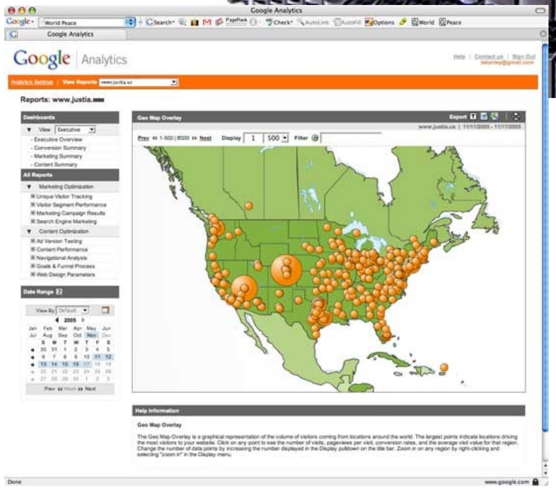
# Reliability: Concepts + Replication

ACS, Yongluan Zhou

(many slides are made by Marcos Vaz Salles)

# Highly-Available Systems

- Content distribution, web, media
  - E.g., YouTube
- Data Stores
  - E.g., Amazon Dynamo, Google F1
- Analytics
  - Long running jobs in Spark / MapReduce / Hadoop
  - Continuous stream processing, e.g., Storm / Samza / Kafka

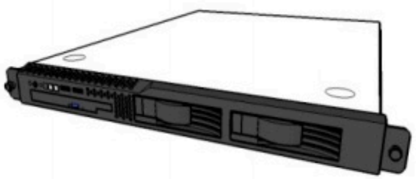



# Throughput Most Important Metric

# Scale is the name of the game

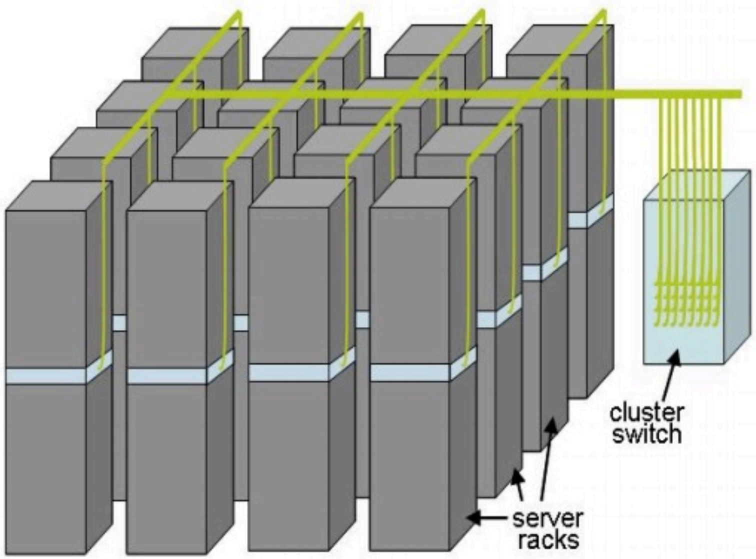
- Volumes of information
  - TBs – PBs of raw data
- Variety of schemas and formats
  - Tens – hundreds of schemas
- Large data centers
  - Tens of thousands of machines

# The Machinery



Servers
- CPUs
- DRAM
- Disks

Racks
- 40-80 servers
- Ethernet switch




Clusters

Source: Dean

# So, everything works?

- Assume a computer has probability of failure p
- If system needs N computers to work, what is probability of system working?

- Probability of one component working: $1-p$
- Probability of all components working: $(1-p)^N$
    - Assuming failures are independent!
    - Correlated failures are the reality – and make it even worse

# Reliability Measures

- Mean Time to Failure (MTTF)
- Mean Time to Repair (MTTR)
- Mean Time Between Failures (MTBF)
- MTBF = MTTF + MTTR
- Availability = MTTF / MTBF
- Downtime = (1 – Availability) = MTTR / MTBF

- Consider N = 10,000 and for one computer,
  MTTF = 30 years

1. How to estimate the value of MTTF for a system that has a long MTTF?
2. How often do you estimate to see a computer failing in the above scenario?

