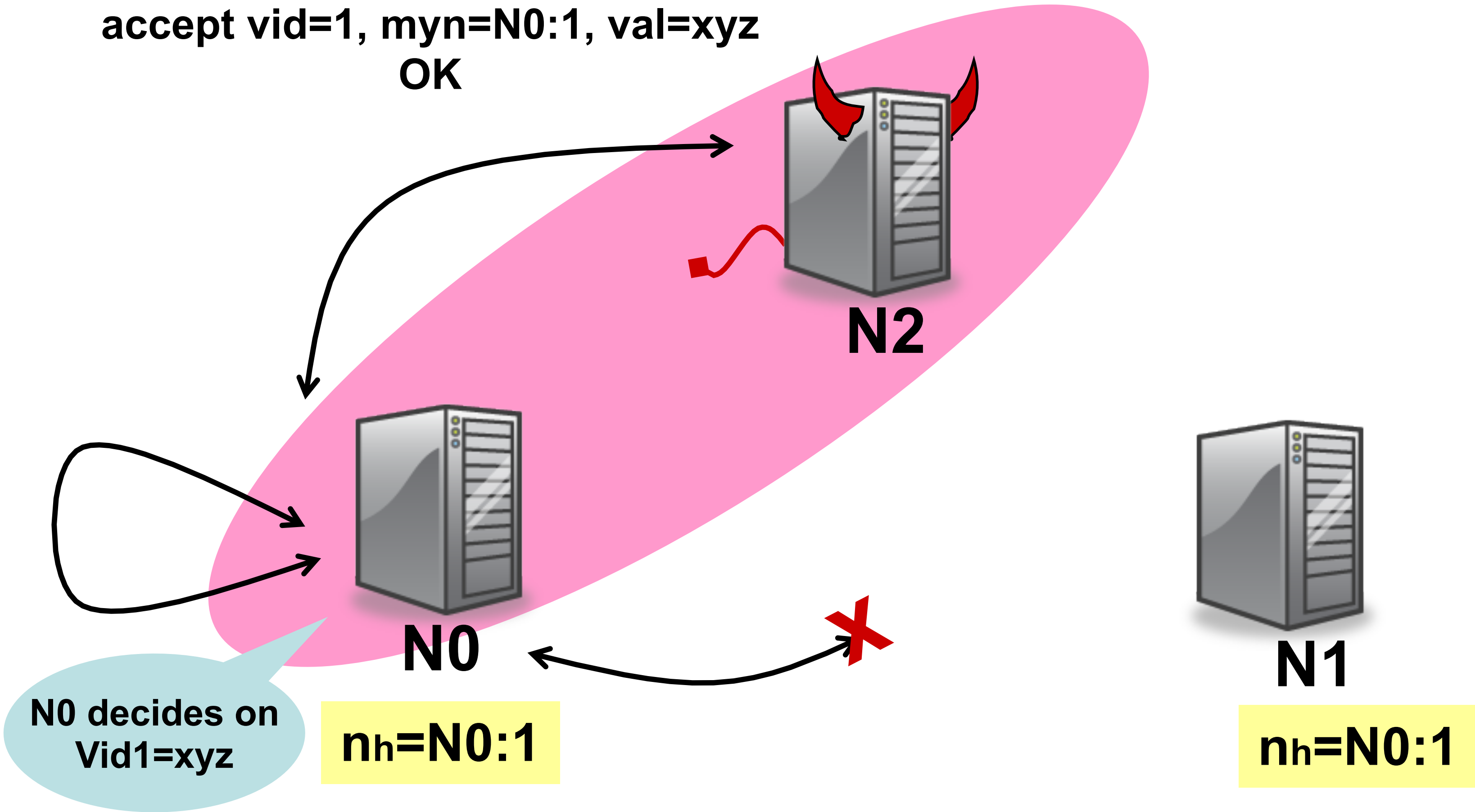
# VR = (Multi-)Paxos

- 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)

