Primary/Backup

CS6450: Distributed Systems

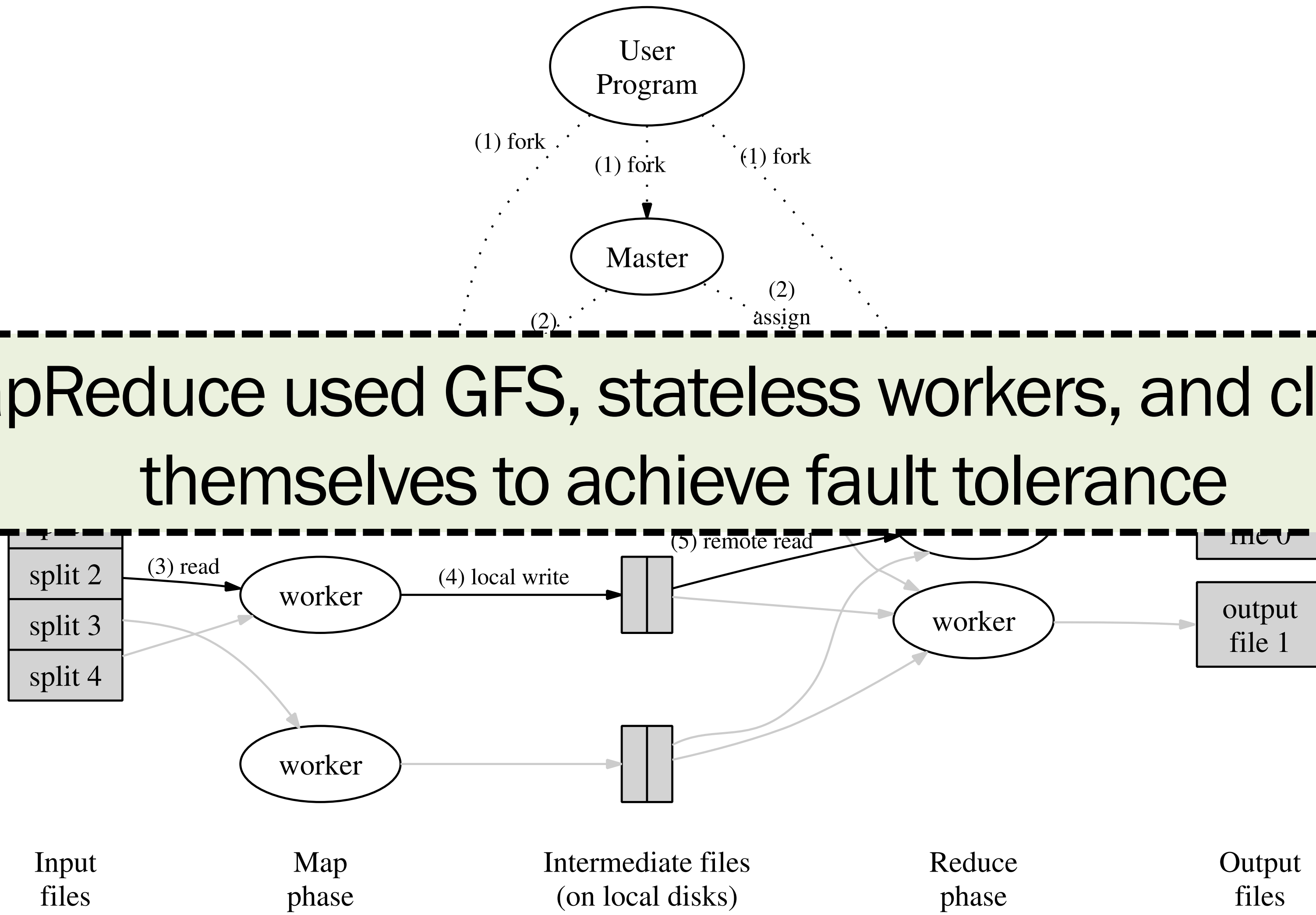
Lecture 4

Ryan Stutsman

Takeaways

- State machine replication for synchronizing multiple replica of an object
  - Order all operations, replicate log, feed (deterministic) operations to all replicas of an object
- Linearizability: appearance of totally ordered atomic application of all client ops consistent with real-world timing of request/response
  - Lets us reason with precondition/postconditions as if object was local and unreplicated
  - Easy to preserve application invariants
- P-B: choose a primary and backup
  - Primary orders operations and replicates them to backups
  - Backups apply operations in order dictated by primary
  - View service/coordinator chooses new primary from up-to-date backups

Simplified Fault Tolerance in MapReduce



MapReduce used GFS, stateless workers, and clients themselves to achieve fault tolerance

Preview of where we’re heading

Replication and Consistency

- Replicate for fault-tolerance and/or performance

- But, now modification is hard
  - Updates to any single copy result in inconsistency

- Hence, replication, consistency, and fault tolerance are all tightly intertwined
  - And, sadly, concurrency too!

Building Correct Stateful Systems?

- NFS wanted performance, so it gave up on consistency
- Given some application-level invariant, if the application shares state over NFS, will the invariant hold?
  - Proof is easy if shared state is read-only
  - Proof may be easy if each application process accesses independent pieces of state
  - Proof seems nearly impossible otherwise
- How do we build systems that maintain application-level invariants?
- What invariants/techniques do those systems need to rely on internally?

Brute Force Proof?

- Given some application-level invariants we could try to prove something knowing the details of the underlying systems it relies on
- Basically intractable
  - Systems are massive, fault scenarios are numerous
  - Cross product of application-level states and states of the underlying systems
- Plus, no "narrow waist", would need to fresh proof effort for each application x system combination

Then what? A Common Approach

- Assume some abstract data type T
  - (Just a type and some methods on that type.)
- Assume that a developer has reasoned about the invariants of that ADT *locally*
- Now, imagine the developer wants to **replicate** instances of that T for fault tolerance
- Can we create systems such that if that ADT is exported over the network its local invariants hold?
  - Yes 👍
- Do we need an application-specific protocol to do it?
  - No 👍
- Can this work even in the presence of faults/crashes?
  - Yes 👍

Piece #1: Replicated State Machines

- Order all operations targeted to an object
- Basically, just log them in order as they come in
- Replicate that log
- Once a log entry is safely replicated, feed the logged operation to the object
- If operations are deterministic, object instances will be “virtually” in sync

- Great, but how does this lead to semantics equivalent to local?

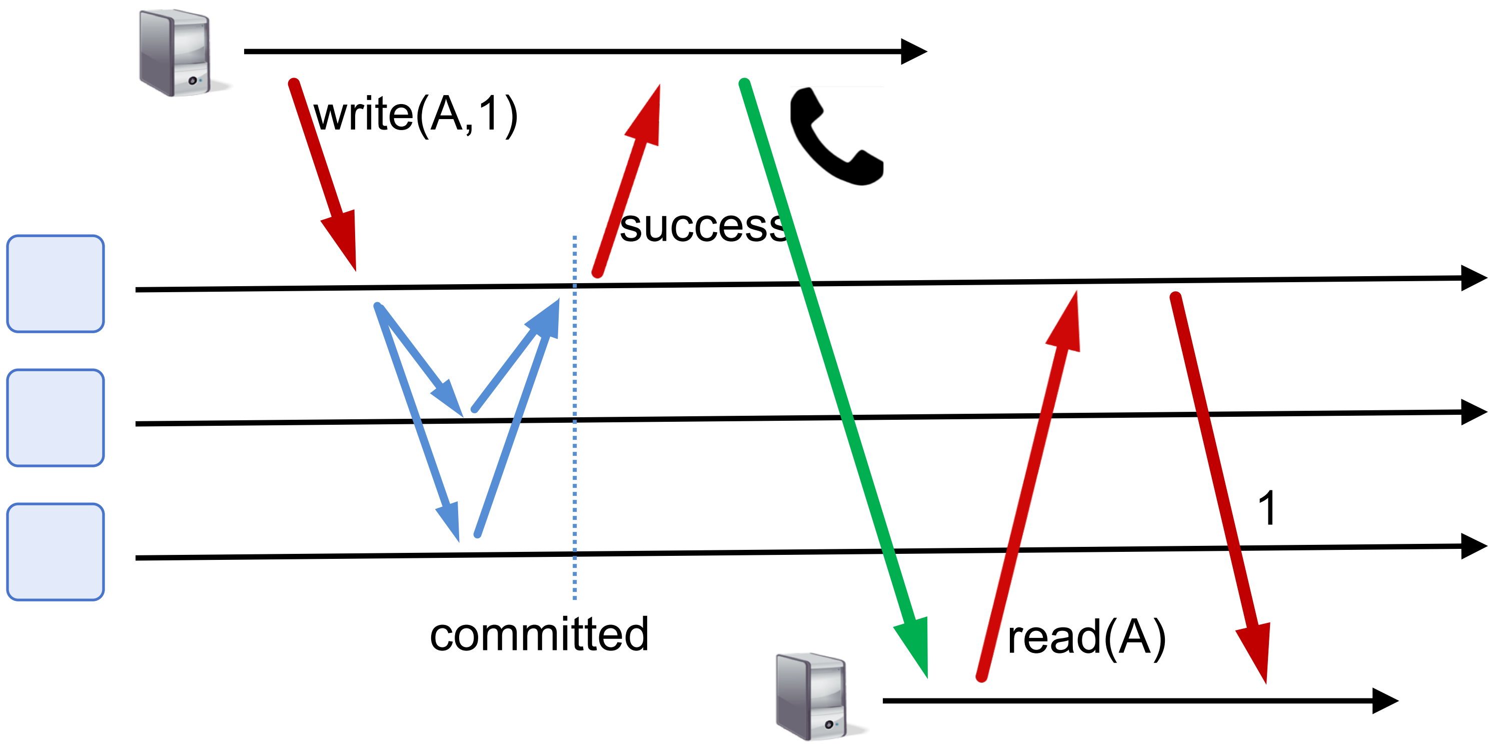
Piece #2: Linearizability (Herlihy and Wing '91)

All operations on an object *appear as if* they are executed *atomically* in *some* (single, total) sequential order between invocation and response

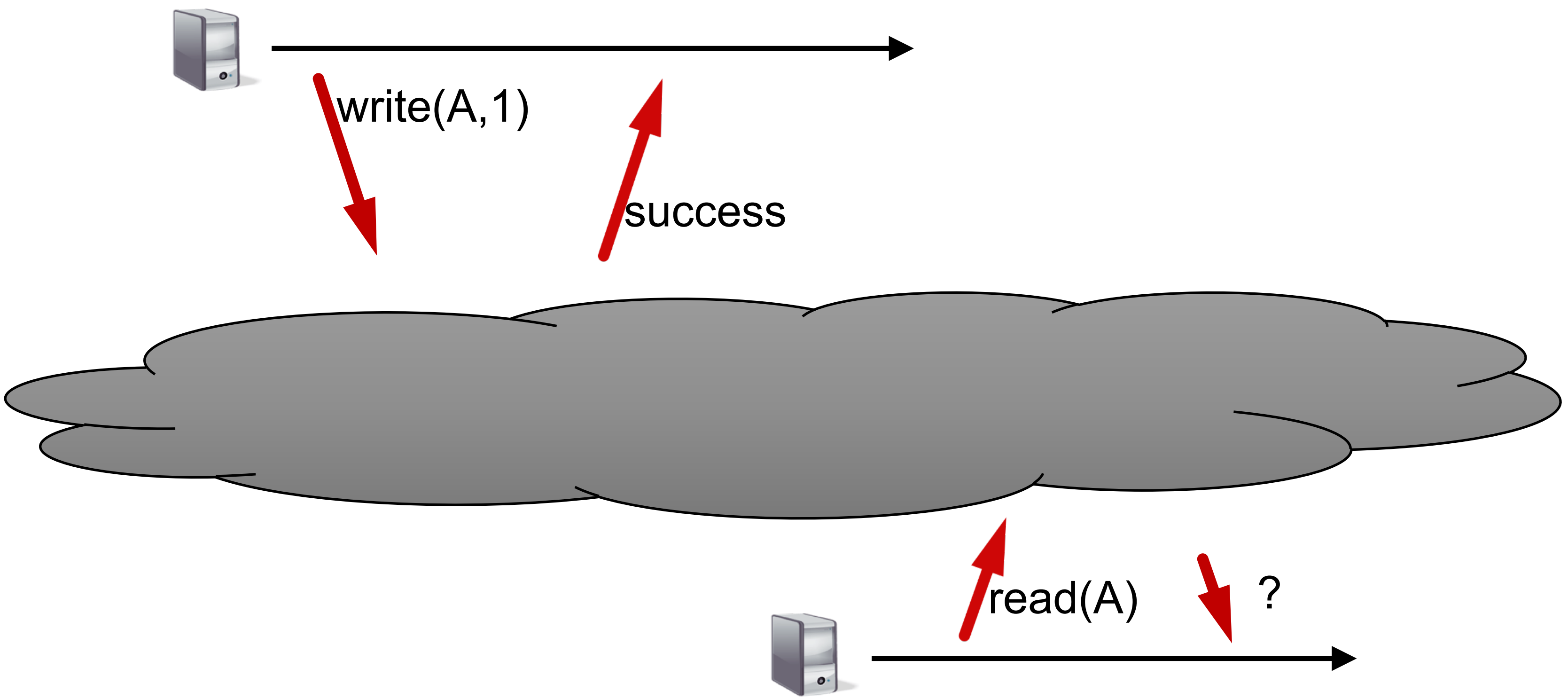
1. Note: consistent with each client's own local ordering
2. Note: consistent with invocation and response
   - Preserves real-time guarantee
   - If X completes before Y starts, then X precedes Y in sequence