# Reliability & Availability

- Things will crash. Deal with it!
  - Assume you could start with super reliable servers (MTBF of 30 years)
  - Build computing system with 10 thousand of those
  - **Watch one failure per day**
  - Facebook* has to mitigate one datacenter outage every two weeks!

- Fault-tolerant software is inevitable

- Typical yearly flakiness metrics
  - 1-5% of your disk drives will die
  - Servers will crash at least twice (2-4% failure rate)

*Kaushik Veeraraghavan, Justin Meza, et al. Maelstrom: Mitigating Datacenter-level Disasters by Draining Interdependent Traffic Safely and Efficiently. In OSDI'2018.

Source: Dean

# The Joys of Real Hardware

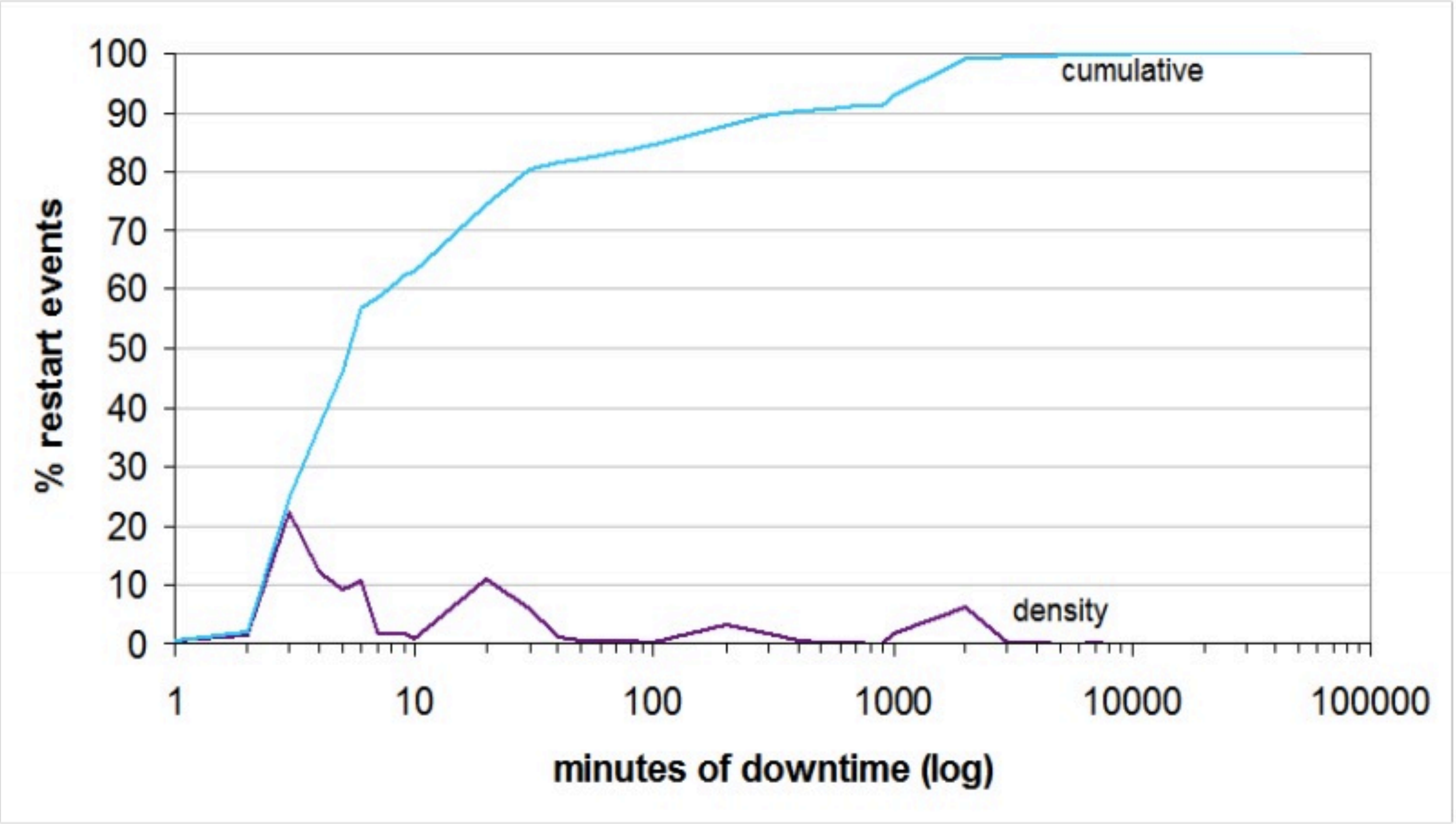
Typical first year for a new cluster:

- ~0.5 overheating (power down most machines in <5 mins, ~1-2 days to recover)
- ~1 PDU failure (~500-1000 machines suddenly disappear, ~6 hours to come back)
- ~1 rack-move (plenty of warning, ~500-1000 machines powered down, ~6 hours)
- ~1 network rewiring (rolling ~5% of machines down over 2-day span)
- ~20 rack failures (40-80 machines instantly disappear, 1-6 hours to get back)
- ~5 racks go wonky (40-80 machines see 50% packetloss)
- ~8 network maintenances (4 might cause ~30-minute random connectivity losses)
- ~12 router reloads (takes out DNS and external vips for a couple minutes)
- ~3 router failures (have to immediately pull traffic for an hour)
- ~dozens of minor 30-second blips for dns
- ~1000 individual machine failures
- ~thousands of hard drive failures
- slow disks, bad memory, misconfigured machines, flaky machines, etc.

Long distance links: wild dogs, sharks, dead horses, drunken hunters, etc.

Source: Dean

# Understanding Downtime Behavior Matters




Source: Dean

# Faults, Errors, and Failures

- Fault
  - Defect that has potential to cause problems
- Error
  - Wrong result caused by an active fault
- Failure
  - Unhandled error that causes interface to break its contract

# Fault Tolerance

- Error detection
  - Use limited redundancy to verify correctness
  - Example: detect damaged frames in link layer
  - **Fail fast:** report error at interface

- Error containment
  - Limiting propagation of errors
  - Example: enforced modularity
  - **Fail stop:** immediately stop to prevent propagation
  - **Fail safe**: transform wrong values into conservative "acceptable" values, but limiting operation
  - **Fail soft**: continue with only a subset of functionality

# Fault Tolerance

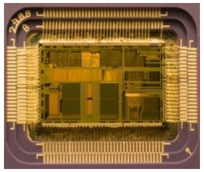
- Error masking
  - Ensure correct operation despite errors
  - Example: reliable transmission, process pairs
- We will focus on error masking
  - Main techniques

# Replication

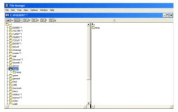
- **MAKE COPIES!!** ☺
  - State-machine replication
  - Asynchronous replication
    - Primary-Site
    - Peer-to-Peer
  - Synchronous replication
    - Read-Any, Write-All
    - Quorums

(loop (print (eval (read))))

Replicated Interpreter



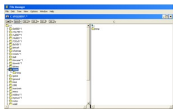
Replicated memory



- Techniques only good enough for a specific **failure model**
  - Nuclear holocaust
  - Component maliciously outputs random gibberish (**Byzantine**)
  - Components **crash** without telling you anything
  - Components are **fail-stop**

# Asynchronous Replication





# Asynchronous Replication

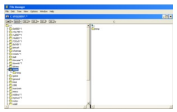
- Allows `WRITE`s to return before all copies have been changed
  - READs nonetheless look at subset of copies
  - Users must be aware of which copy they are reading, and that copies may be out-of-sync for short periods of time.