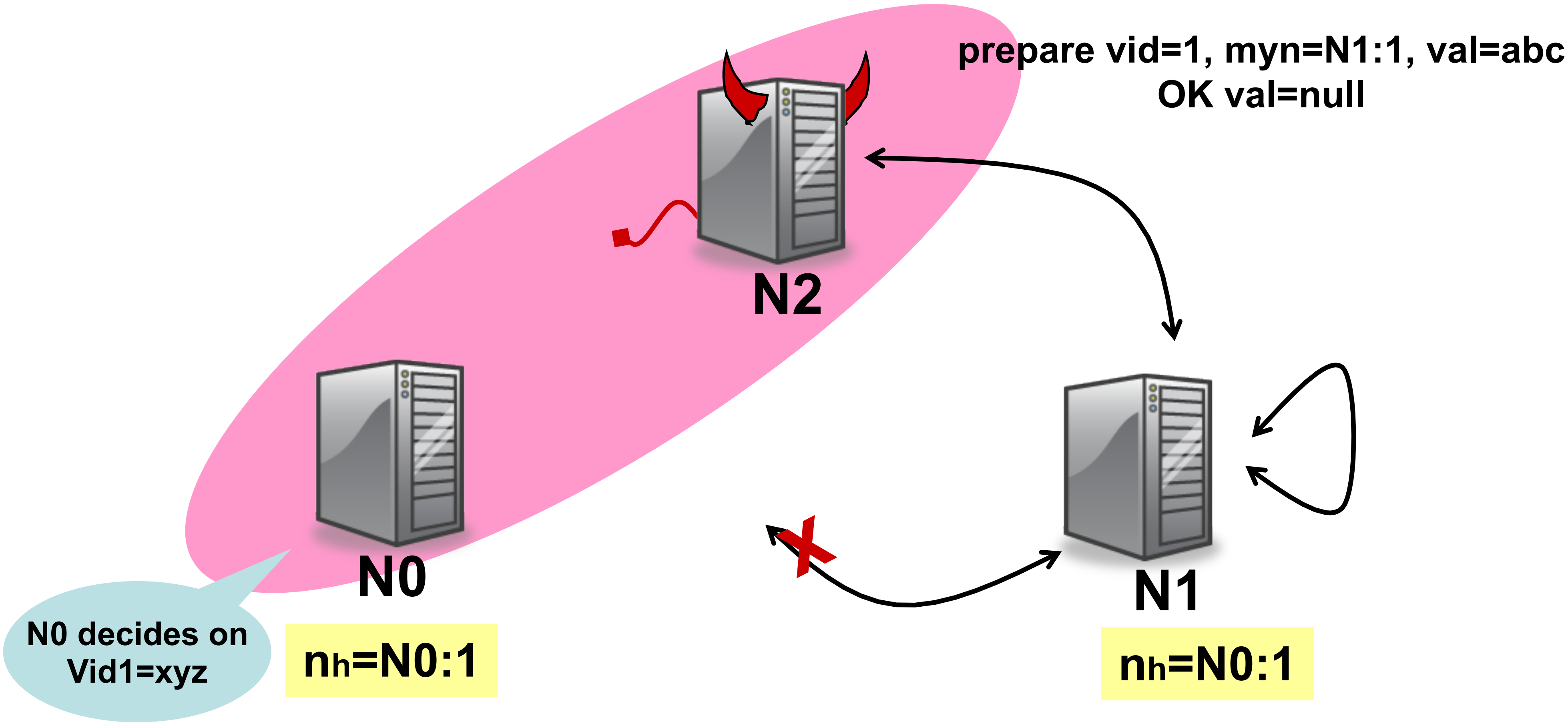
# Paxos under Byzantine faults



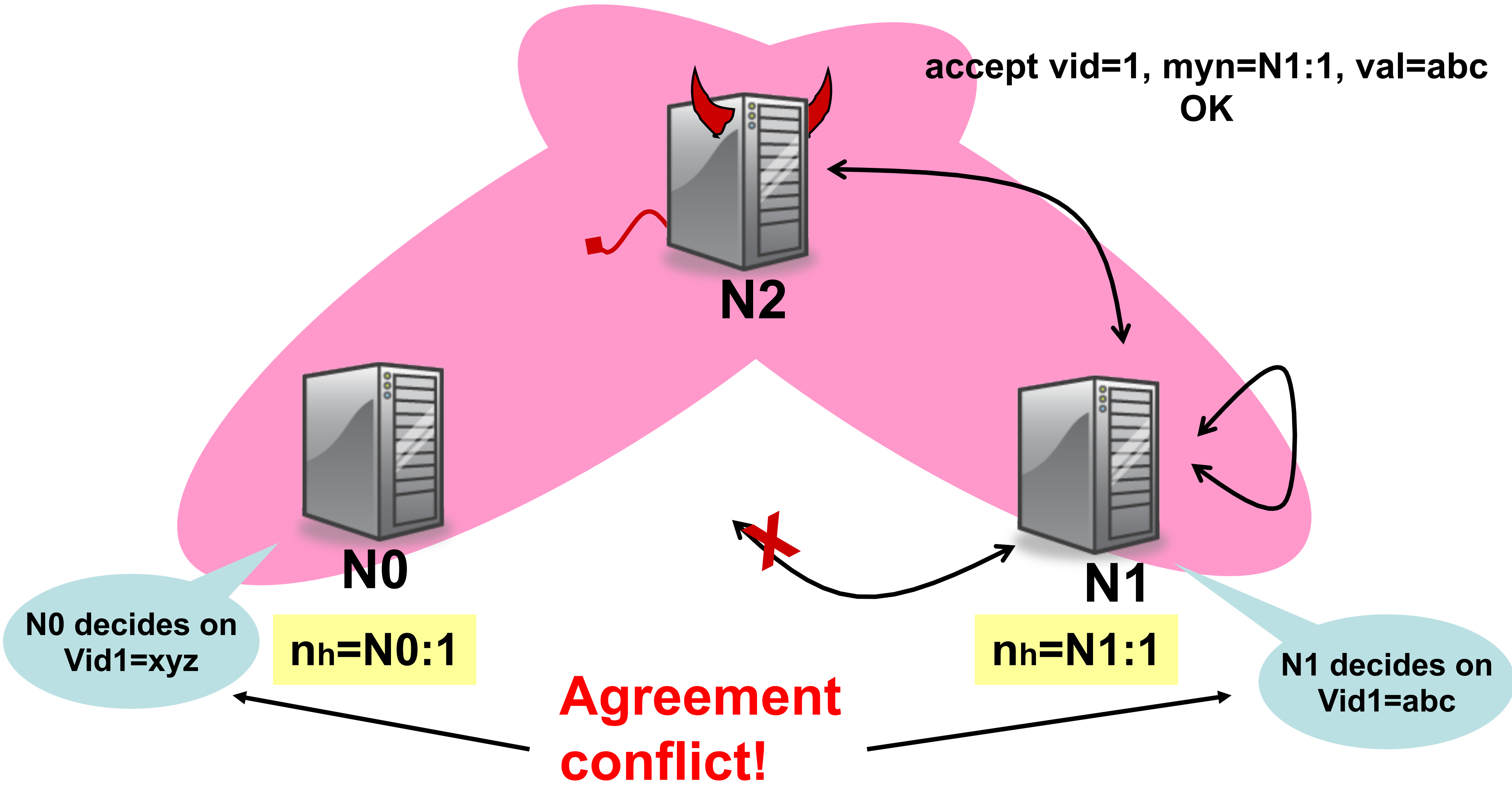
# Paxos under Byzantine faults



# Paxos under Byzantine faults



# Paxos under Byzantine faults



# Is Crashed Failure Good Enough?

- Byzantine failures are on the rise
  - Malicious successful attacks become more serious
  - Software errors are more due to the growth in size and complexity of software
  - Faulty replicas exhibit Byzantine behaviors
- How to reach agreement even with Byzantine failures?