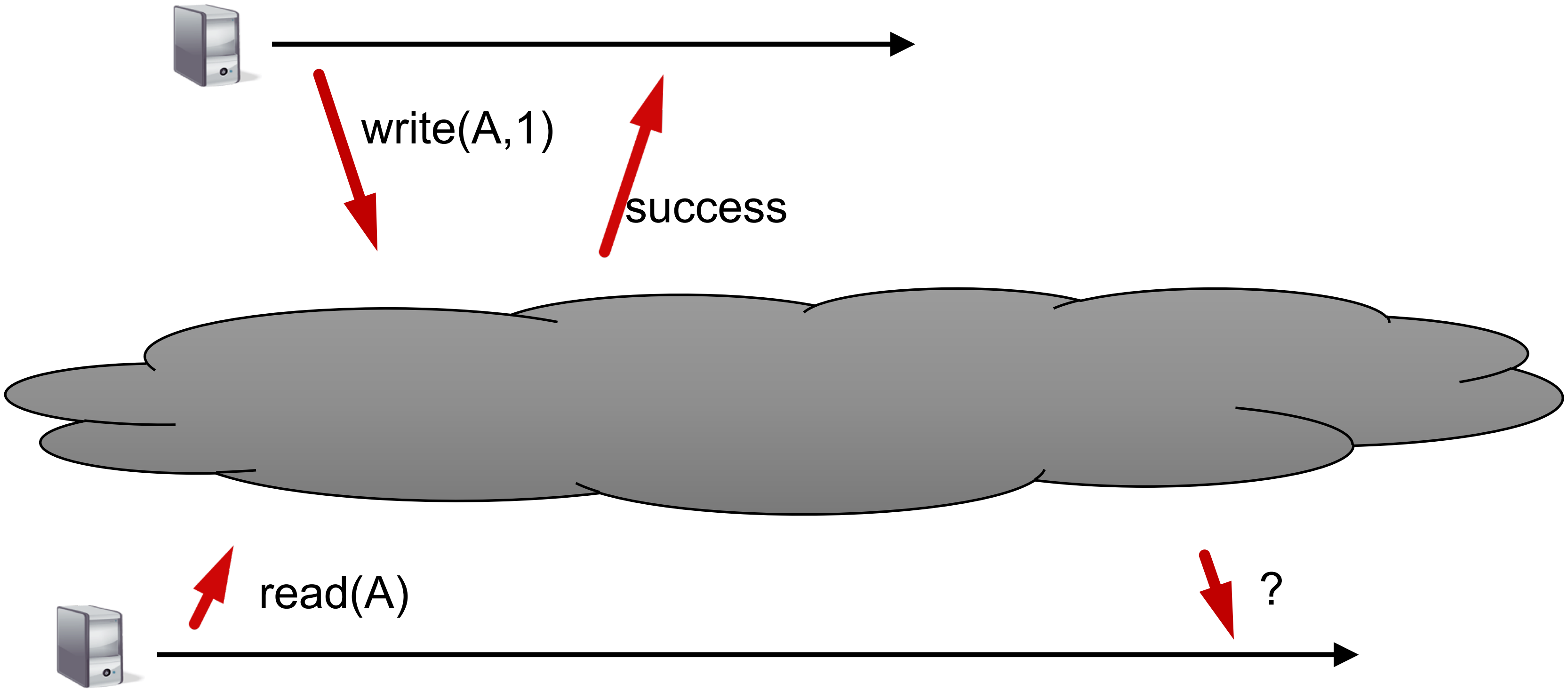
- Once write completes, all later reads (by wall-clock start time) should return value of that write or value of later write.
- Once read returns particular value, all later reads should return that value or value of later write.

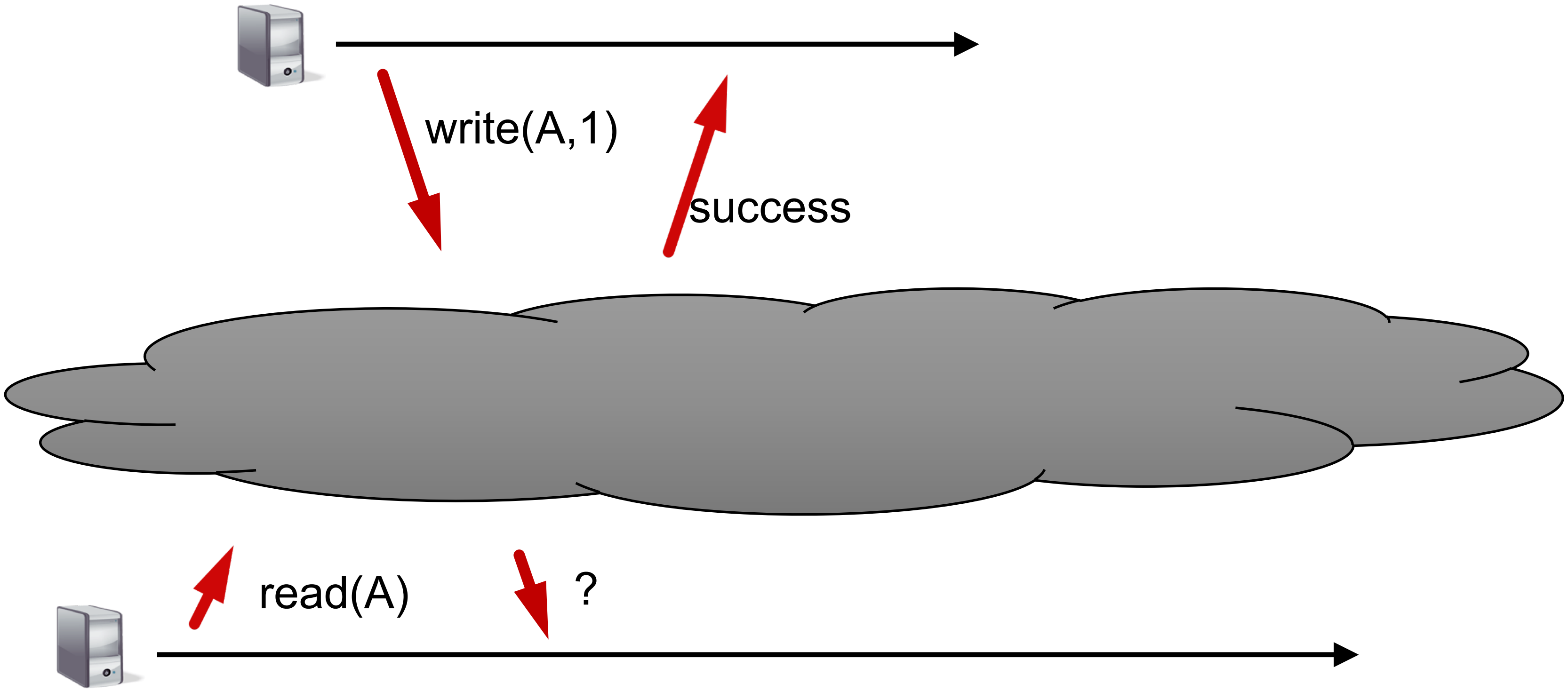
Intuition: Real-time ordering

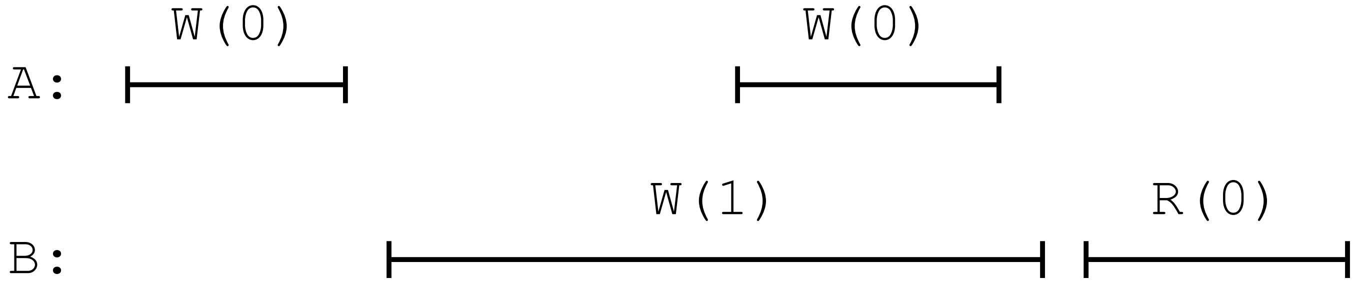


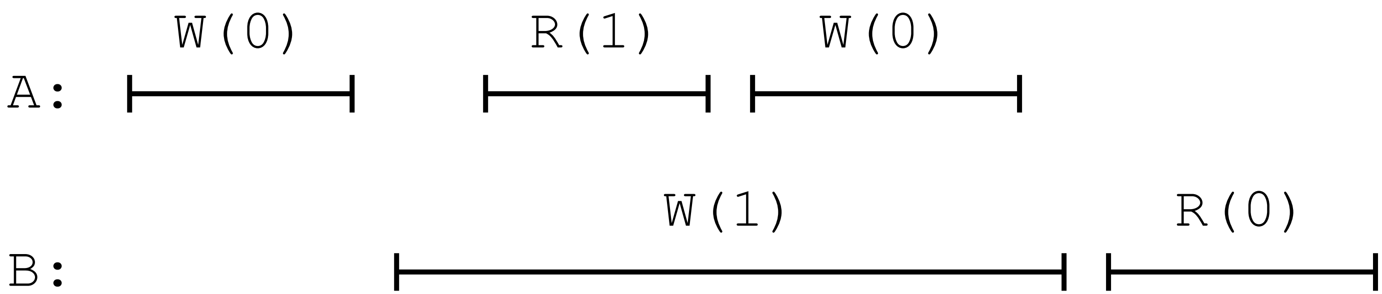
- Once write completes, all later reads (by wall-clock start time) should return value of that write or value of later write.
- Once read returns particular value, all later reads should return that value or value of later write.

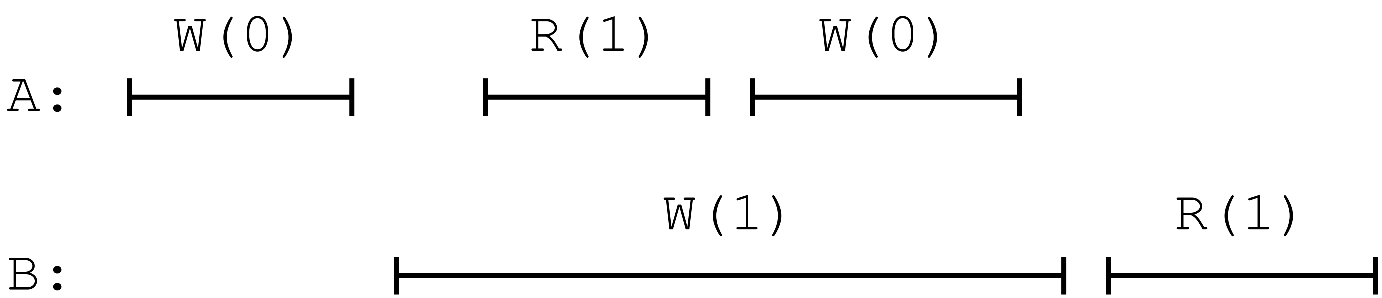

write(A,1)
success
read(A)
?


write(A,1)
success
read(A)
?


write(A,1)
success
read(A)
?


