Time and Clocks

CS6450: Distributed Systems

Lecture 5

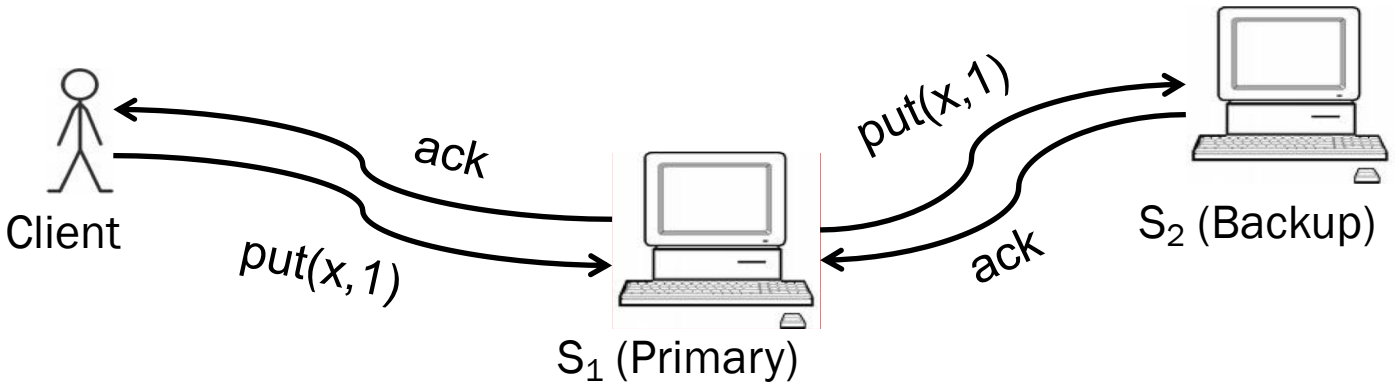
Ryan Stutsman

Takeaways

- 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
- Wall clocks drift, so they are all skewed
  - Synchronize to bound skew, but still left with uncertainty
  - Internal sync sometimes sufficient
  - Getting less sub-ms sync challenging due to network

Primary-Backup

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

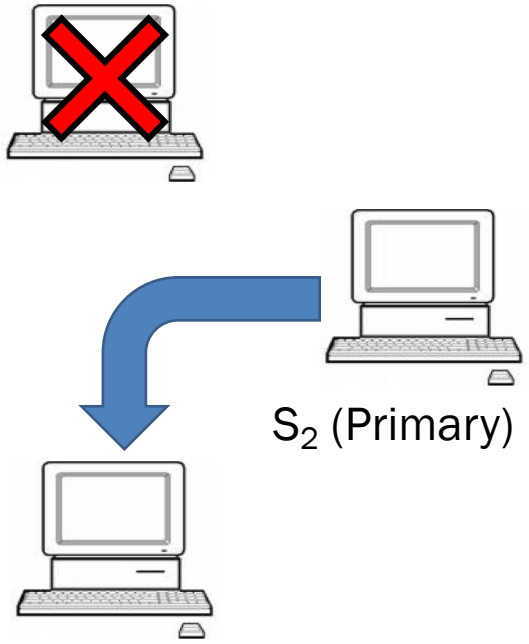
- For now, **assume** view server never fails

The view server decides the current view

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

Client

(1, S_1, S_2)
(2, S_2, –)
(3, S_2, S_3)



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

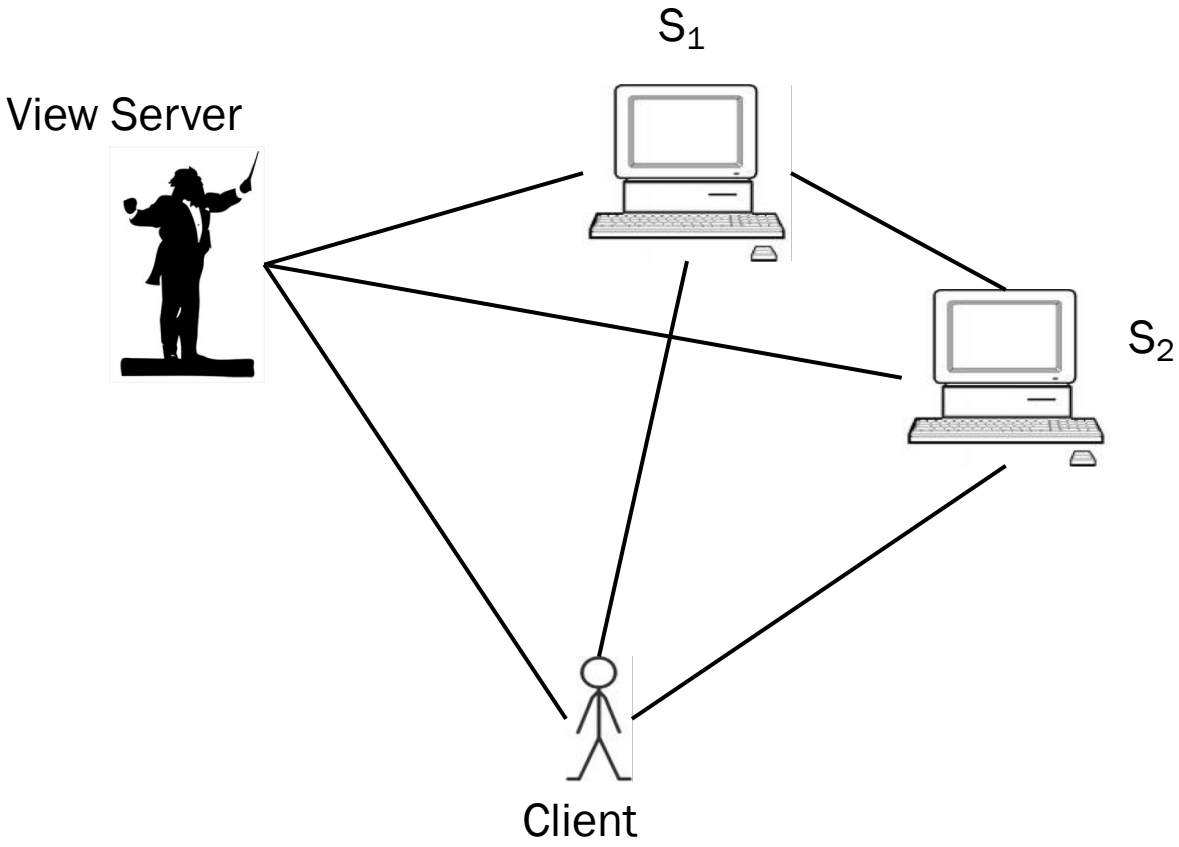
Transitioning between views

- *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

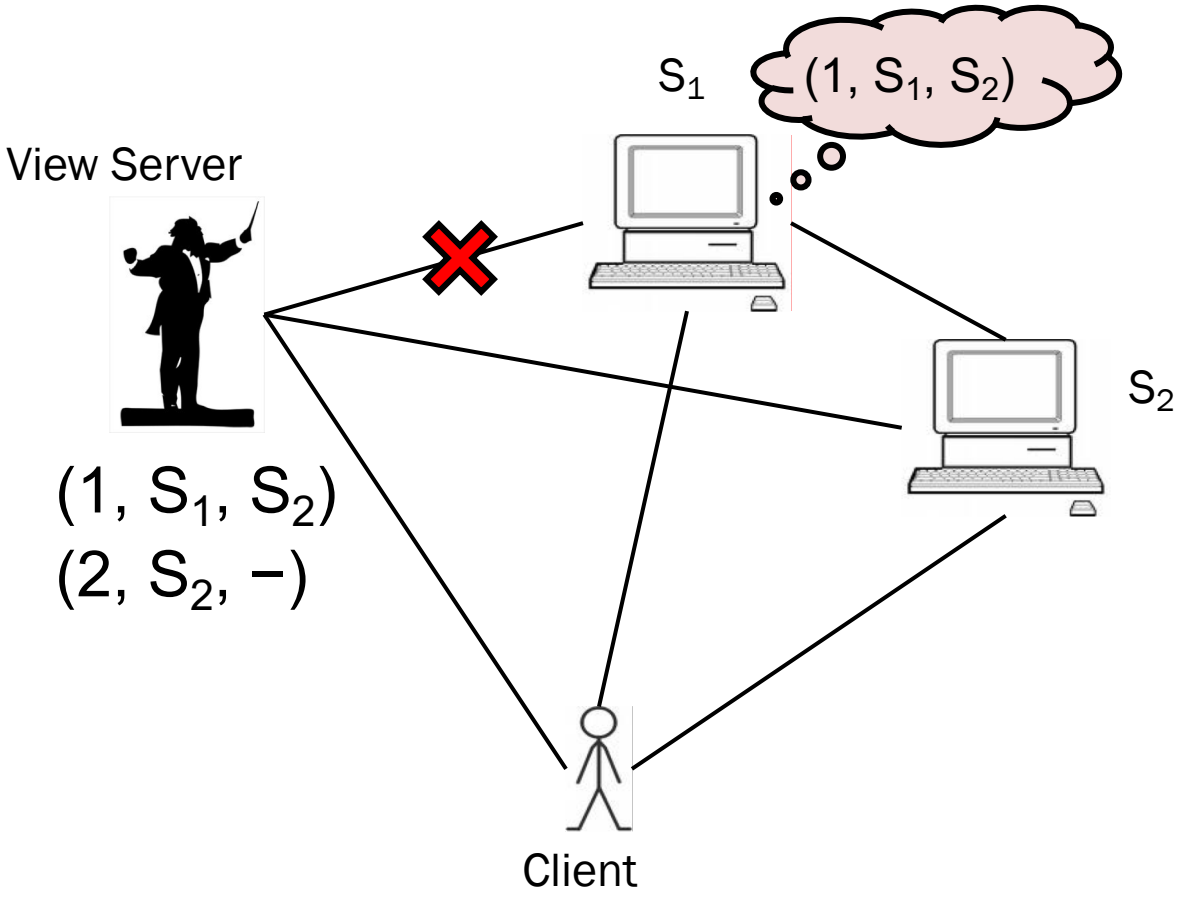
- *How does view server know whether backup is up to date?*
  - View server sends ***view-change*** message to all
  - New primary **must ack new view** once new backup is up-to-date