# Practical Byzantine Fault Tolerance*

- Is introduced almost 20 years after Paxos
- Model in PBFT is practical
  - Asynchronous network
  - Byzantine failure
- Performance is better
  - Low overhead, can run in real applications
- Adoption in industry
  - See Tendermint, IBM's Openchain, and ErisDB

*http://pmg.csail.mit.edu/papers/osdi99.pdf

# Core Algorithm

# Goal and Basic Idea

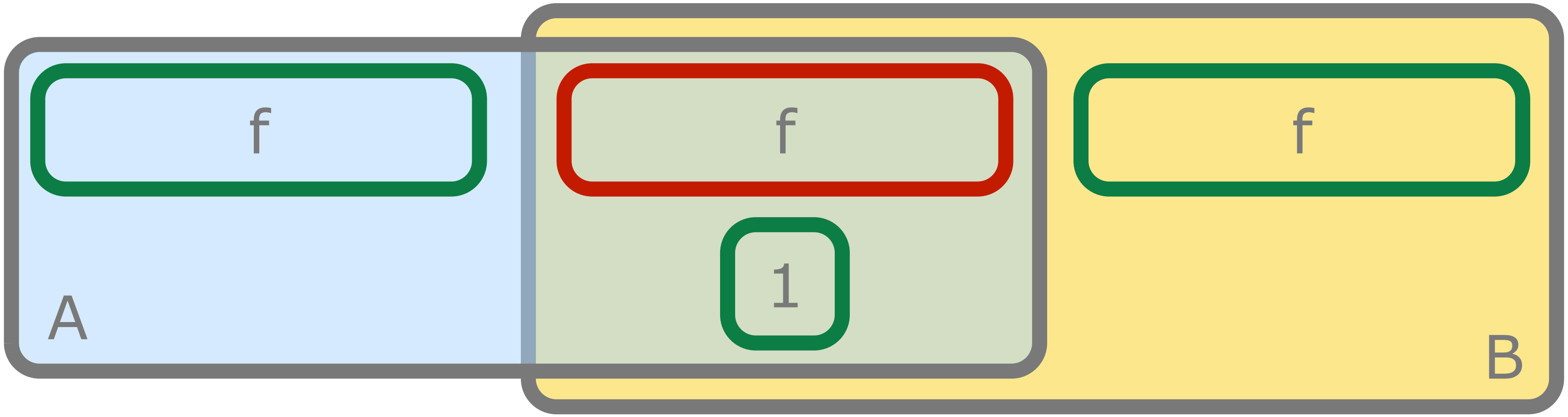
- Goal: build a linearizable replicated state machine
  - Replicate for fault prevention, not for scalability/availability
  - Agree on operations and their order
- Basic idea: get same statement from enough nodes to know that non-faulty nodes are in same state
  - Assume 3f+1 nodes, with at most f faults
  - Assume signed messages
  - Assume deterministic behavior
  - Assume no systematic failures

# The cf+1 of BFT

- f+1 nodes
  - One node must be non-faulty, ensuring correct answer
- 2f+1 nodes
  - A majority of nodes must be non-faulty, providing *quorum*, i.e., locking in state of system
- 3f+1 nodes
  - A quorum must be available, even if f nodes are faulty