A:
W(0)
W(0)
B:
W(1)
R(0)


A:
W(0)
R(1)
W(0)
B:
W(1)
R(0)


A:
W(0)
R(1)
W(0)
B:
W(1)
R(1)

The Dream

- Sequence all operations in a log
- Replicate logged operations safely, even if the system is in a state of partial failure
- Apply them in order and atomically to replicas of an object

- Result: our single-method, local reasoning works, but our object is more reliable than a single-node object

Rewinding: Primary-Backup

Plan

1. Introduction to Primary-Backup replication
2. If we have time: VMWare's fault-tolerant virtual machine

Primary-Backup: Goals

- **Mechanism:** Replicate and separate servers

- **Goal #1:** Provide a highly reliable service
  - Despite some server and network failures
    - Continue operation after failure

- **Goal #2:** Servers should behave just like a single, more reliable server

State machine replication

- **Any server** is essentially a *state machine*
  - Set of (key, value) pairs is **state**
  - Operations **transition** between states

- Need an op to be executed on all replicas, or none at all
  - *i.e.*, we need **distributed** all-or-nothing atomicity
  - If op is deterministic, replicas will end in same state

- **Key assumption:** Operations are deterministic
  - This assumption can be (carefully) relaxed

Primary-Backup (P-B) approach

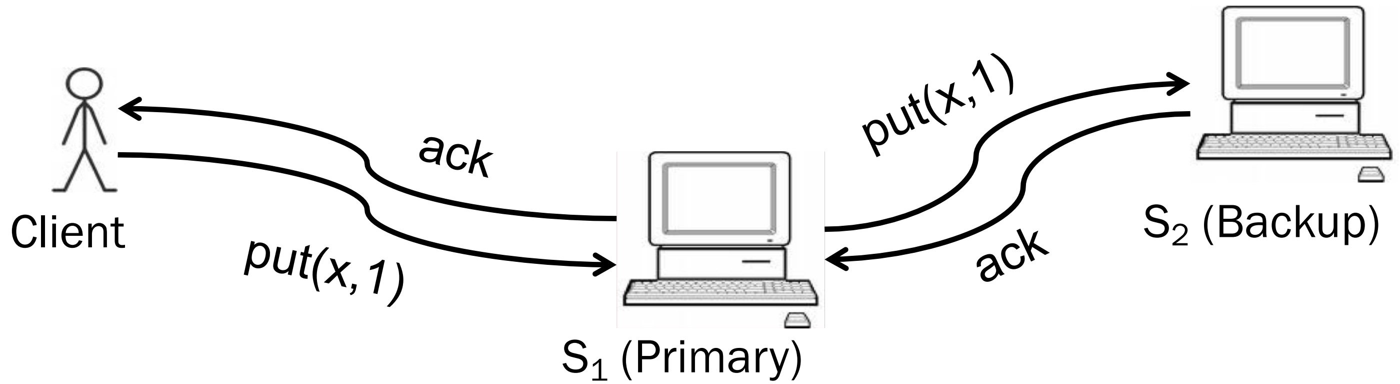
- Nominate one server the *primary,* call the other the *backup*
  - Clients send all operations (get, put) to current primary
  - The primary **orders** clients' operations
- Should be only **one primary at a time**

Need to keep clients, primary, and backup in sync: who is primary and who is backup

Challenges

- Network and server **failures**
- Network **partitions**
  - Within each network partition, near-perfect communication between servers
  - Between network partitions, **no communication between servers**

Primary-Backup (P-B) approach



1. Primary logs the operation locally

2. Primary sends operation to backup and waits for ack
   - Backup performs or just adds it to its *log*

3. Primary performs op and **acks** to the client
   - After backup has applied the operation and ack'ed

View server

- A *view server* decides who is primary, who is backup
  - Clients and servers depend on view server
    - Don’t decide on their own (might not agree)

- Challenge in designing the view service:
  - Only want one primary at a time
  - Careful protocol design needed

- For now, **assume** view server never fails

Monitoring server liveness

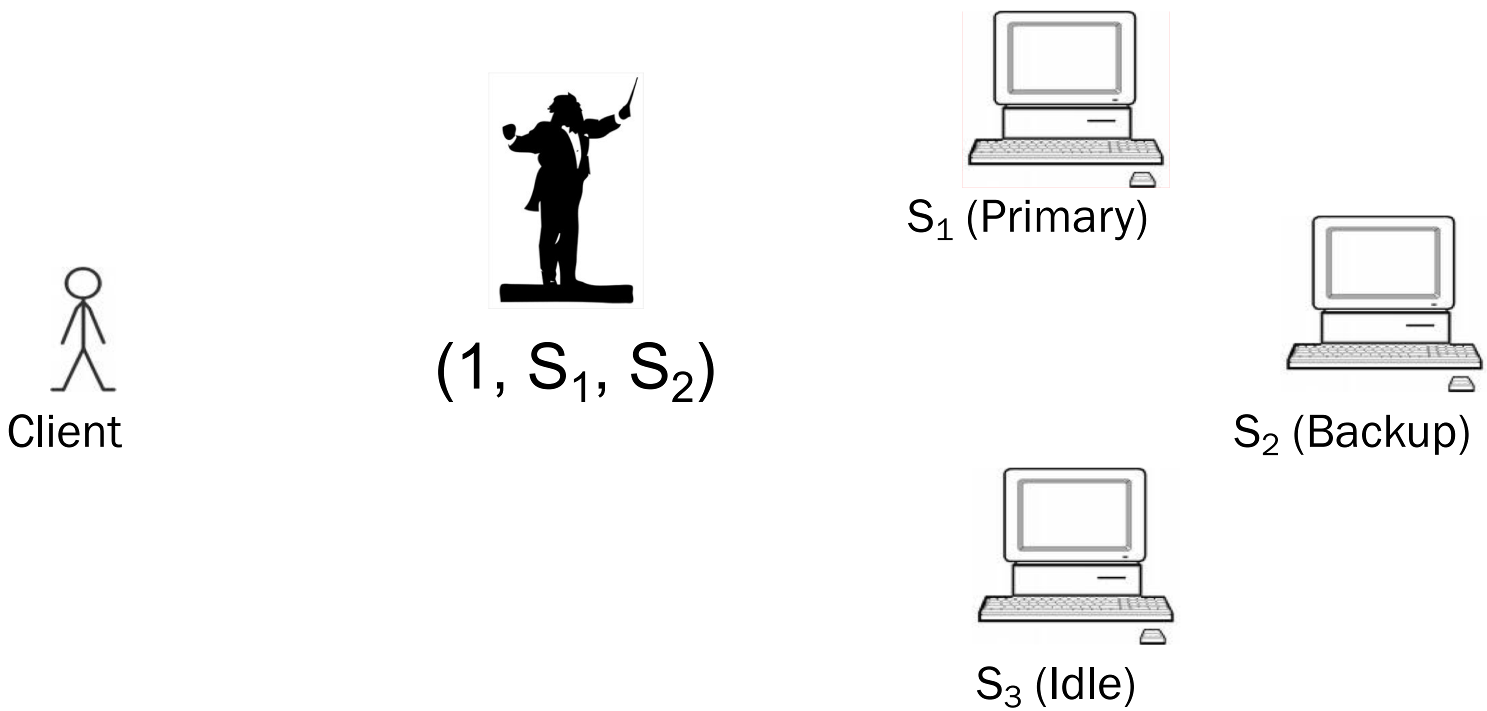
- Each replica periodically *pings* view server
  - View server declares replica *dead* if it missed N pings in a row
  - Considers replica *alive* after single ping

Monitoring server liveness

- Each replica periodically *pings* view server
  - View server declares replica *dead* if it missed N pings in a row
  - Considers replica *alive* after single ping
- *Can a replica **be alive but declared "dead"** by view server?*
  - Yes, if network failure or partition
  - Common approach: use leases if clock skew assumption is ok

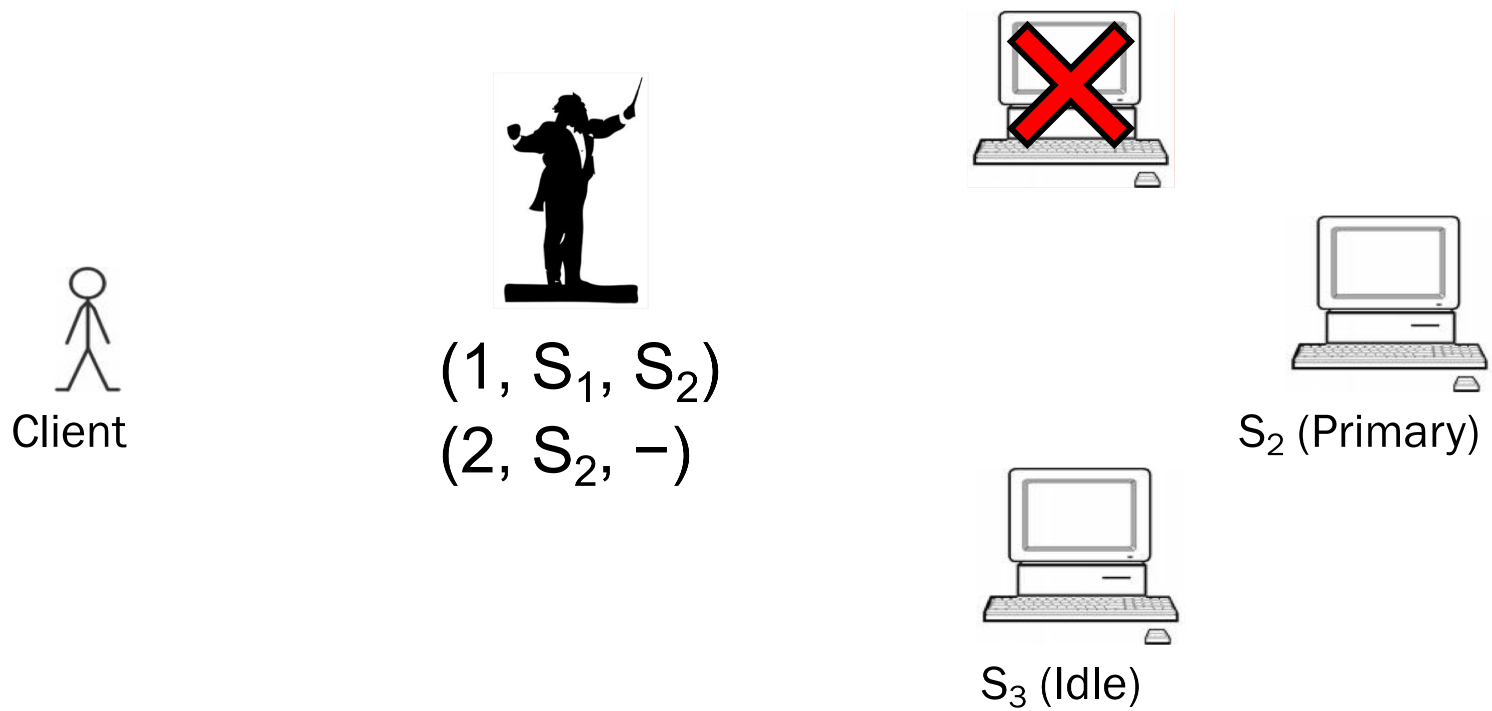
The view server decides the current view

- *View* = (view #, primary server, backup server)