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

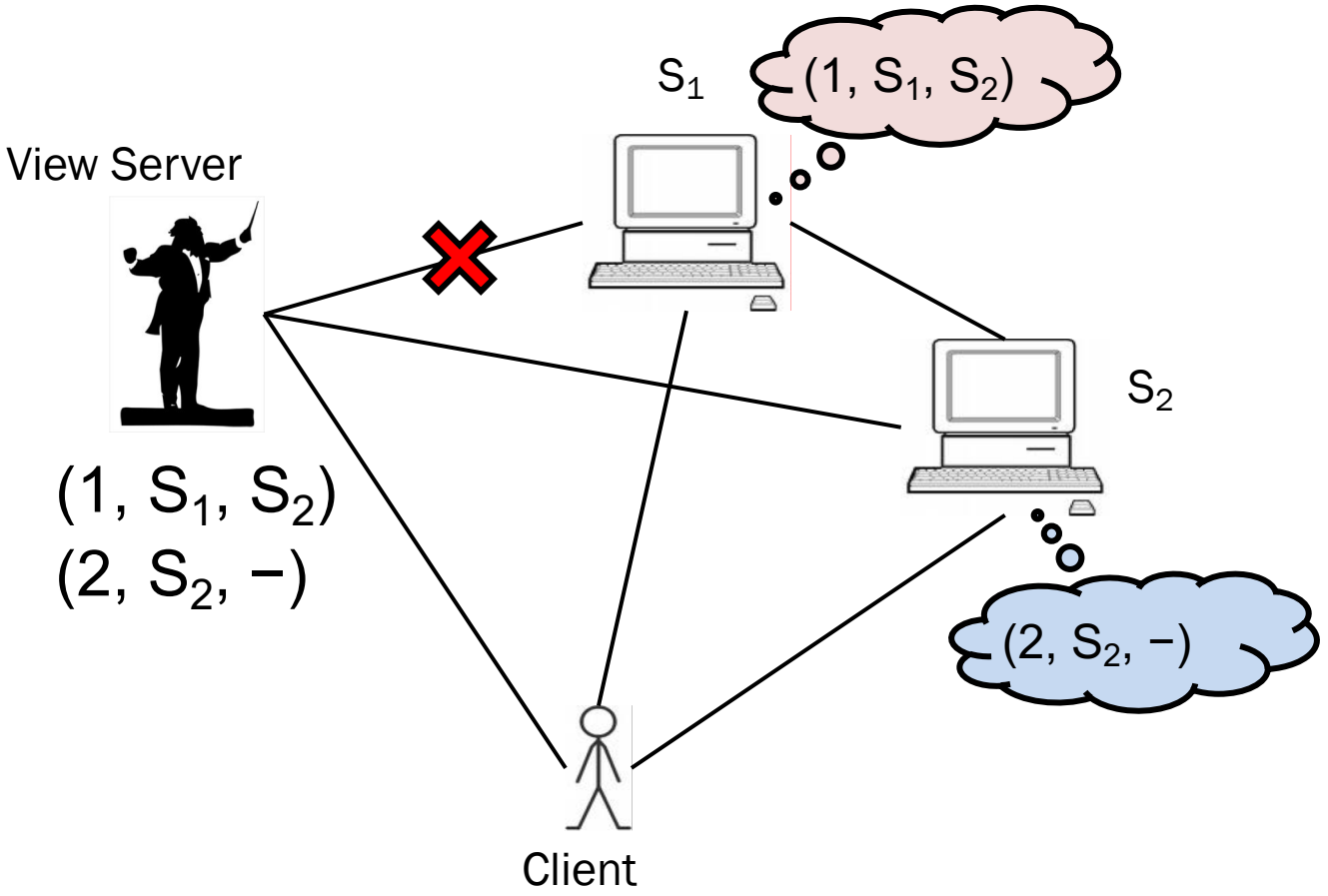
Split Brain?



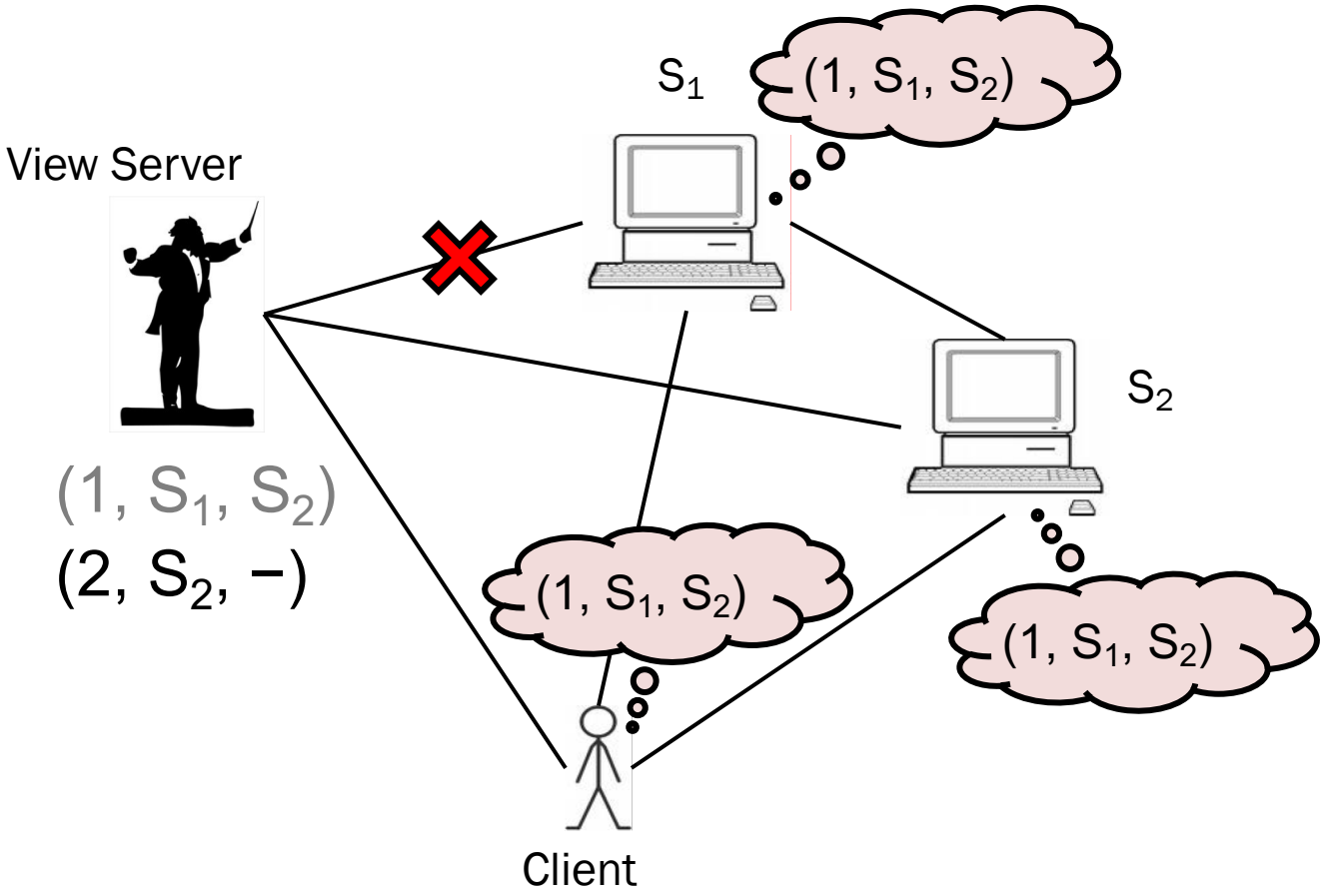
Split Brain?