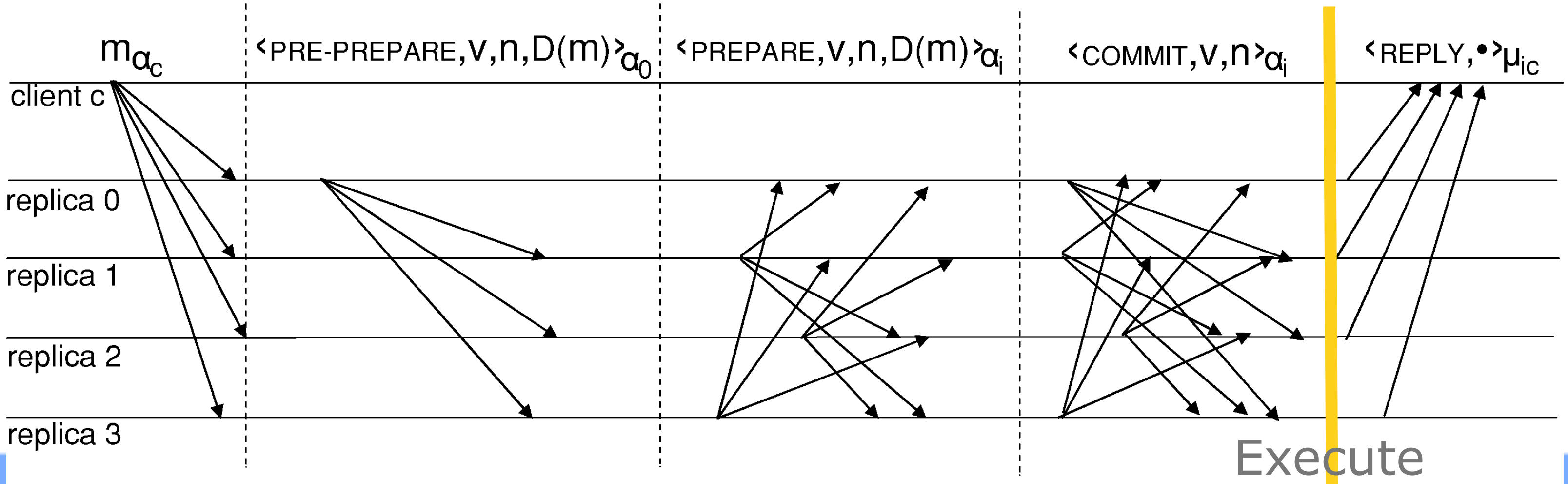
# Properties of a Quorum

- Intersection
  - A gets 2f+1 responses with value x
  - B gets 2f+1 responses with value y
  - Then: x=y because A and B must share ≥1 non-faulty node
- Availability
  - With 3f+1 nodes, 2f+1 are non-faulty



# Overview of Core Algorithm

- Client multicasts request to replicas
- Primary assigns order, backups agree to order, quorum
  - Three phases: pre-prepare, prepare, and commit
- Replicas execute requested operation, reply to client
- Client waits for f+1 responses with *same* value
  - At least one response comes from a non-faulty node



# A View From a Client

- Client multicasts request <REQUEST, o, t, c>$_c$
  - o = operation, t = local timestamp, c = client id
- Client collects replies <REPLY, v, t, c, i, r>$_i$
  - v = view number, i = replica number, r = result
- Client uses result from f+1 correct replies
  - Valid signatures
  - f+1 different values of i
  - But same t and r

# There's Work to Do

- Replica accepts $\langle \text{REQUEST}, o, t, c \rangle_{c}$
  - Ensures signature is valid
  - Logs request to track progress through algorithm

# Pre-Prepare Phase

- Primary multicasts <PRE-PREPARE, v, n, d>$_{\mathbf{p}}$
  - n = sequence number, d = digest of request
  - Proposes commit order for operation
- Backups accept and log pre-prepare message
  - Signature is valid, d matches actual request
  - Backup is in view v
  - Backup has *not* accepted different message for n
- Backups enter prepare phase

# Prepare Phase

- Backups multicast <PREPARE, v, n, d, i>$_i$ to all replicas
- Replicas accept and log prepare messages
  - If signatures, v, n, and d match
- Operation is prepared on replica #i iff
  - #i has pre-prepare msg and 2f matching prepare msgs
- Once prepared, replica does ... ?

# Are We There Yet? No!

- Prepared certificate says “we agree on n for m in v”
  - Still need to agree on order across view changes
- Alternative take
  - If prepared, replica #i knows there is a quorum
    - But messages can be lost etc., so others may not know
    - So, we still need to agree that we have quorum
- Solution: one more phase
  - Once prepared, replica enters commit phase

# Commit Phase

- All replicas multicast <COMMIT, v, n, i>$_i$
  - Also accept and log others' correct commit messages
- Operation is committed on replica #i iff
  - Operation is prepared
  - Replica #i has 2f+1 matching commit messages (incl. own)
- Once committed, replica is ready to perform operation
  - But only after all operations with lower sequence numbers have been performed

# The Log

- So far: log grows indefinitely
  - Clearly impractical
- Now: periodically checkpoint the state
  - Each replica computes digest of state
  - Each replica multicasts digest across replicas
  - 2f+1 such digests represent quorum (lock-in)
    - Can throw away log entries for older state, which is captured in *stable* checkpoint