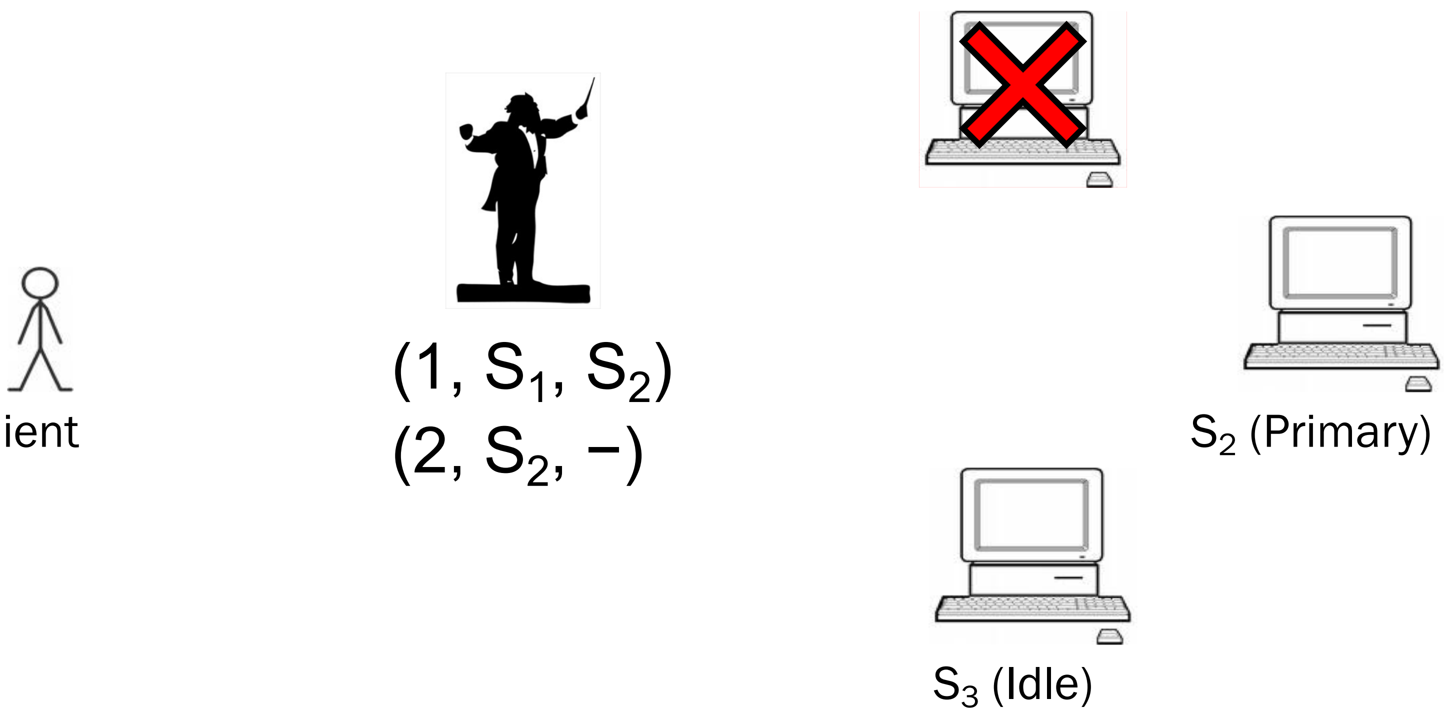
The view server decides the current view

- *View* = (view #, primary server, backup server)



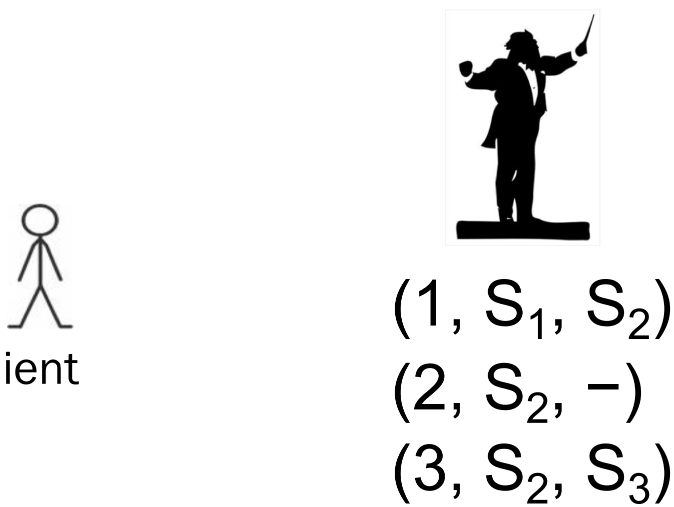
The view server decides the current view

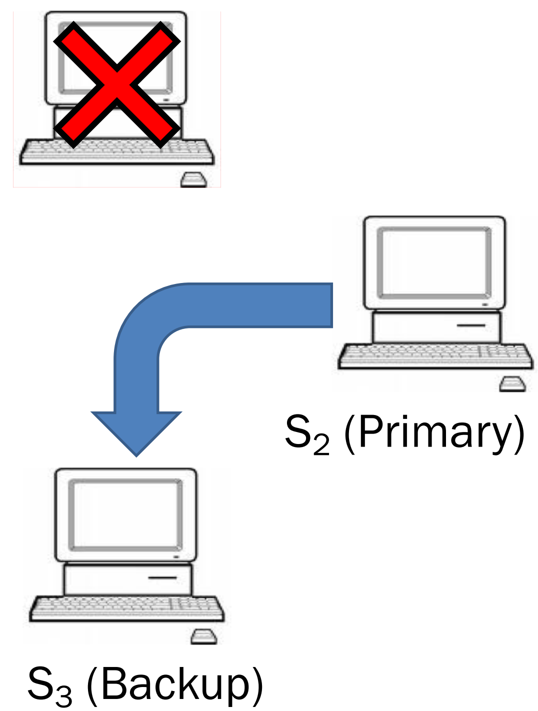
- *View* = (view #, primary server, backup server)



The view server decides the current view

- *View* = (view #, primary server, backup server)





Challenge: All parties make their own local decision of the current view number

Agreeing on the current view

- In general, any number of servers can ping view server

- Okay to have a view with a primary and **no backup**

- Want everyone to **agree** on the **view number**
  - **Include** the view # in RPCs between all parties

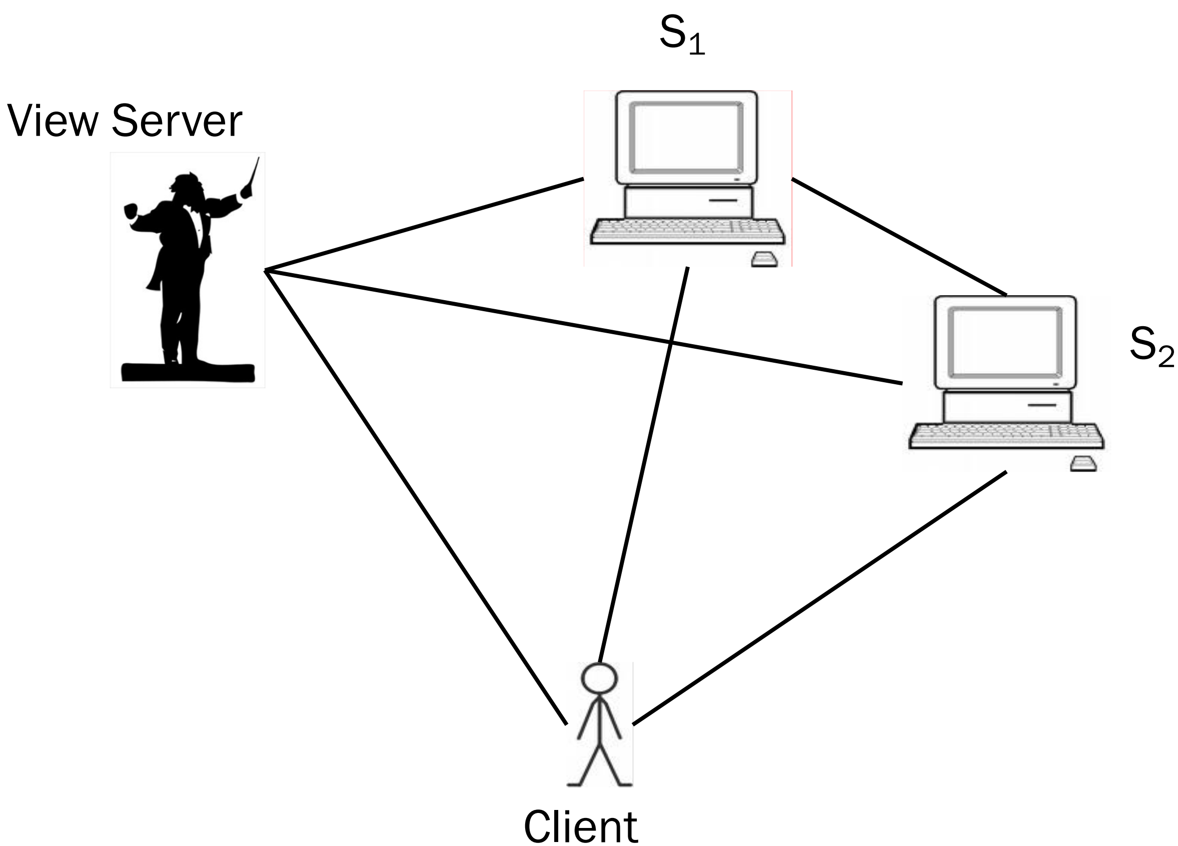
Transitioning between views

- *How do we ensure new primary has up-to-date state?*
  - Only promote a previous backup
    - *i.e.,* don't make a previously-idle server primary
  - State transfer can take awhile, so may take time for backup to take up role

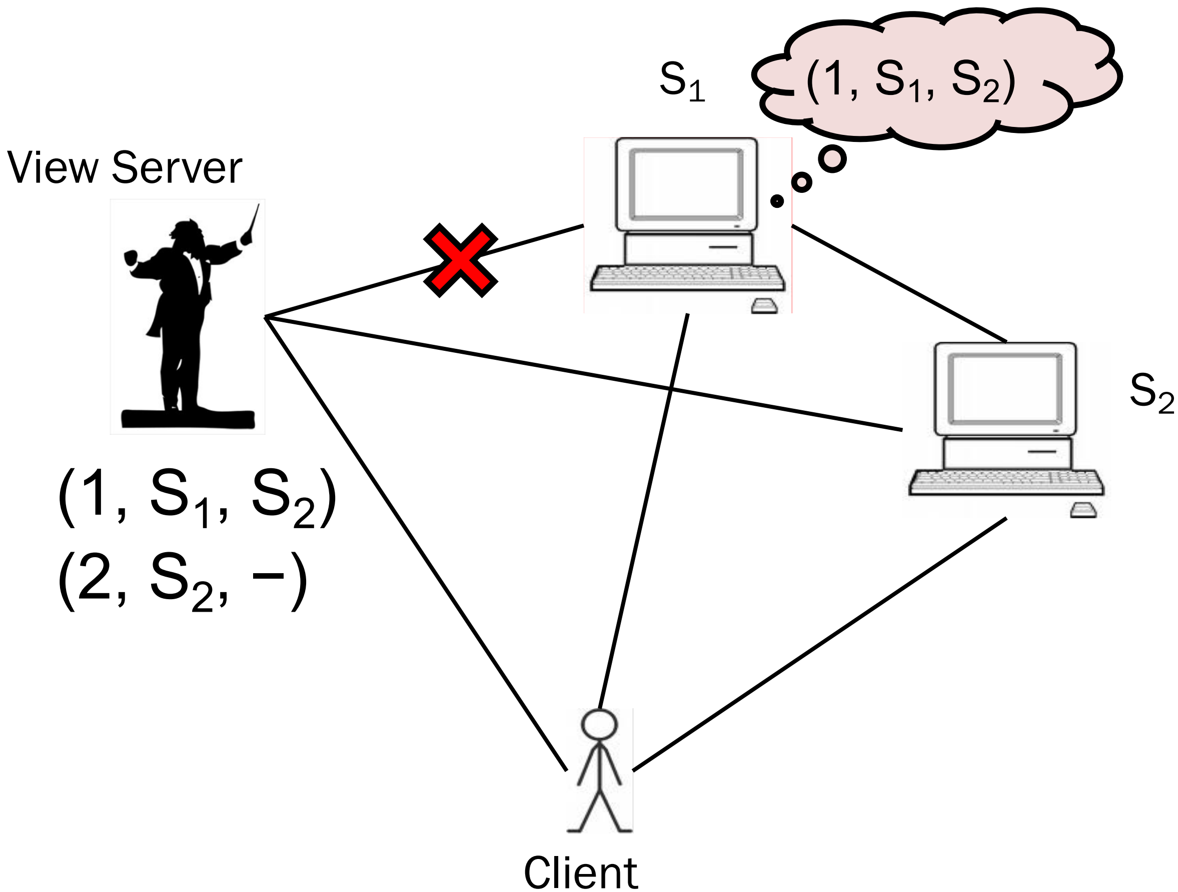
- *How does view server know whether backup is up to date?*
  - View server sends *view-change* message to all
  - New primary **must ack new view** once new backup is up-to-date

  - View server cannot move to (another) new view until ack
    - Even if new primary has or <u>appears</u> to have failed

Split Brain?