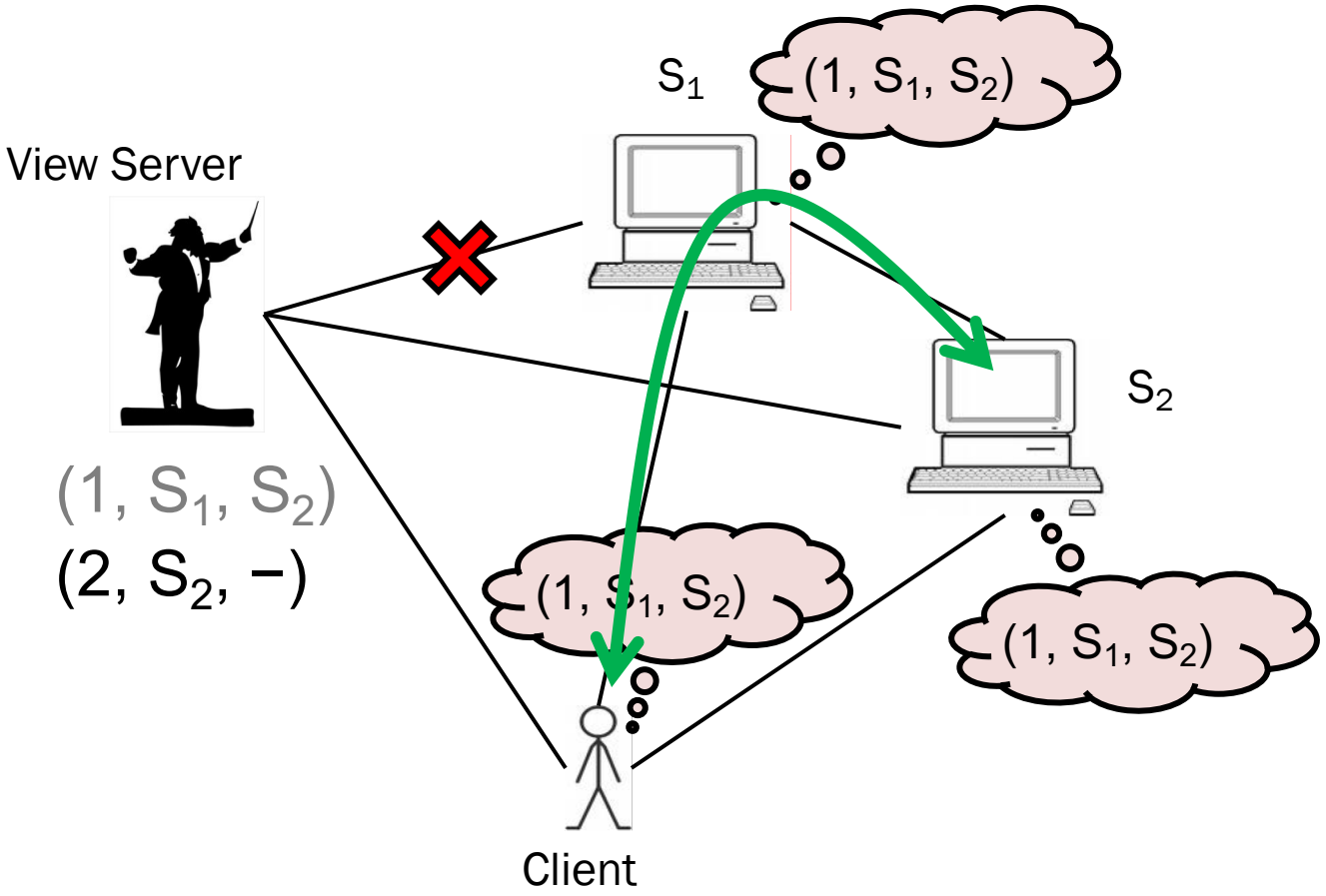
Split Brain?