# The rationale of the three-phase protocol

Divya Sivasankaran

# Three Phase Protocol - Goals

Ensure safety and liveness despite asynchronous nature

- Establish total order of execution of requests (*Pre-prepare* + *Prepare*)
- Ensure requests are ordered consistently across views (*Commit*)

Recall: View is a configuration of replicas with a primary p = v mod |R|

REQUEST → ***PRE-PREPARE*** → ***PREPARE*** → ***COMMIT*** → REPLY

# Three Phases:

- Pre-prepare
  - Acknowledge a ***unique sequence number*** for the request
- Prepare
  - The replicas agree on this sequence number
- Commit
  - Establish total order across views

REQUEST → ***PRE-PREPARE*** → ***PREPARE*** → ***COMMIT*** → REPLY

# Definitions

- Request message m
- Sequence number n
- Signature - $\sigma$
- View - v
- Primary replica - p
- Digest of message D(m) $\rightarrow$ d

# Pre-prepare

Purpose: acknowledge a unique sequence number for the request

- SEND
  - The primary assigns the request a sequence number and broadcasts this to all replicas

- A backup will ACCEPT the message iff
  - d, v, n, $\sigma$ are valid
  - (v,n) has not been processed before for another digest (d)

REQUEST → ***PRE-PREPARE*** → PREPARE → COMMIT → REPLY

# Prepare

Purpose: The replicas agree on this sequence number

After backup i accepts <PRE-PREPARE> message

- SEND
  - multicast a <PREPARE> message acknowledging n, d, i and v

- A replica will ACCEPT the message iff
  - d, v, n, $\sigma$ are valid

REQUEST → PRE-PREPARE → ***PREPARE*** → COMMIT → REPLY

# Prepared

Predicate prepared(m,v,n,i) = T iff replica i

- <PRE-PREPARE> for m has been received
- **2f+1**(incl itself) distinct & valid <PREPARE> messages received

Guarantee

Two **different** messages can never have the same sequence number

i.e., *<u>Non-faulty replicas agree on total order for requests within a view</u>*

REQUEST → PRE-PREPARE → ***PREPARE*** → COMMIT → REPLY

# Commit

Purpose: Establish total order across views

Once prepared(m,v,n,i) = T for a replica i

- Send
  - multicast <COMMIT> message to all replicas

- All replicas ACCEPT the message iff
  - d, v, n, $\sigma$ are valid

REQUEST → PRE-PREPARE → PREPARE → ***COMMIT*** → REPLY

# Committed

Predicate committed(m,v,n,i) = T iff replica i

- prepared(m,v,n,i) = T
- **2f+1**(incl itself) distinct & valid <COMMIT> messages received

Guarantee

Total ordering across views (*Proof will be shown later*)

REQUEST → PRE-PREPARE → PREPARE → ***COMMIT*** → REPLY

# Executing Requests

Replica i executes request iff

- committed(m,v,n,i) = T
- All requests with lower seq# are already executed

Once executed, the replicas will directly send <REPLY> to the client

But, what if the primary is faulty? How can we ensure the system will recover?

REQUEST → PRE-PREPARE → PREPARE → COMMIT → ***REPLY***

# View Change

Irvan

# View Change

All is good if primary is good

But everything changed when primary is faulty...

# Problem (Case 1)

Sequence number 1: **INSERT (APPLE) INTO FRUIT**

Sequence number 4: **INSERT (PEAR) INTO FRUIT**

Sequence number 5: **SELECT * FROM FRUIT**

The replica will be stuck waiting for request with sequence number 2...