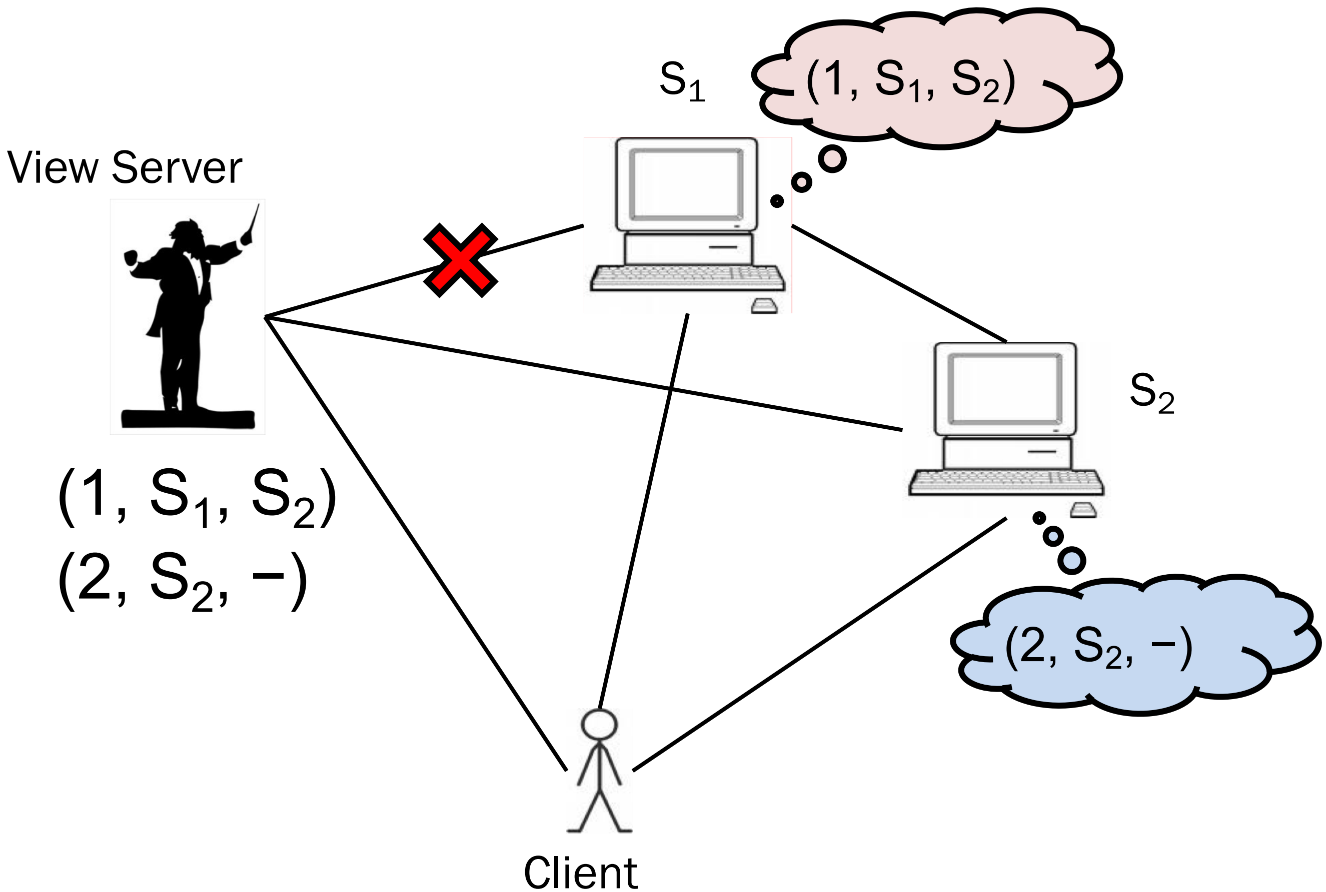


Split Brain?



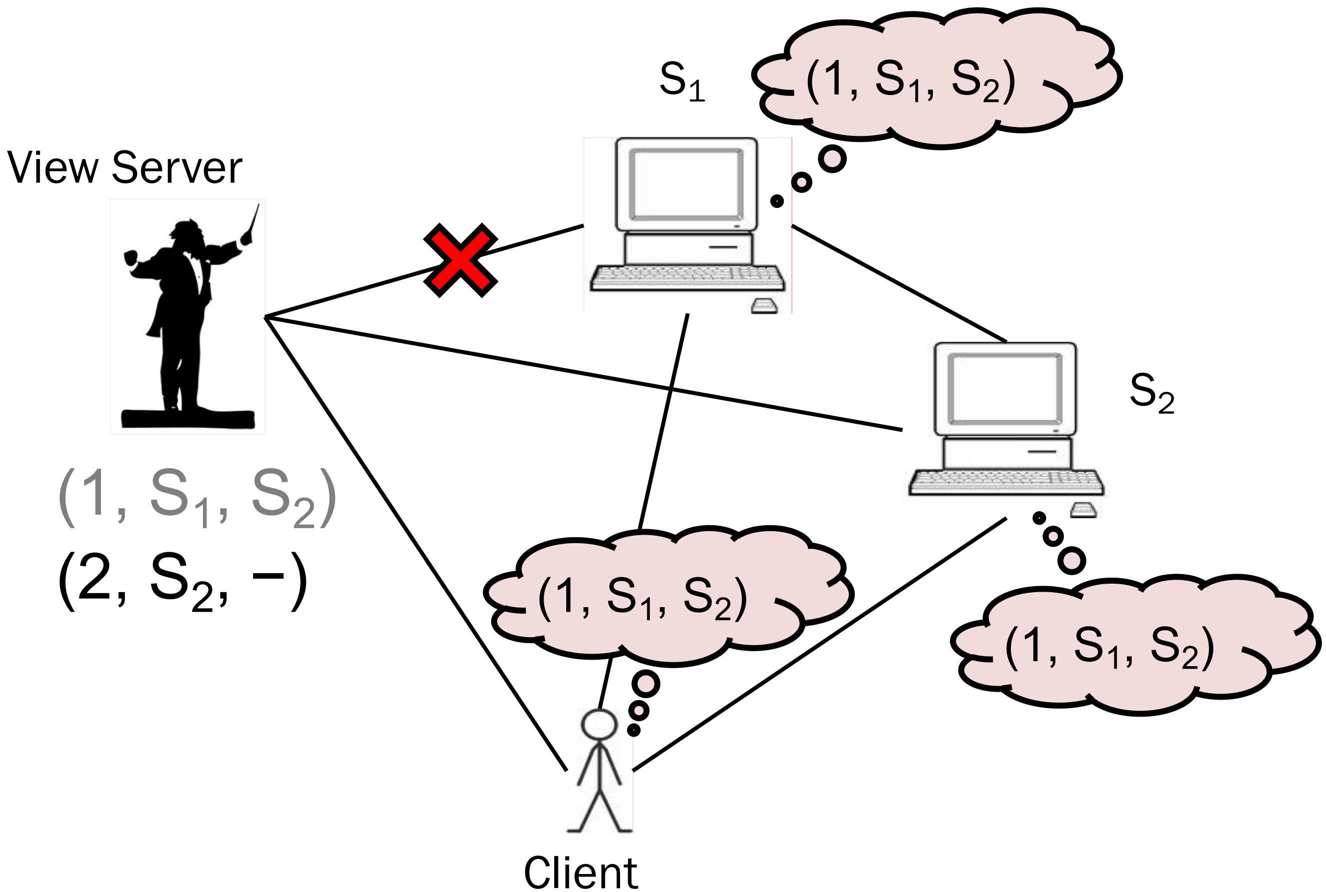
$(1, S_1, S_2)$
$(2, S_2, -)$

Split Brain?