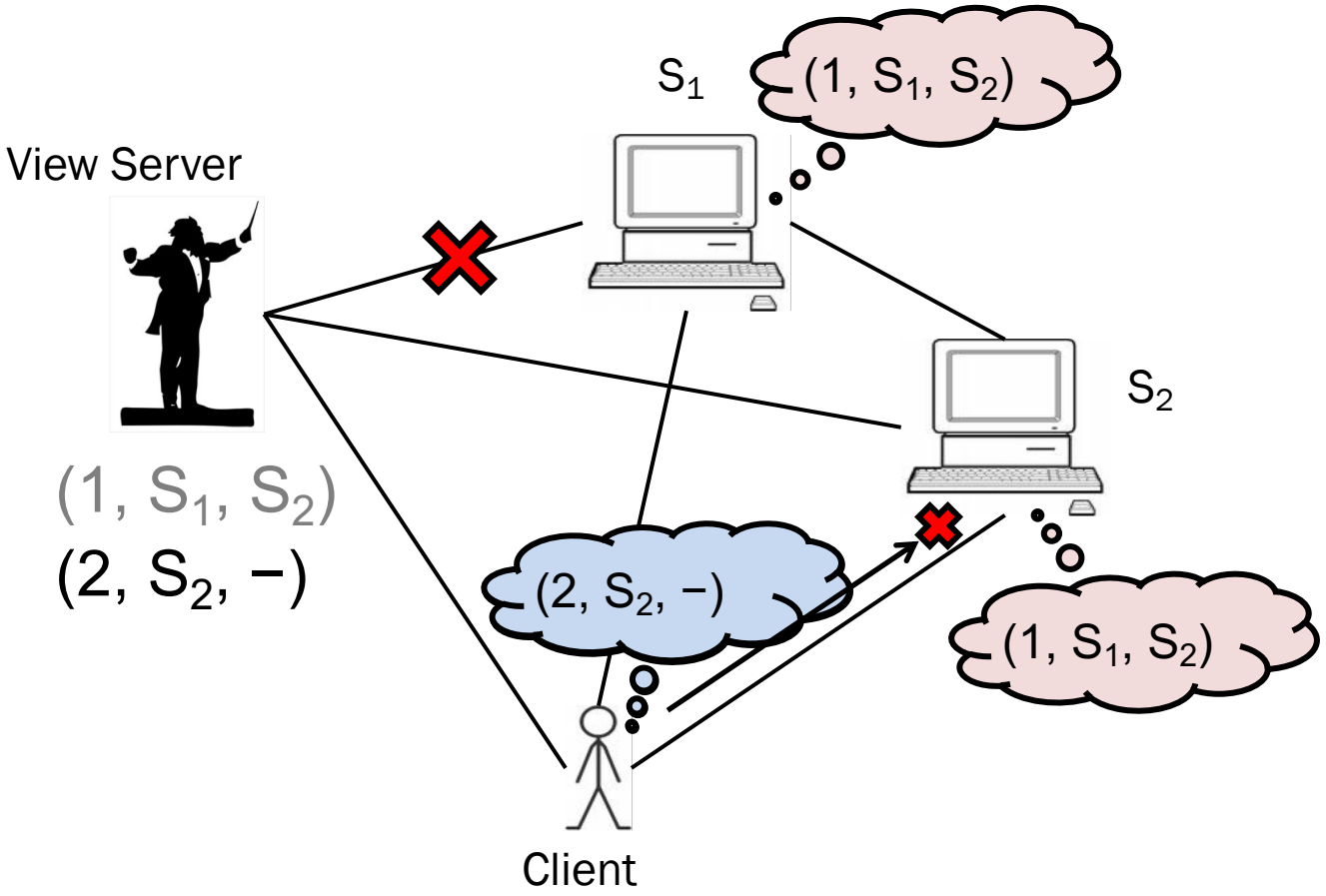
Server S_2 in the old view



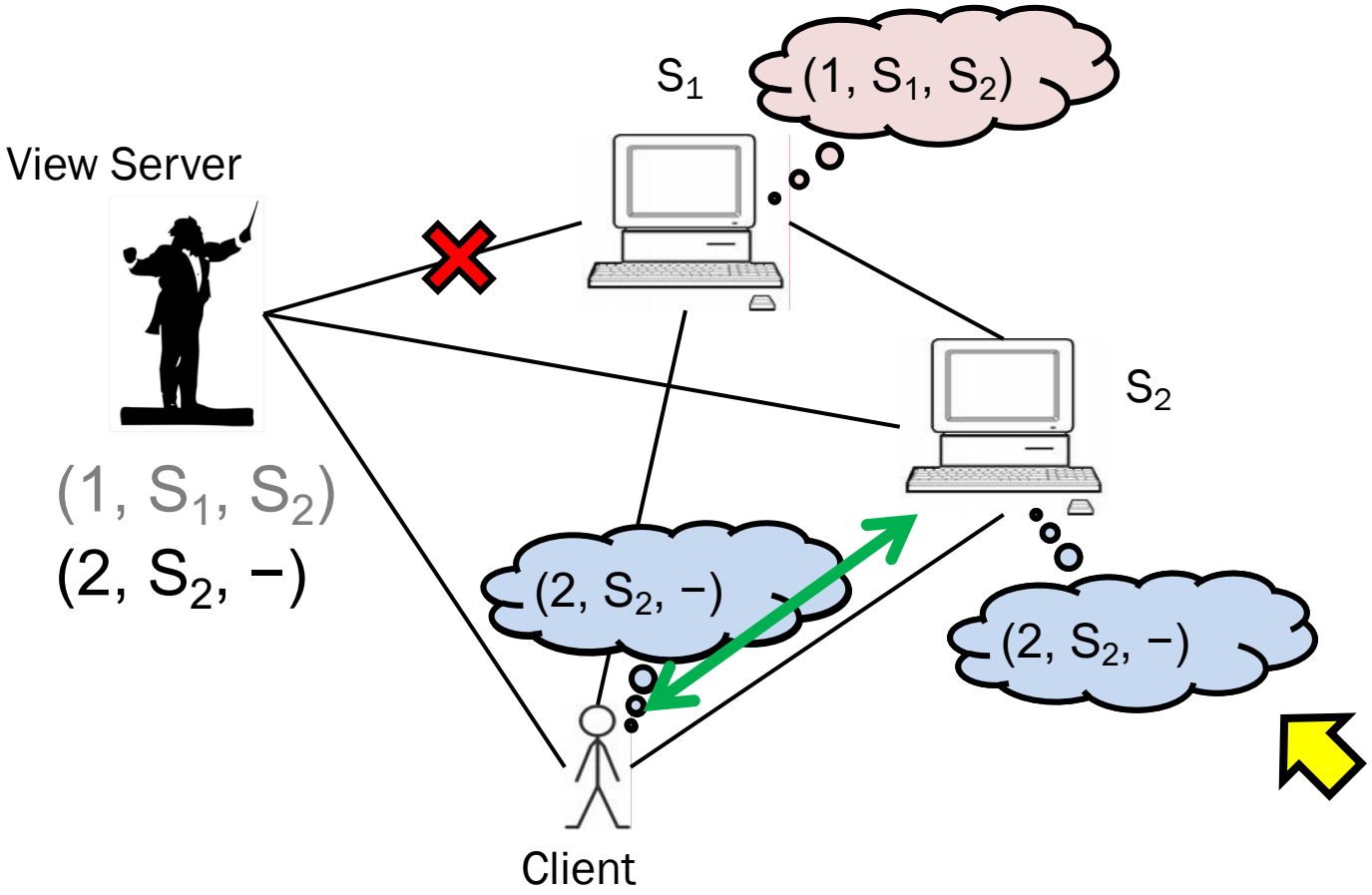
Server S_2 in the old view



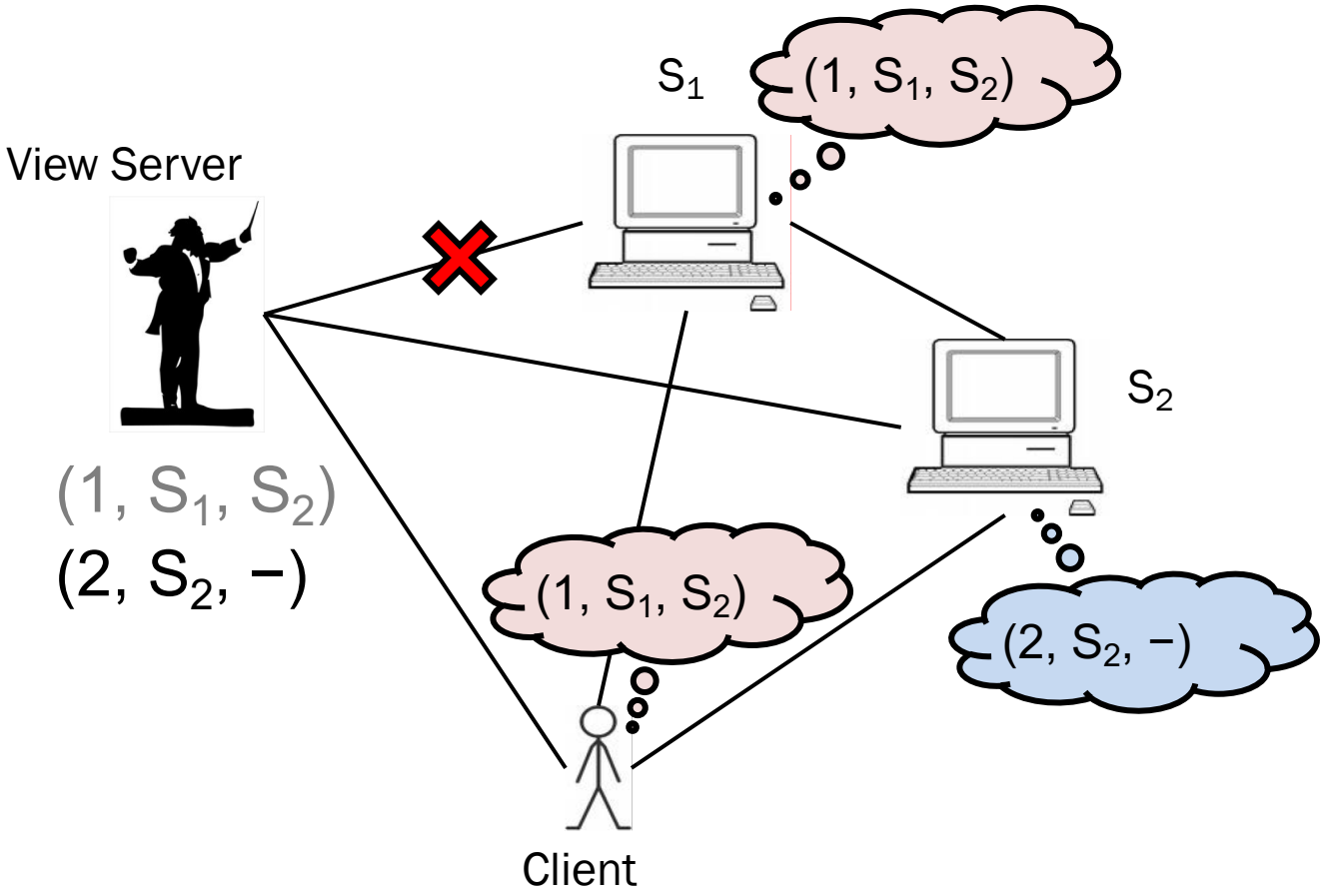
Server S_2 in the old view



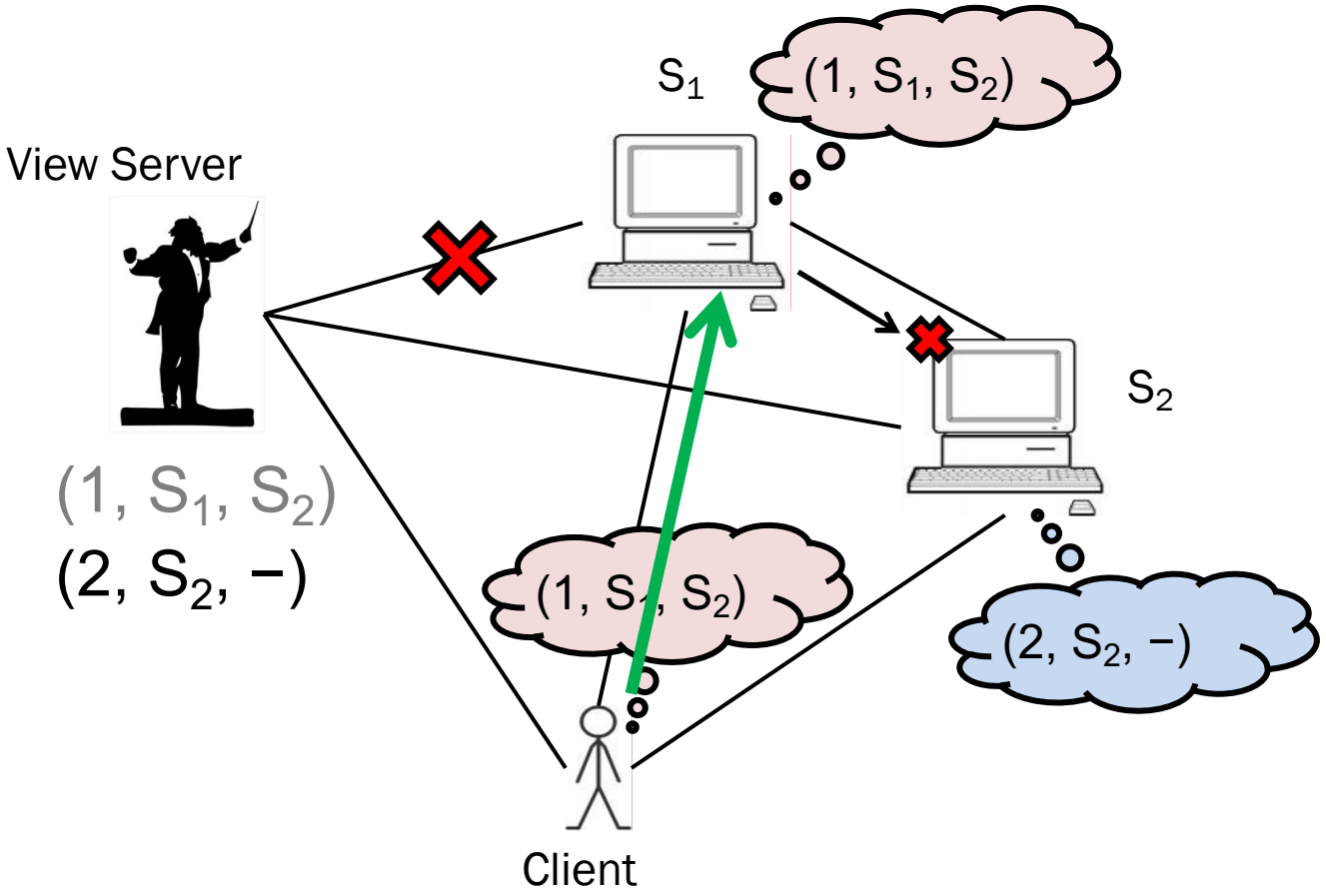
Server S_2 in the old view



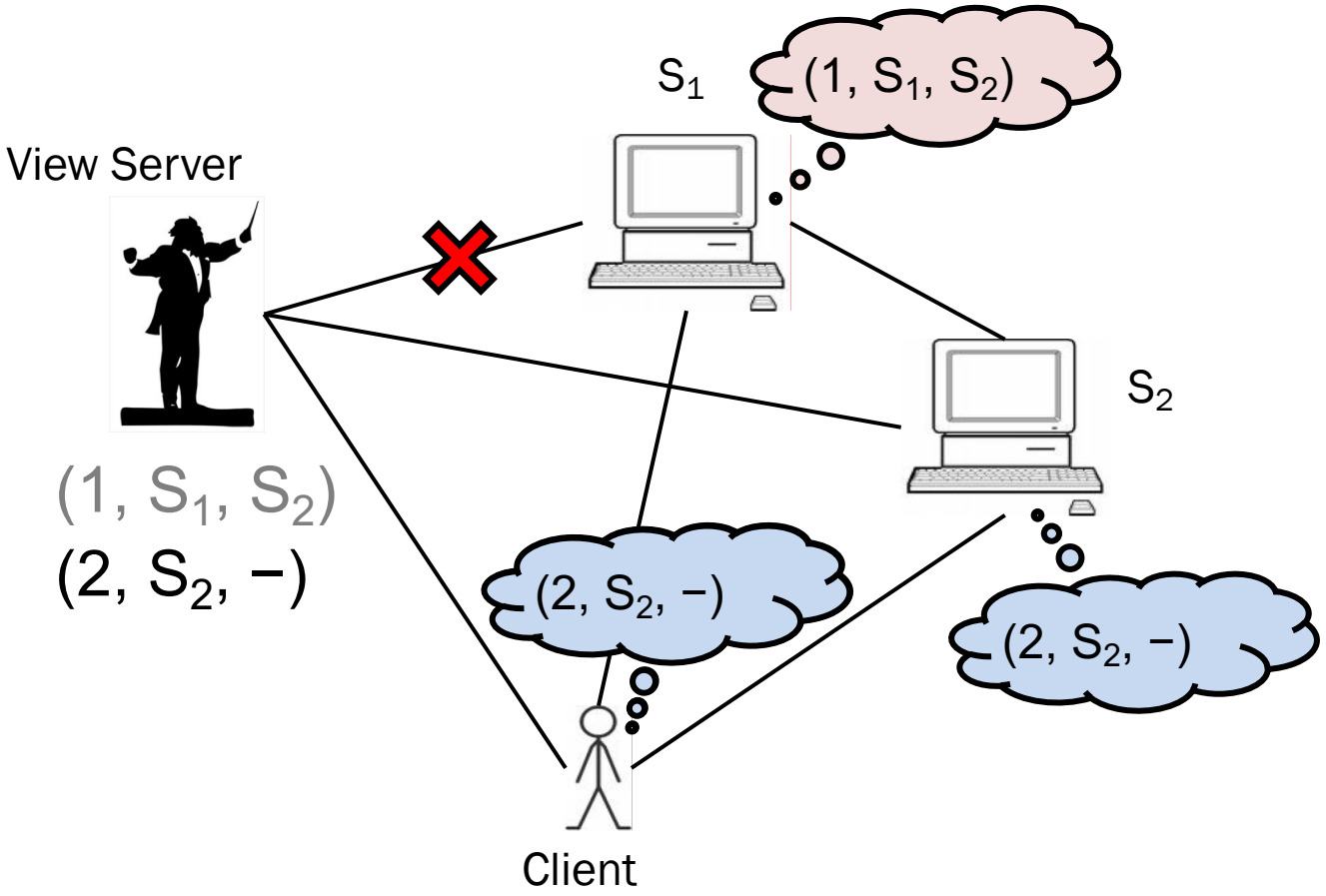
Server S_2 in the new view



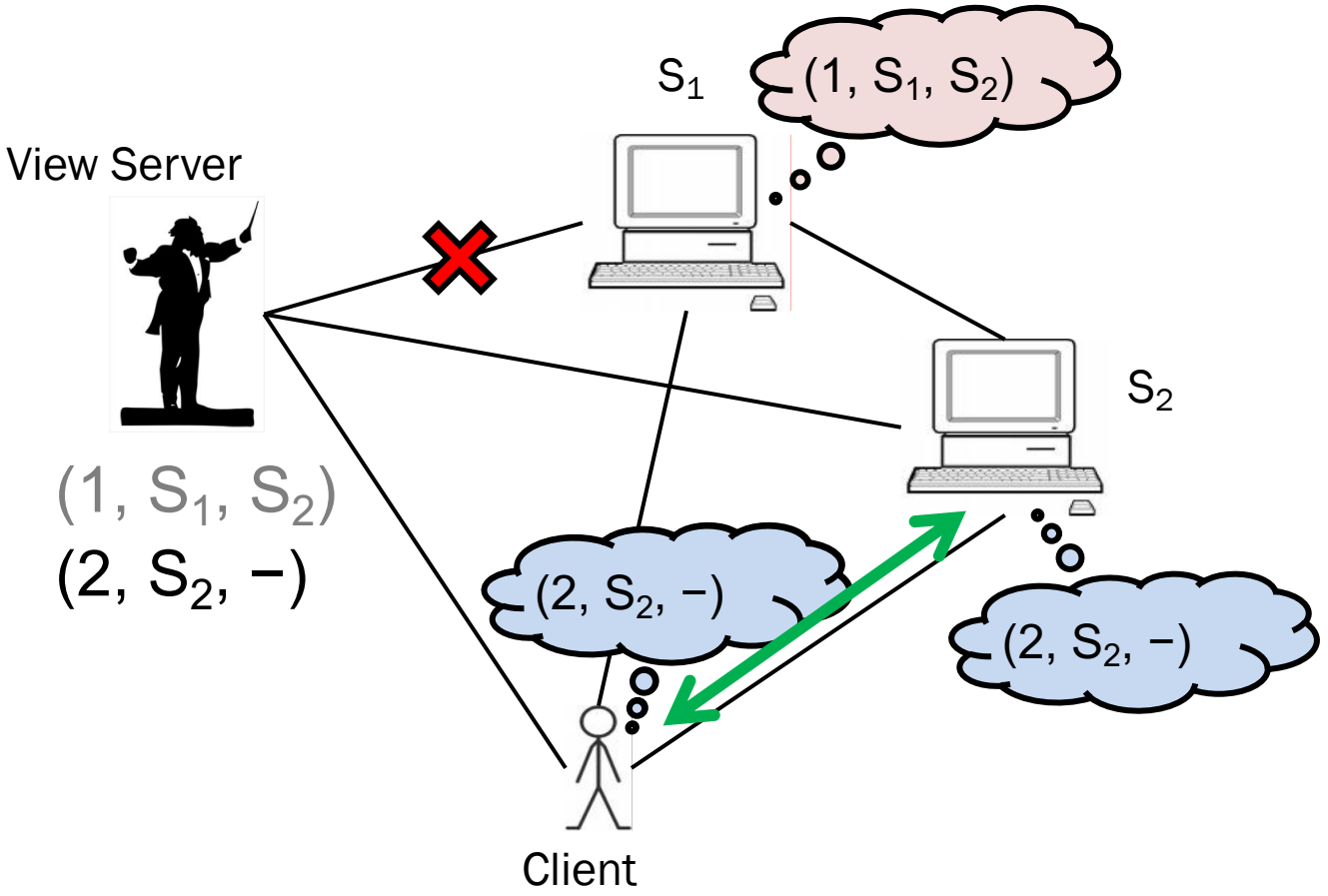