# View Change Idea

- Whenever a lot of non-faulty replicas detect that the primary is faulty, they together begin the *view-change operation*.
    - More specifically, if they are stuck, they will suspect that the primary is faulty
    - The primary is detected to be faulty by using timeout
    - **Thus this part depends on the synchrony assumption**
    - They will then change the view
        - The primary will change from replica p to replica (p+1)%|R|

# Initiating View Change

- Every replica that wants to begin a view change sends a <VIEW-CHANGE> message to EVERYONE
  - Includes the current state so that all replicas will know which requests haven't been committed yet (due to faulty primary).
  - List of requests that was **prepared**
- When the new primary receives **2f+1** <VIEW-CHANGE> messages, it will begin the view change

# The Corresponding Message

Sequence number 1: **INSERT (APPLE) INTO FRUIT**

Sequence number 4: **INSERT (PEAR) INTO FRUIT**

Sequence number 5: **SELECT * FROM FRUIT**

Replica 1 <VIEW-CHANGE> message:

<VIEW-CHANGE, **SEQ1: INSERT (APPLE)**, **SEQ4: INSERT (PEAR)**, **SEQ5: SELECT ***>

# View-Change and Correctness

1) New primary gathers information about which requests that need committing

- This information is included in the <VIEW-CHANGE> message
- All replicas can also compute this since they also receive the <VIEW-CHANGE> message
  - Will avoid a faulty new primary making the state inconsistent

2) New primary sends <NEW-VIEW> to all replicas

3) All replicas perform 3 phases on all the requests again

# Example

<VIEW-CHANGE, **SEQ1: INSERT (APPLE)**, **SEQ4: INSERT (PEAR)**, **SEQ5: SELECT ***>
<VIEW-CHANGE, **SEQ2: INSERT (KIWI)**, **SEQ4: INSERT (PEAR)**, **SEQ5: SELECT ***>

Sequence number 1: **INSERT (APPLE) INTO FRUIT**
Sequence number 2: **INSERT (KIWI) INTO FRUIT**

Sequence number 4: **INSERT (PEAR) INTO FRUIT**
Sequence number 5: **SELECT * FROM FRUIT**

...Will still get stuck on sequence number 3?

# Example

<VIEW-CHANGE, **SEQ1: INSERT (APPLE)**, **SEQ4: INSERT (PEAR)**, **SEQ5: SELECT ***>
<VIEW-CHANGE, **SEQ2: INSERT (KIWI)**, **SEQ4: INSERT (PEAR)**, **SEQ5: SELECT ***>

Sequence number 1: **INSERT (APPLE) INTO FRUIT**
Sequence number 2: **INSERT (KIWI) INTO FRUIT**
Sequence number 3: PASS
Sequence number 4: **INSERT (PEAR) INTO FRUIT**
Sequence number 5: **SELECT * FROM FRUIT**

Sequence numbers with missing requests are replaced with a “no-op” operation - a “fake” operation.

# State Recomputation

- Recall the new primary needs to recompute which requests need to be committed again.
- Redoing all the requests is expensive
- Use checkpoints to speed up the process
  - After every 100 sequence number, all replicas save its current state into a checkpoint
  - Replicas should agree on the checkpoints as well.