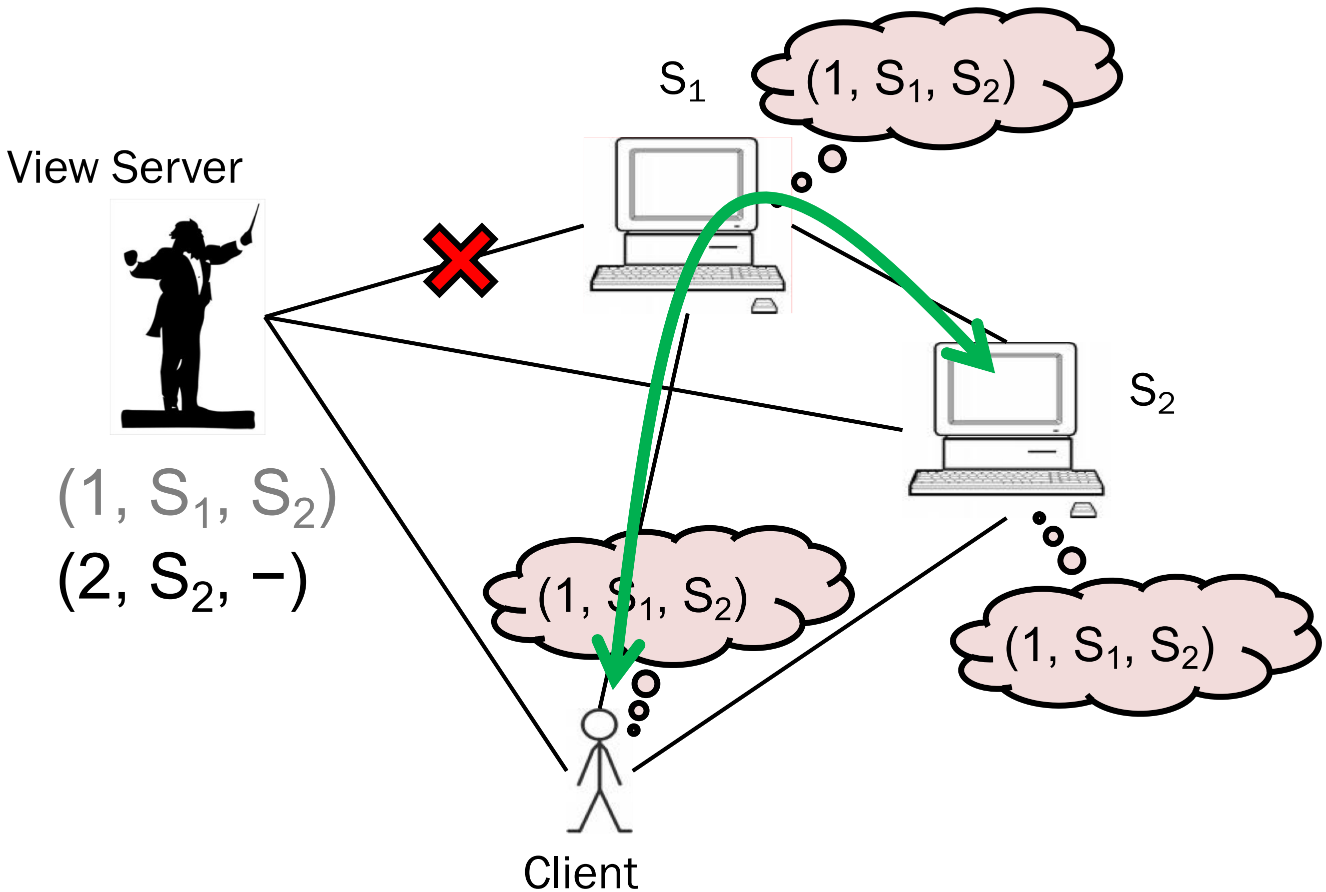


$(1, S_1, S_2)$
$(2, S_2, -)$

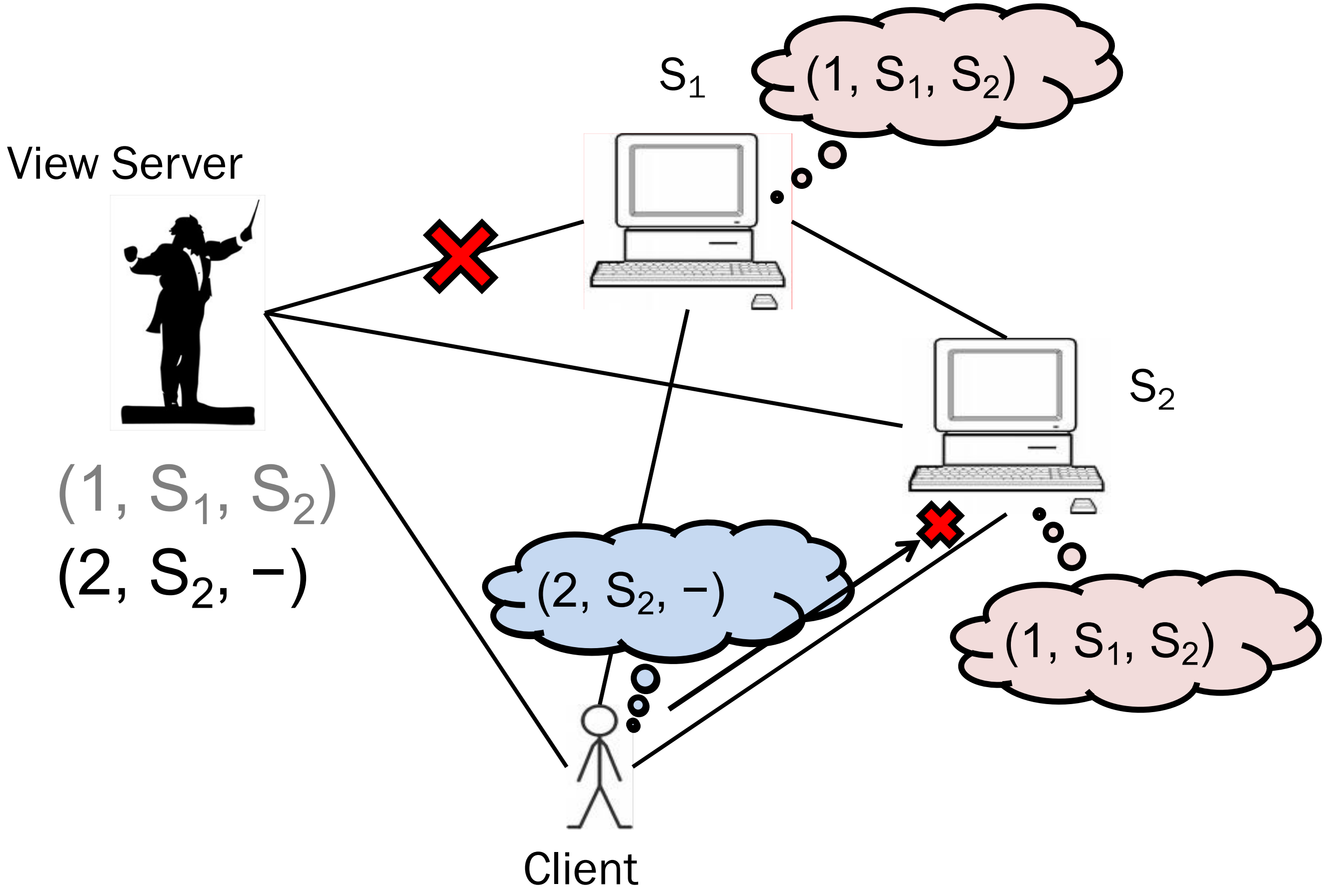
Server S_2 in the old view



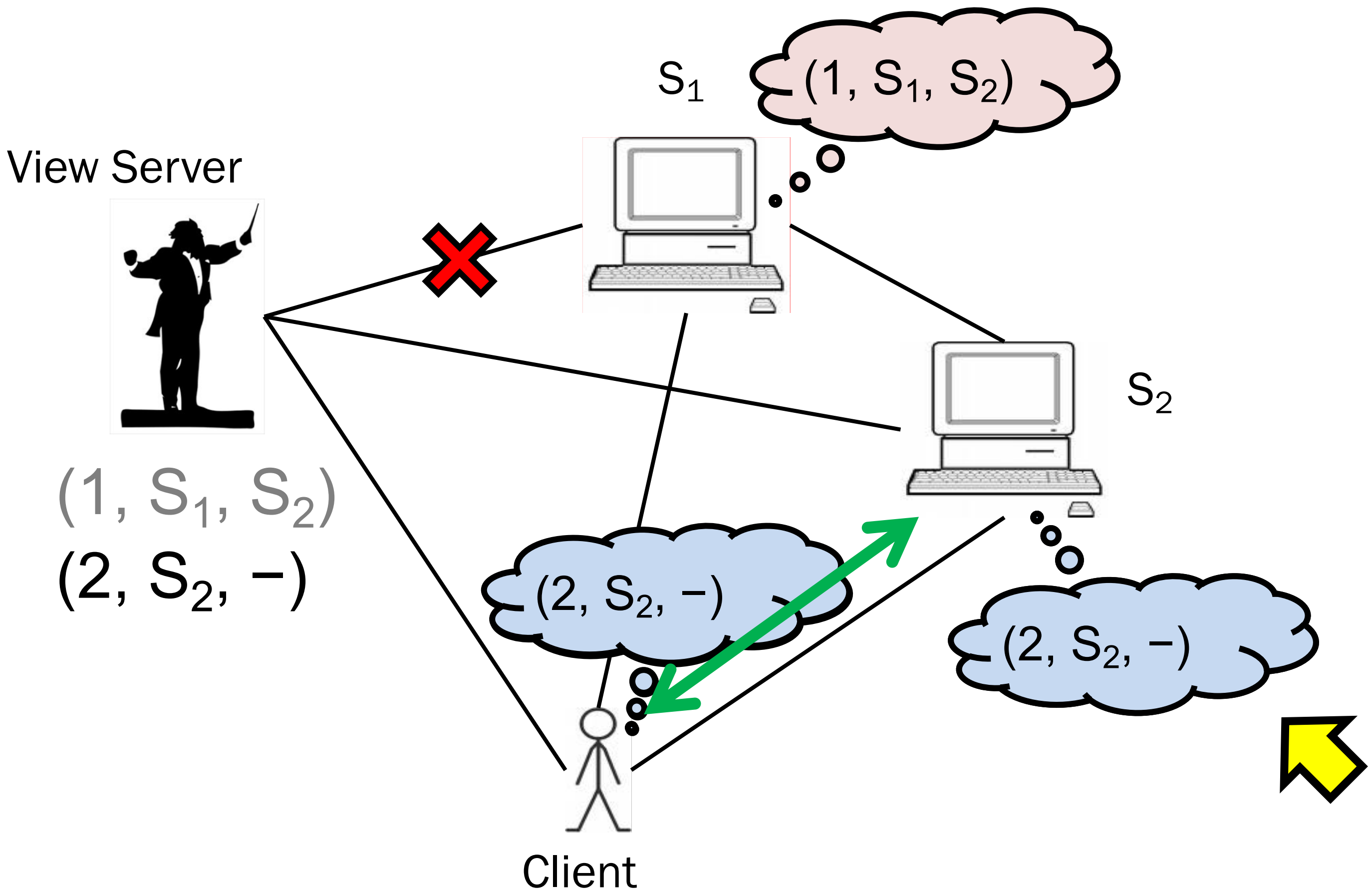
Server S_2 in the old view



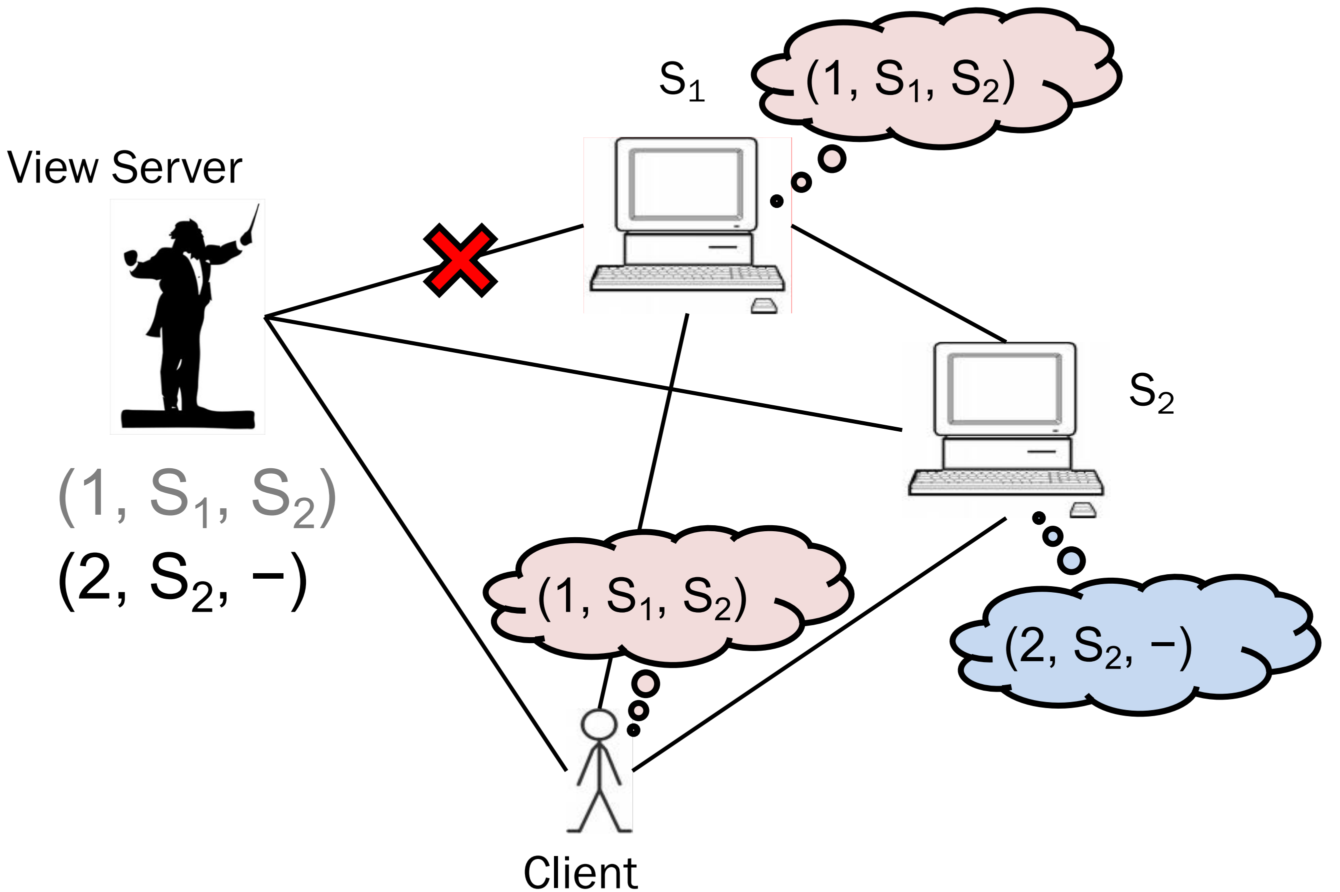
Server S_2 in the old view



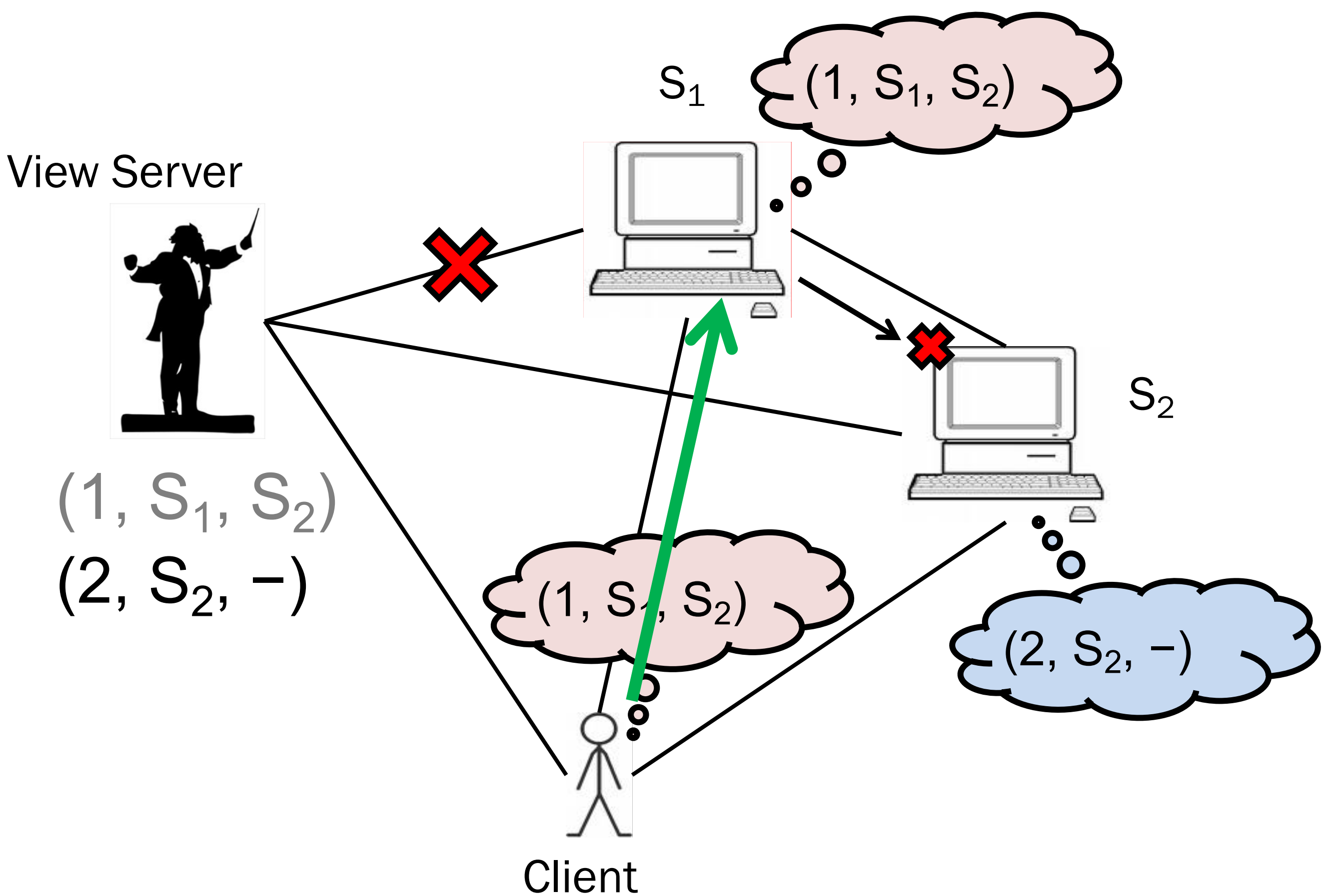
Server S_2 in the old view



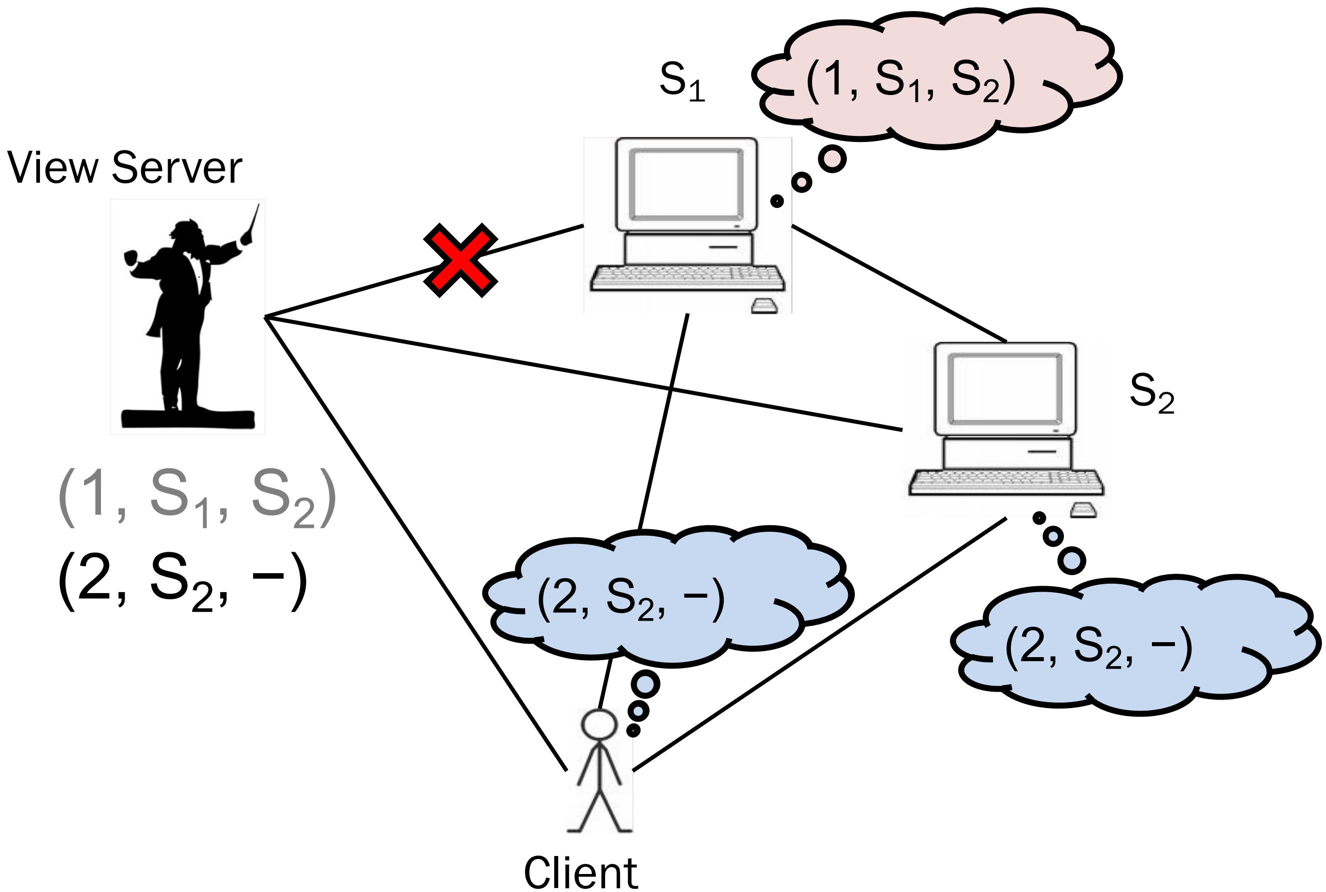
Server S_2 in the new view



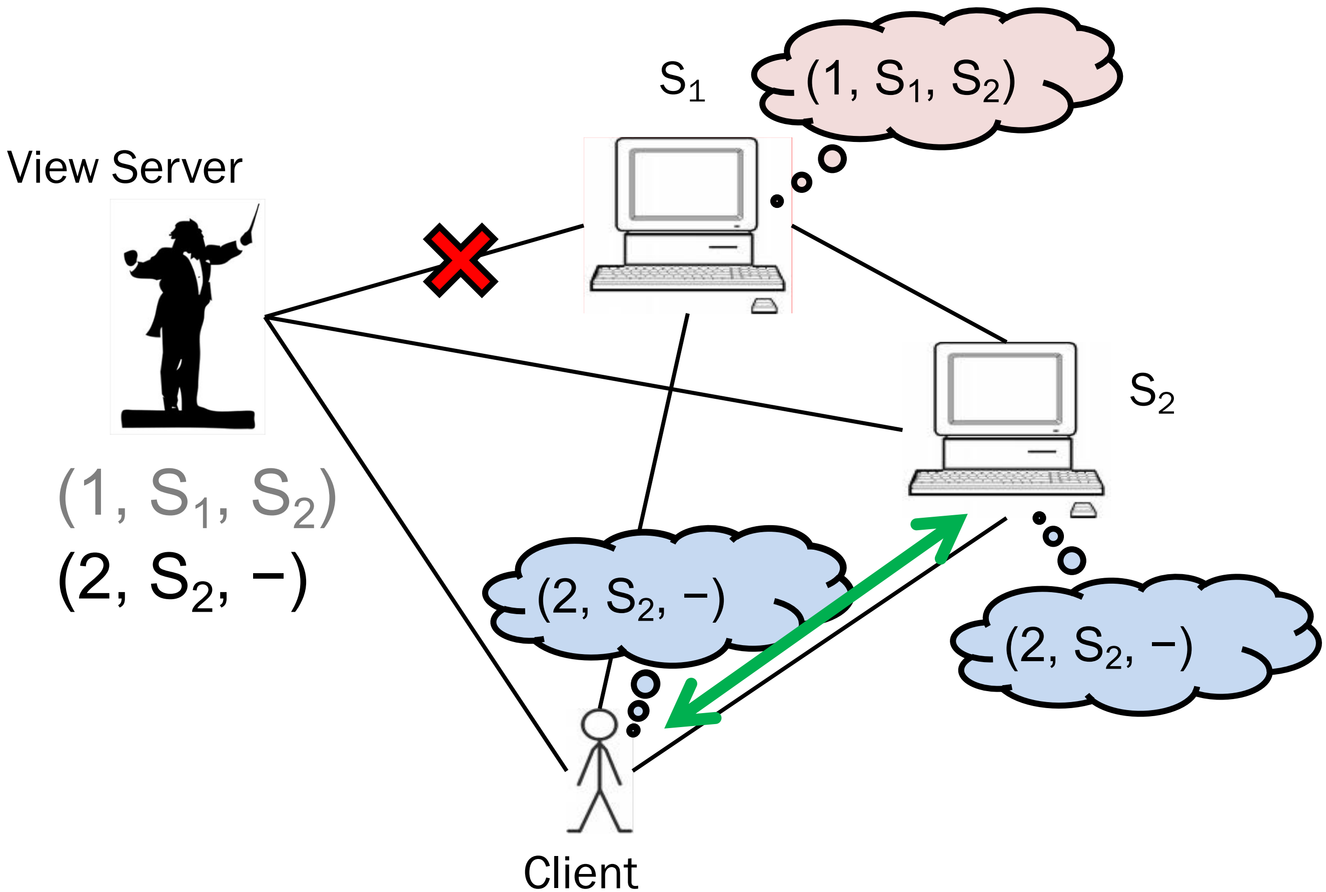
Server S_2 in the new view



Server S_2 in the new view



Server S_2 in the new view



State transfer via operation log

- *How does a new backup get the current state?*
  - If S_2 is backup in view i but was not in view $i-1$
  - S_2 asks primary to transfer the state

- One alternative: transfer the **entire operation log**

Simple, but inefficient (operation log is long)

State transfer via snapshot

- Every op must be either **before** or **after** state transfer
  - If op **before** transfer, transferred state must **reflect** op
  - If op **after** transfer, primary **forwards the op** to the backup after the state transfer finishes

- If each client has only one RPC outstanding at a time, state = map + result of the last RPC from each client
  - (Had to save this anyway for “at most once” RPC)

Summary of rules

1. View *i*'s **primary** must have been primary/backup in view $i-1$

2. A **non-backup** must reject forwarded requests
   - Backup accepts forwarded requests only if they are in its idea of the current view

3. A **non-primary** must reject direct client requests

4. Every operation must be **before or after** state transfer

Primary-Backup: Summary

- First step in our goal of making **stateful** replicas fault-tolerant

- Allows replicas to provide continuous service despite **persistent net and machine failures**

- Finds repeated application in practical systems (next)

Questions

- What happens if view service is down/partitioned?
  - View service could use P-B, but then…

- Suppose we want to use P-B to tolerate up to *f* simultaneous failures
  - How many replicas do we need?