Server S_2 in the new view



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

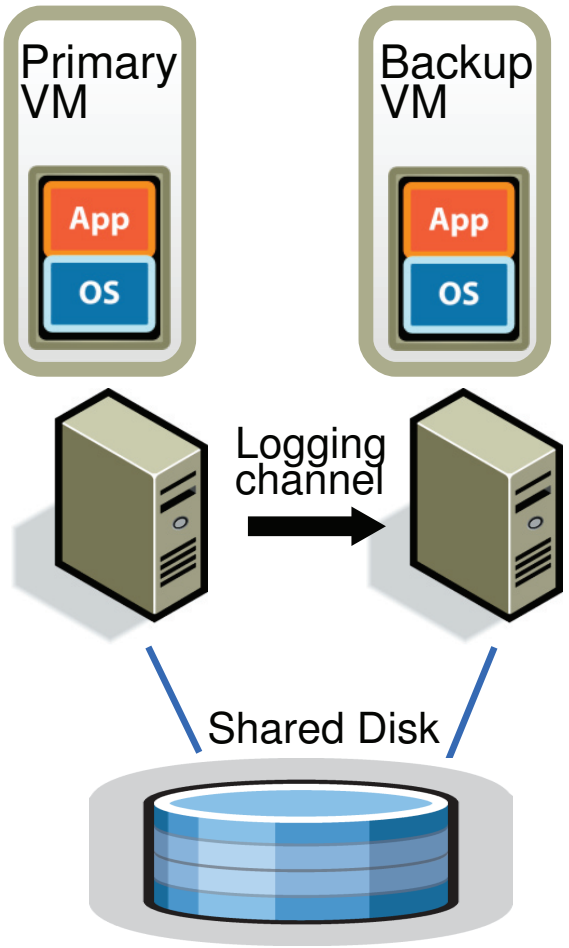
- First step in our goal of making **stateful** replicas fault-tolerant
- Allows replicas to provide continuous service despite persistent net and machine failures
- Finds repeated application in practical systems (next)

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?