# Other types of problems...

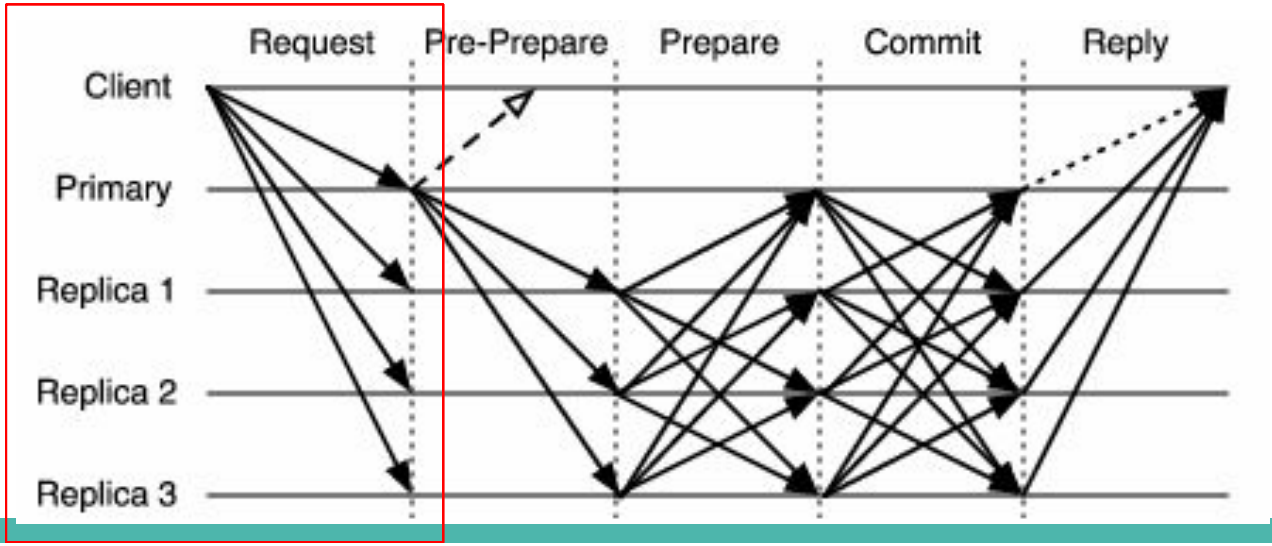
- What happens if the new primary is also faulty?
  - Use another timeout in the view-change
    - When the timeout expires, another replica will be chosen as primary
    - Since there are at most f faulty replicas, the primary can be consecutively faulty for at most f times
- What happen if a faulty primary picks a huge sequence number? For example, 10,000,000,000?
  - The sequence number must lie within a certain interval
  - This interval will be updated periodically

# Problem (Case 2)

- Client sends request to primary
- Primary doesn’t forward the request to the replicas…

# Client Full Protocol

- Client sends a request to the primary that they knew
  - The primary may already change, this will be handled
- If they do not receive reply within a period of time, it broadcast the request to all replicas



# Replica Protocol

- If a replica receive a request from a client but not from the primary, they send the request to the primary,
- If they still do not receive reply from primary within a period of time, they begin view-change

# Some Correctness

To convince you that the view-change protocol preserves safety, we will show you one of the key proofs

# Correctness of View-Change

- We will show that if at any moment a replica has **committed** a request, then this request will ALWAYS be re-committed in the view-change

# Proof Sketch

- Recall that a request will be re-committed in the view-change if they are included in at least one of the <VIEW-CHANGE> messages
- A **committed** request implies there are at least f+1 non-faulty replicas that ***prepared*** it.
- Proof:
    - There are 2f+1 <VIEW-CHANGE> messages
    - For any request **m** that has been committed, there are f+1 non-faulty replicas that ***prepared* m**
    - Since |R| = 3f+1, at least one non-faulty replicas mu prepared **m** and sent the <VIEW-CHANGE> message

# Notes

- This safety lemma is one of the reasons we need to have a three phase protocol instead of two phase protocols
    - In particular, if we only have two phases, we cannot guarantee that if a request has been committed, it will be prepared by a majority of non-faulty replicas. Thus it's possible that an committed request will not be re-committed… -- violates safety.

# Optimization

- Reduce the cost of communication
- Reduce message delays
- Improve the performance read-only operations
- ……

# PBFT inspires much follow-on work

- BASE: Using abstraction to improve fault tolerance, R. Rodrigo et al, SOSP 2001
- R.Kotla and M. Dahlin, High Throughput Byzantine Fault tolerance. DSN 2004
- J. Li and D. Mazieres, Beyond one-third faulty replicas in Byzantine fault tolerant systems, NSDI 07
- Abd-El-Malek et al, Fault-scalable Byzantine fault-tolerant services, SOSP 05
- J. Cowling et al, HQ replication: a hybrid quorum protocol for Byzantine Fault tolerance, OSDI 06
- Zyzzyva: Speculative Byzantine fault tolerance SOSP 07
- Tolerating Byzantine faults in database systems using commit barrier scheduling SOSP 07
- Low-overhead Byzantine fault-tolerant storage SOSP 07
- Attested append-only memory: making adversaries stick to their word SOSP 07

# Slides (Re-)used in This Talk

- Loi Luu, Hung Dang, Divya Sivasankaran, Irvan, Zheyuan Gao (NUS);
- Dan Ports (UW)
- Jinyang Li (NYU)
- Robert Grimm (NYU)