P-B with VMs

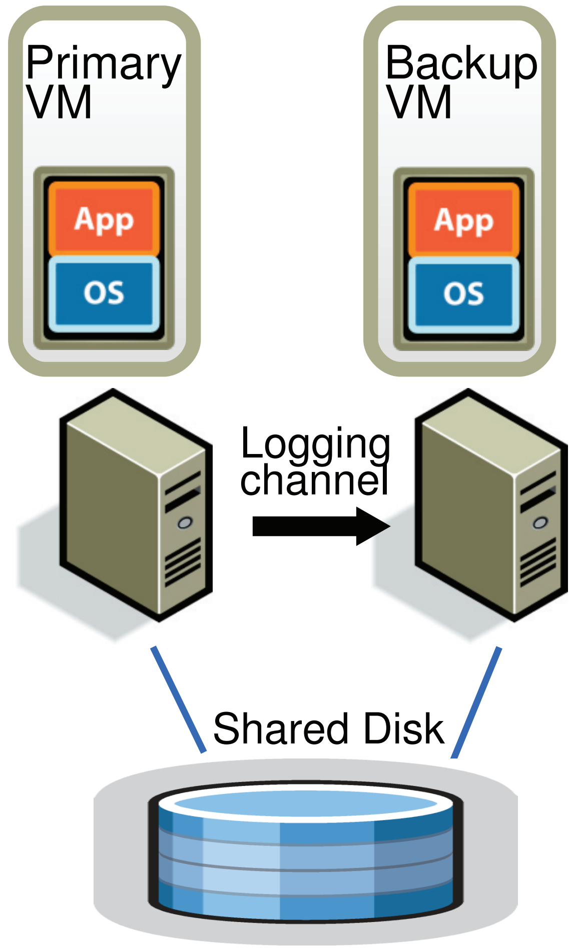
VMware vSphere Fault Tolerance (VM-FT)

- Goals:

1. Replication of the whole virtual machine
2. Completely transparent to applications and clients
3. High availability for any existing software

Overview

- Two virtual machines (*primary, backup*) on different bare metal
- *Logging channel* runs over network
- Fiber channel-attached *shared disk*



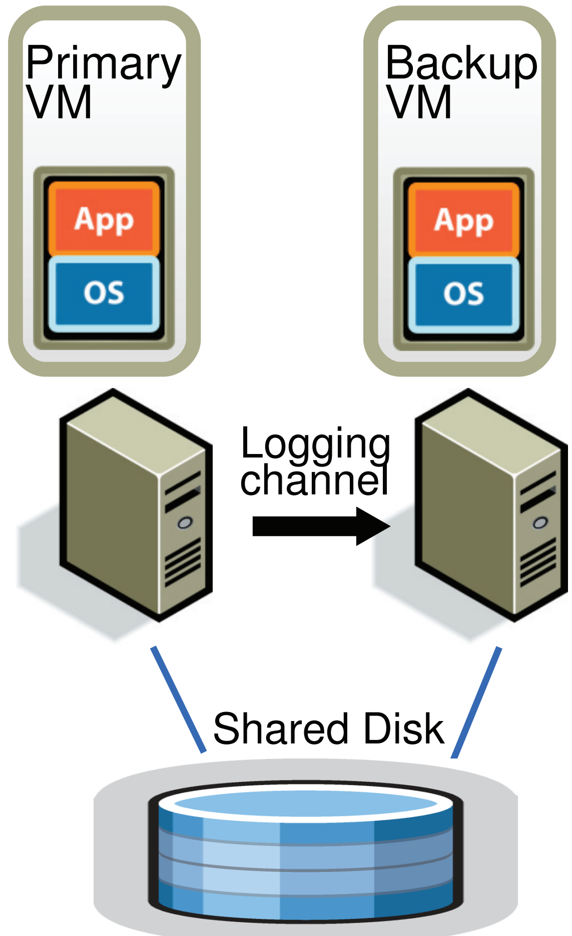
Virtual Machine I/O

- VM inputs
  - Incoming network packets
  - Disk reads
  - Keyboard and mouse events
  - Clock timer interrupt events

- VM outputs
  - Outgoing network packets
  - Disk writes

Overview

- **Primary** sends **inputs** to backup
- **Backup outputs** dropped
- Primary-backup **heartbeats**
  - If primary fails, backup takes over



VM-FT: Challenges

1. Making the backup an exact replica of primary

2. Making the system behave like a single server

3. Avoiding two primaries (Split Brain)

Log-based VM replication

- **Step 1:** Hypervisor at the primary logs the causes of non-determinism:

1. Log results of input events
   - Including current program counter value for each

2. Log results of non-deterministic instructions
   - *e.g.* log result of timestamp counter read (RDTSC)

Log-based VM replication

- **Step 2:** Primary hypervisor sends log entries to backup hypervisor over the logging channel

- Backup hypervisor replays the log entries
  - **Stops backup VM** at next input event or non-deterministic instruction
    - Delivers same input as primary

    - Delivers same non-deterministic instruction result as primary

VM-FT Challenges

1. Making the backup an exact replica of primary

2. Making the system behave like a single server
   - FT Protocol

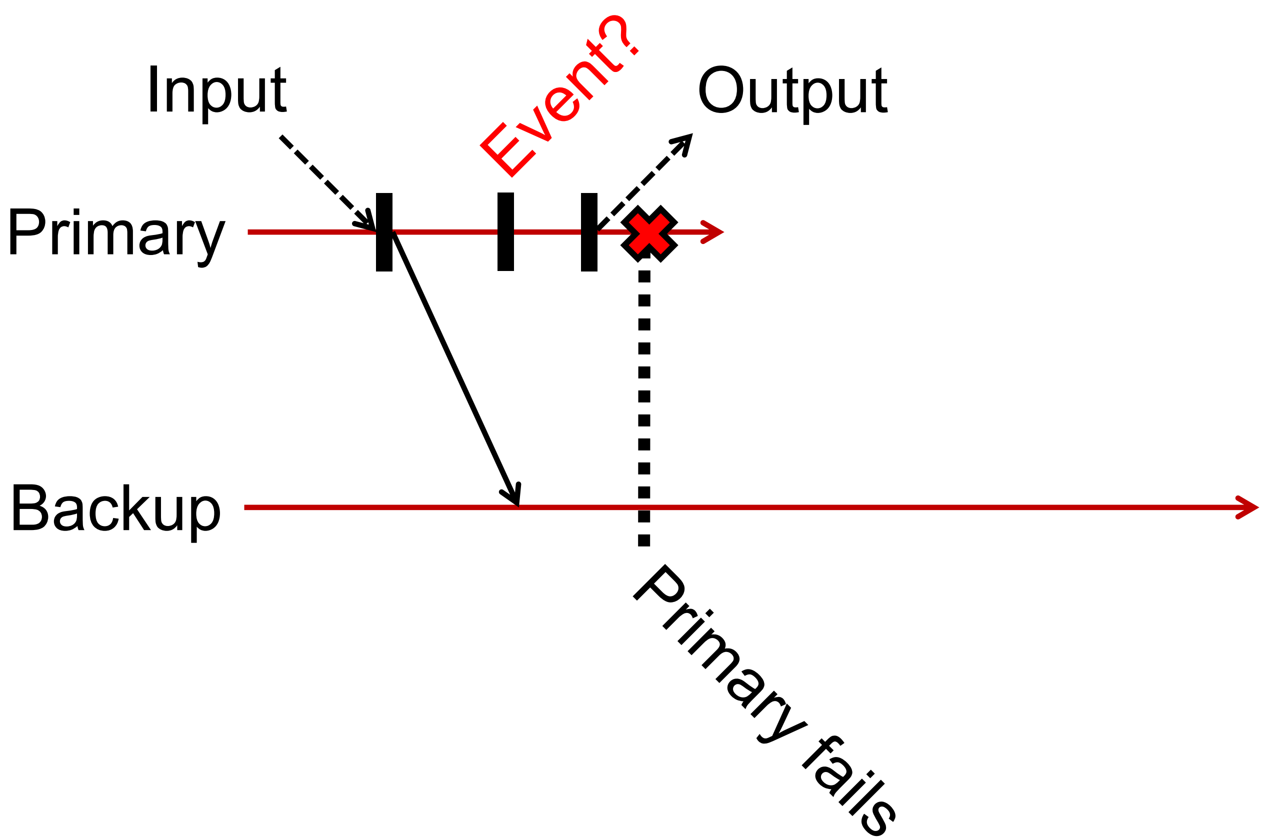
3. Avoiding two primaries (Split Brain)

Primary to backup failover

- When backup takes over, non-determinism will make it execute differently than primary would have done
  - This is okay!

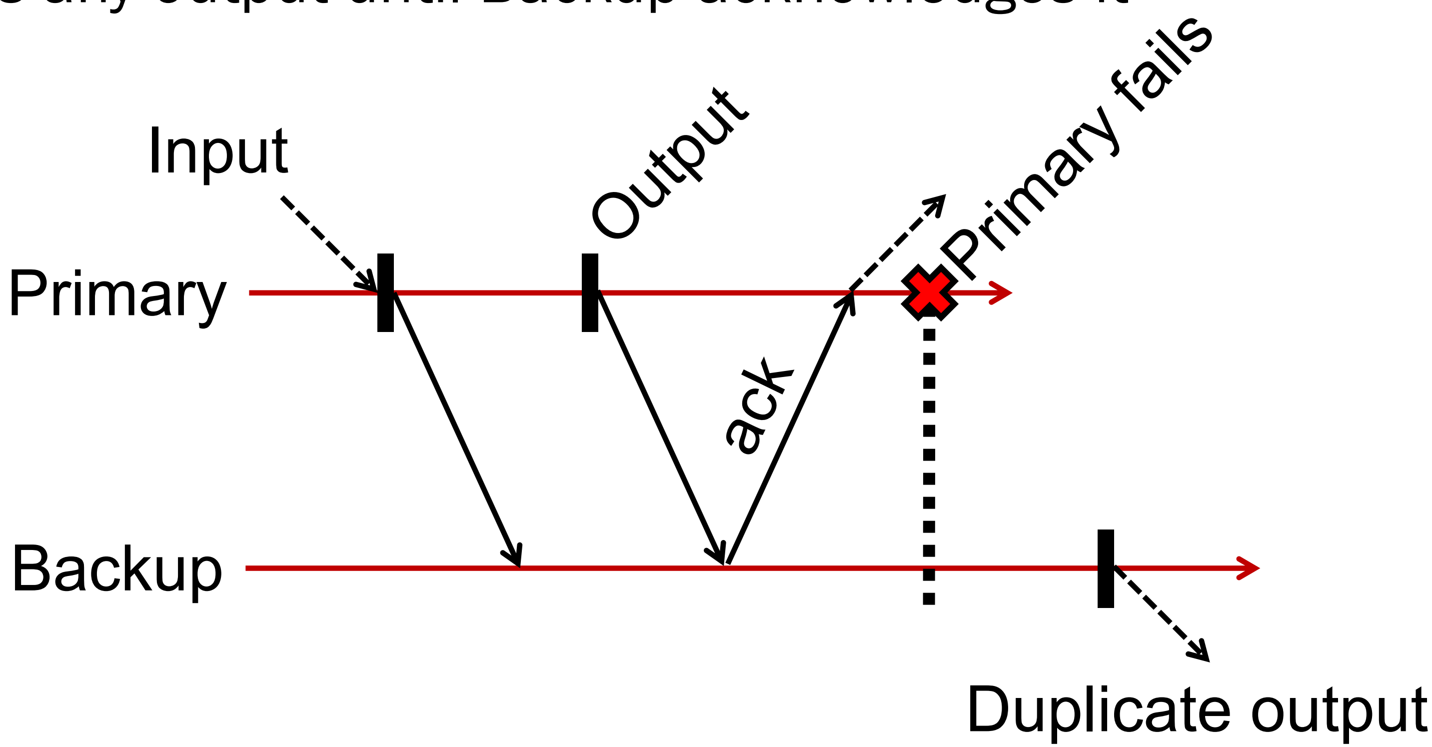
- **Output requirement:** When backup VM takes over, its execution is **consistent** with **outputs** the primary VM has already sent

The problem of inconsistency



FT protocol

- Primary **logs each output** operation
  - **Delays** any output until Backup acknowledges it



Can restart execution at an output event

VM-FT: Challenges

1. Making the backup an exact replica of primary

2. Making the system behave like a single server

3. Avoiding two primaries (Split Brain)
    - Logging channel may break

Detecting and responding to failures

- Primary and backup each run UDP heartbeats, monitor logging traffic from their peer
- Before "going live" (backup) or finding new backup (primary), execute an **atomic test-and-set** on a variable in shared storage
  - Note the "lease" assumption; on successful CAS assume old primary has "stopped"
  - Why is this guaranteed here?
- If the replica finds variable already set, it **aborts**

VM-FT: Conclusion

- Challenging application of primary-backup replication
- Design for correctness and consistency of replicated VM outputs despite failures
- Performance results show generally high performance, low logging bandwidth overhead