VMware Fault Tolerance

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



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

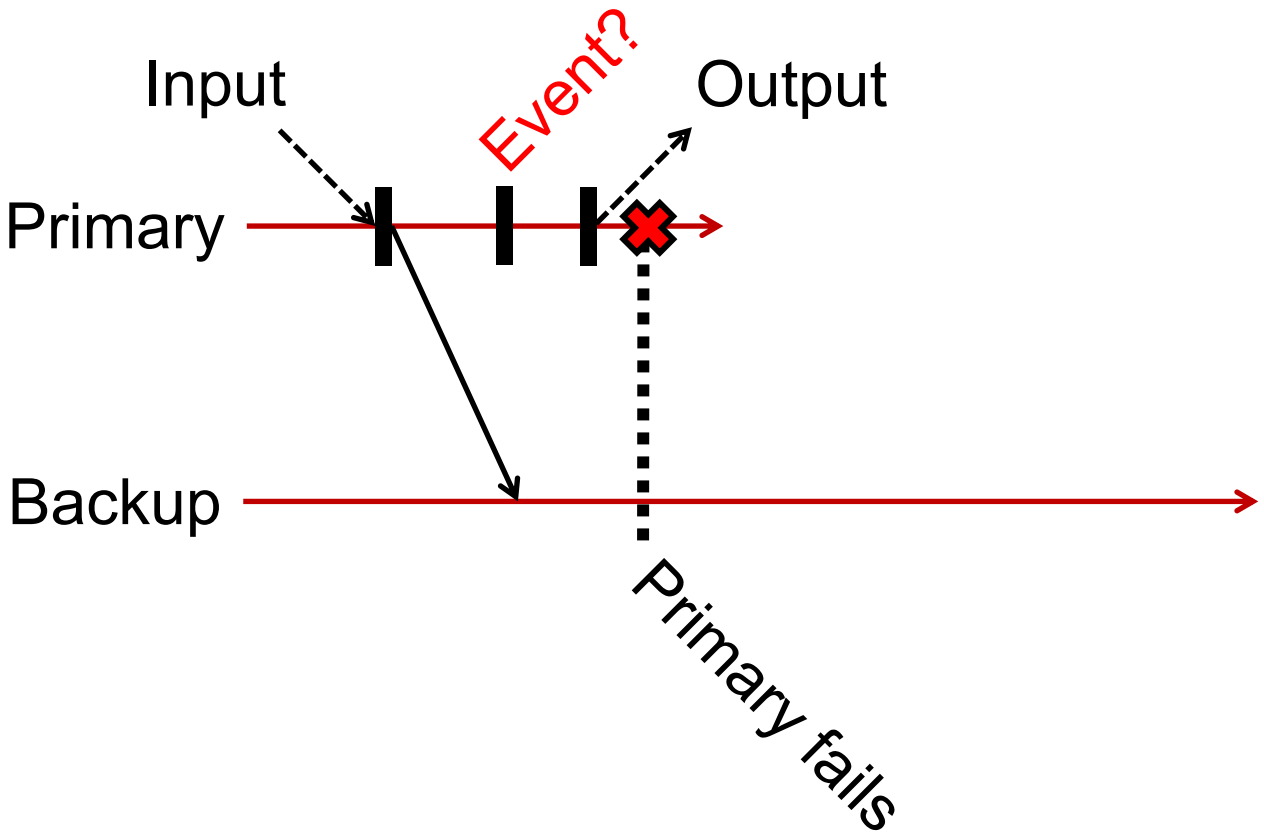
2. Log results of non-deterministic instructions
   - *e.g.* log result of timestamp counter read (RDTSC)

Log-based VM replication

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

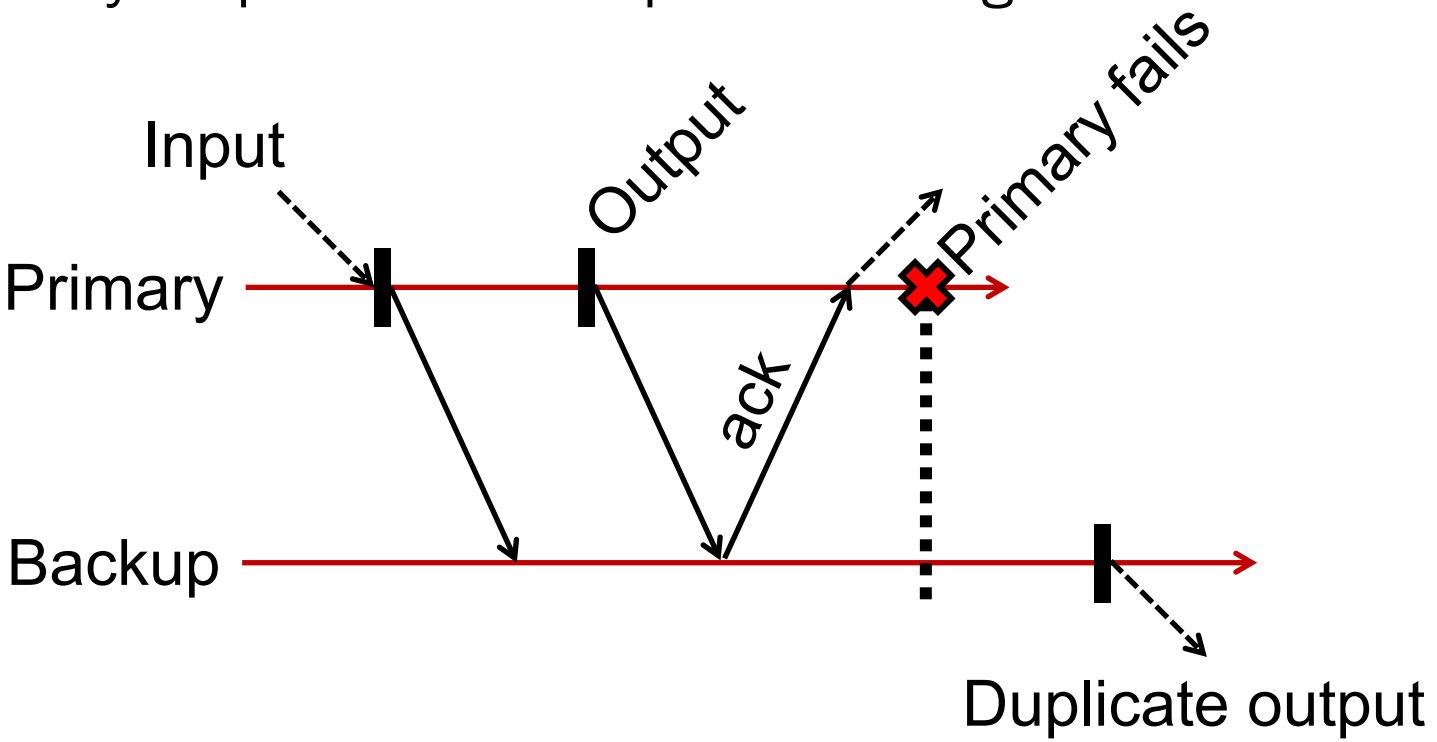
- Backup hypervisor **replays** the log entries
  - **Stops backup VM** at next input event or non-deterministic instruction
    - Delivers **same input** as primary
    - Delivers **same non-deterministic instruction result** as primary

The problem of inconsistency



FT protocol

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



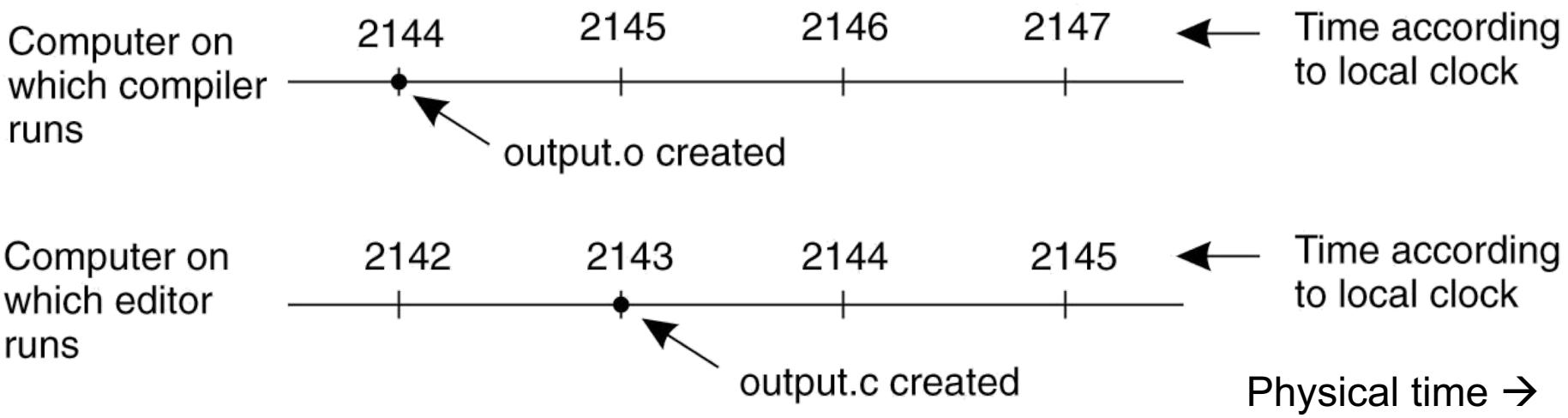
Can restart execution at an output event

Time & Clocks

Today

1. The need for time synchronization
2. “Wall clock time” synchronization
3. Logical Time

A distributed edit-compile workflow

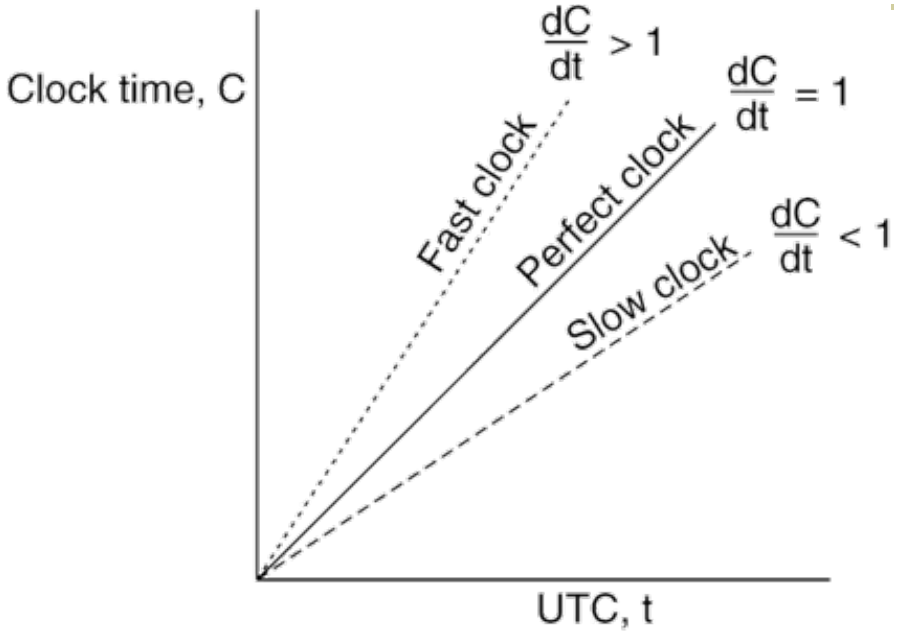


- 2143 < 2144 ➔ make **doesn't call compiler**

Lack of time synchronization result – a possible object file mismatch

Skew and Drift

- **Skew:** the difference between the times on two clocks (at any instant)
- **Drift:** the difference *per unit of time* from some ideal reference clock



What makes time synchronization hard?

1. Quartz oscillator **sensitive** to temperature, age, vibration, radiation
   - Accuracy *about* one part per million (**one second** of **clock drift** over **12 days**)

2. The internet is:
   - ***Asynchronous:*** arbitrary message **delays**
   - ***Best-effort***: messages **don't always arrive**

Today

1. The need for time synchronization

2. **"Wall clock time" synchronization**
   - Cristian's algorithm, Berkeley algorithm, NTP

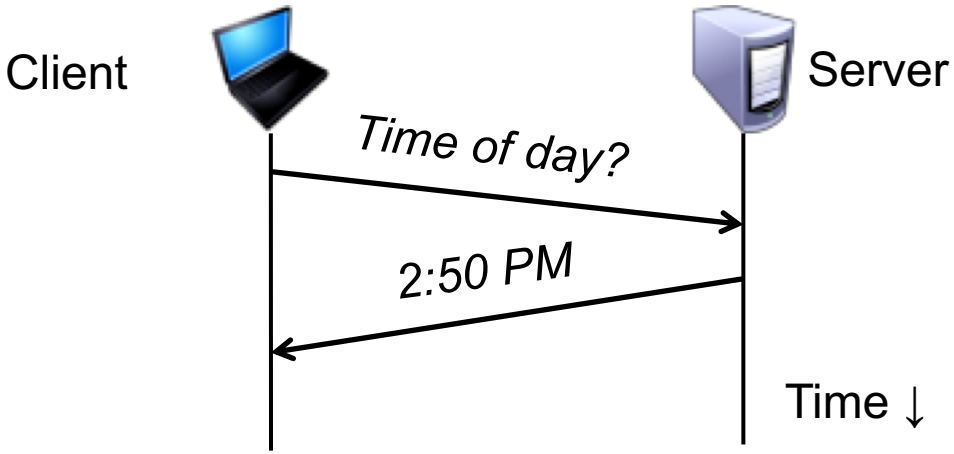
3. Logical Time
   - Lamport clocks
   - Vector clocks

Just use Coordinated Universal Time?

- UTC is broadcast from radio stations on land and satellite (*e.g.*, the Global Positioning System)
  - Computers with receivers can synchronize their clocks with these timing signals
- Signals from land-based stations are accurate to about 0.1–10 milliseconds
- Signals from GPS are accurate to about one microsecond
  - *Why can't we put GPS receivers on all our computers?*

Synchronization to a time server

- Suppose a server with an accurate clock (*e.g.*, GPS-disciplined crystal oscillator)
  - Could simply issue an RPC to obtain the time:



- But this doesn't account for network latency
  - **Message delays** will have **outdated** server's answer

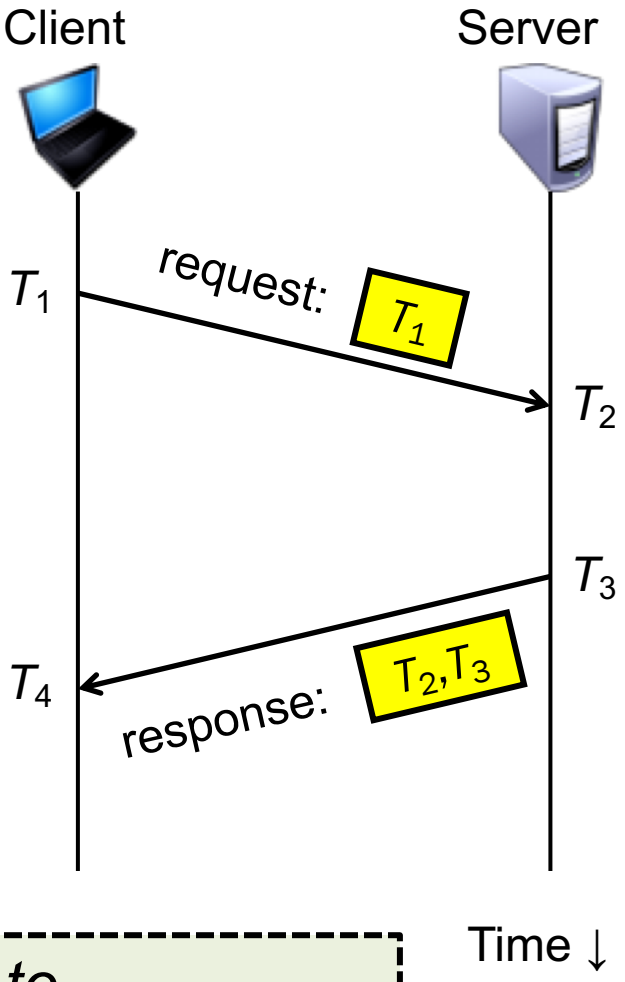
Cristian's algorithm: Outline

1. Client sends ***request***, timestamped with local clock T_1

2. Server timestamps its receipt of the request T_2 with its local clock

3. Server sends a ***response*** packet with its local clock T_3 and T_2

4. Client locally timestamps its receipt of server's response T_4



How can the client use these timestamps to synchronize its local clock to the server's local clock?

Cristian's algorithm: Offset sample calculation

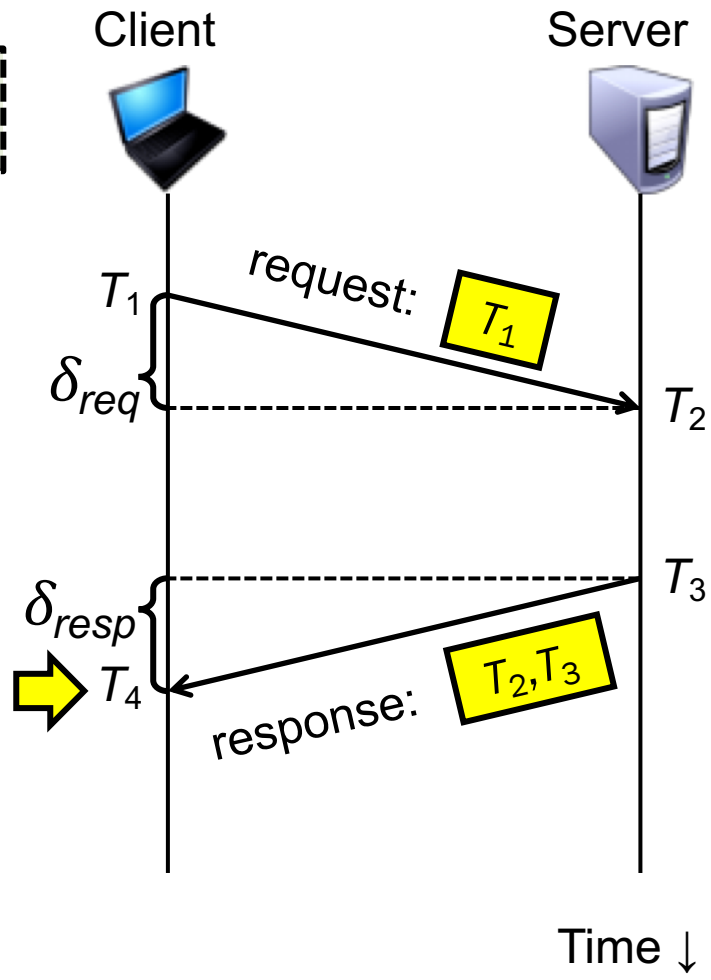
Goal: Client sets clock ← $T_3 + \delta_{resp}$

- Client samples *round trip time* δ = $\delta_{req} + \delta_{resp} = (T_4 - T_1) - (T_3 - T_2)$

- But client knows δ, not δ_{resp}

Assume: $\delta_{req} \approx \delta_{resp}$

Client sets clock ← $T_3 + \frac{1}{2}\delta$



Today

1. The need for time synchronization

2. **"Wall clock time" synchronization**
   - Cristian's algorithm, **Berkeley algorithm,** NTP

3. Logical Time
   - Lamport clocks
   - Vector clocks

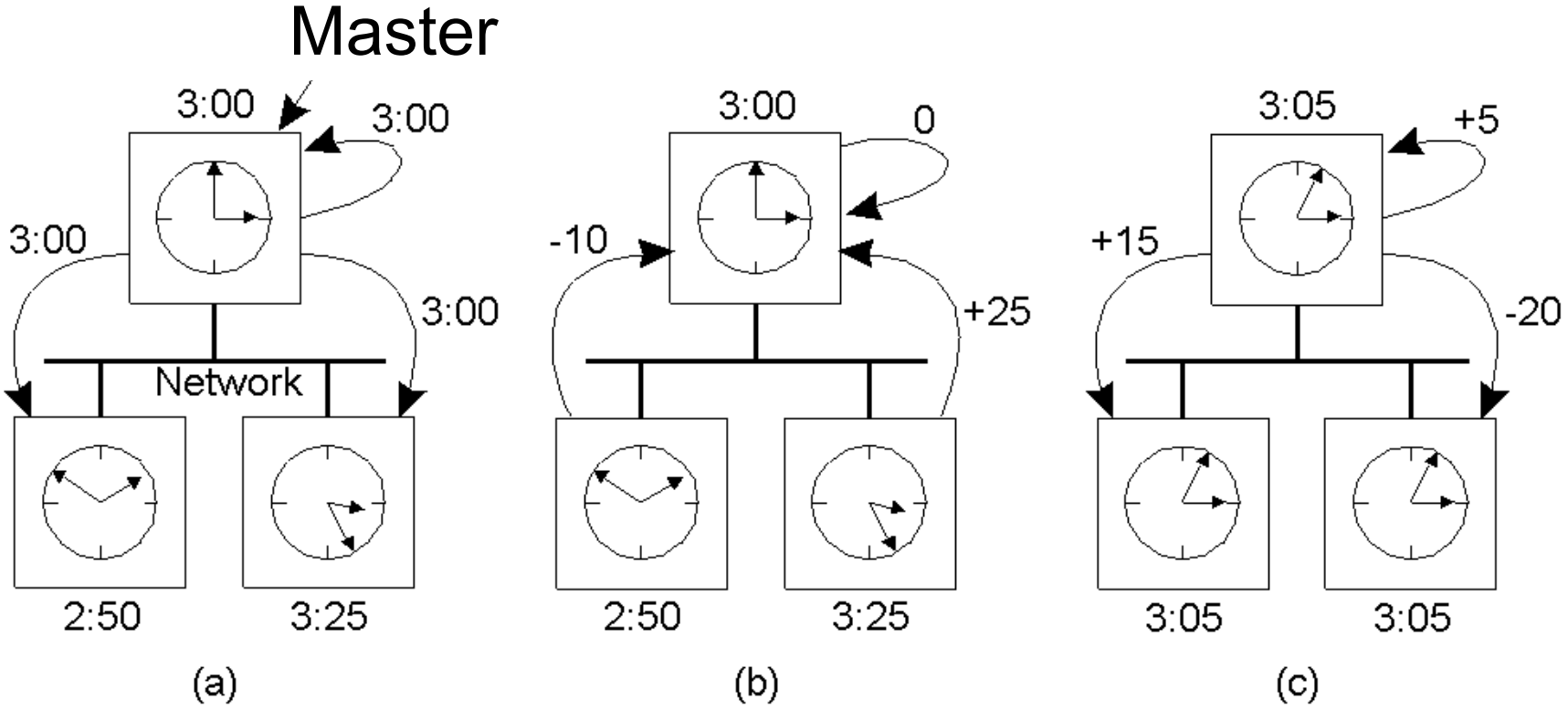
Berkeley algorithm

- A **single time server** can **fail**, blocking timekeeping

- The *Berkeley algorithm* is a distributed algorithm for timekeeping
  - Assumes all machines have **equally-accurate local clocks**
  - Obtains **average** from participating computers and synchronizes clocks to that average

Berkeley algorithm

- *Master machine*: polls *L* other machines using Cristian's algorithm → $\{\theta_i\}$ ($i = 1 \ldots L$)



- Note: Berkeley algorithm is *internal* synchronization
  - May not sync with “real-world” time
  - Does it matter or not?

Today

1. The need for time synchronization

2. **"Wall clock time" synchronization**
   - Cristian's algorithm, Berkeley algorithm, **NTP**

3. Logical Time
   - Lamport clocks
   - Vector clocks

The Network Time Protocol (NTP)

- Enables clients to be accurately synchronized to UTC despite message delays

- Provides reliable service
  - Survives lengthy losses of connectivity
  - Communicates over redundant network paths

- Provides an accurate service
  - Unlike the Berkeley algorithm, leverages heterogeneous accuracy in clocks

NTP: System structure

- Servers and time sources are arranged in layers (*strata*)
  - Stratum 0: High-precision time sources themselves
    - *e.g.*, atomic clocks, shortwave radio time receivers
  - Stratum 1: NTP servers **directly connected** to Stratum 0
  - Stratum 2: NTP servers that synchronize with Stratum 1
    - Stratum 2 servers are **clients of** Stratum 1 servers
  - Stratum 3: NTP servers that synchronize with Stratum 2
    - Stratum 3 servers are **clients of** Stratum 2 servers
- Users' computers synchronize with Stratum 3 servers

NTP operation: Server selection

- Messages between an NTP client and server are exchanged in pairs: request and response
  - Use Cristian's algorithm

- For i^{th} message exchange with a particular server, calculate:
  1. Clock offset θ_i from client to server
  2. Round trip time δ_i between client and server

- Over last eight exchanges with server k, the client computes its dispersion $\sigma_k = \max_i \delta_i - \min_i \delta_i$
  - Client uses the server with **minimum dispersion**

- Cristian's algorithm used only one sided delay
  - potential inaccuracy is half the additional RTT delay

NTP operation: How to change time

- Can't just change time: Don't want time to run backwards
  - Recall the make example

- Instead, change the **update rate** for the clock
  - Changes time in a more gradual fashion
  - Prevents inconsistent local timestamps

Clock synchronization: Take-away points

- Clocks on different systems will always behave differently
  - Disagreement between machines can result in undesirable behavior
- NTP, Berkeley clock synchronization
  - Rely on timestamps to estimate network delays
  - **100s μs–ms accuracy**
  - Clocks never exactly synchronized
- Often inadequate for distributed systems
  - Often need to reason about the **order of events**
  - Might need precision on the order of **ns**

Next time

1. The need for time synchronization

2. "Wall clock time" synchronization
   - Cristian's algorithm, Berkeley algorithm, NTP

3. Logical Time
   - Lamport clocks
   - Vector clocks