- Two approaches: Primary Site and Peer-to-Peer replication
  - Difference lies in how many copies are "*updatable*" or "*master copies*".

Source: Ramakrishnan & Gehrke (partial)

# Primary Site Replication

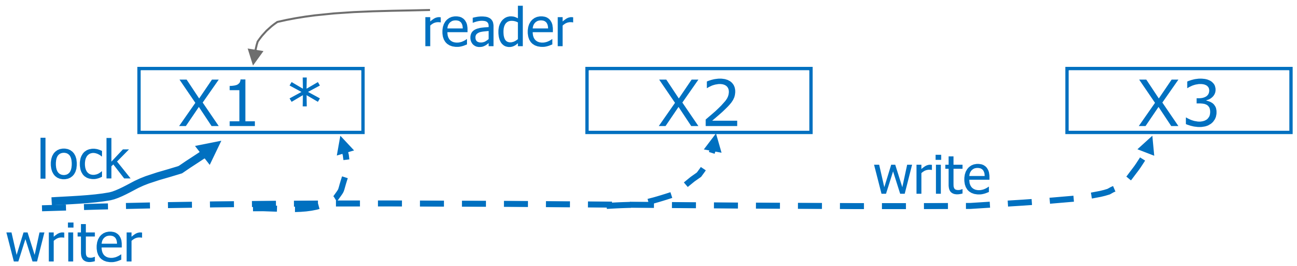




- Exactly one copy is designated the primary or master copy. Replicas at other sites cannot be directly updated
  - The primary copy is published
  - Other sites subscribe to this copy; these are secondary copies
- Main issue: How are changes to the primary copy propagated to the secondary copies?
  - Done in two steps: First, `CAPTURE` changes made at primary; then `APPLY` these changes
  - Many possible implementations for `CAPTURE` and `APPLY`

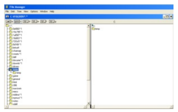
Source: Ramakrishnan & Gehrke (partial)

# Primary copy




- Writers lock & update primary copy and propagate the update to other copies
- Readers lock and access primary copy
- Widely adopted, e.g. many database systems





# Peer-to-Peer Replication

- More than one of the copies of an object can be a master in this approach
  - Changes to a master copy must be propagated to other copies
  - If two master copies are changed in a conflicting manner, this must be resolved. (e.g., Site 1: Joe's age changed to 35; Site 2: to 36)

- Best used when conflicts do not arise
- **Examples**
  - Each master site owns a disjoint fragment of the data
  - Updating rights owned by one master at a time
  - Operations are associative-commutative

Source: Ramakrishnan & Gehrke (partial)

# Eventual consistency

- If no new updates are made to an object, after some inconsistency window closes, all accesses will return the same "last" updated value

- **Prefix property:**
  - If Host 1 has seen write $w_{i,2}$: $i^{th}$ write accepted by host 2
  - Then 1 has all writes $w_{j,2}$ (for $j<i$) accepted by 2 prior to $w_{i,2}$

How do we know the precedence relationship?

- Assumption: write conflicts will be easy to resolve
  - Even easier if whole-"object" updates only

Source: Freedman

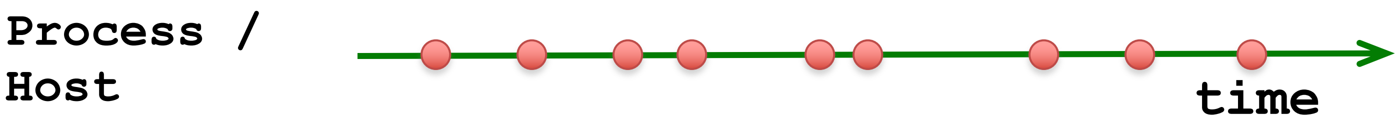
# Events and Histories

- Processes execute sequences of events
- Events can be of 3 types:
  - local, send, and receive
- The local history $h_p$ of process p is the sequence of events executed by process

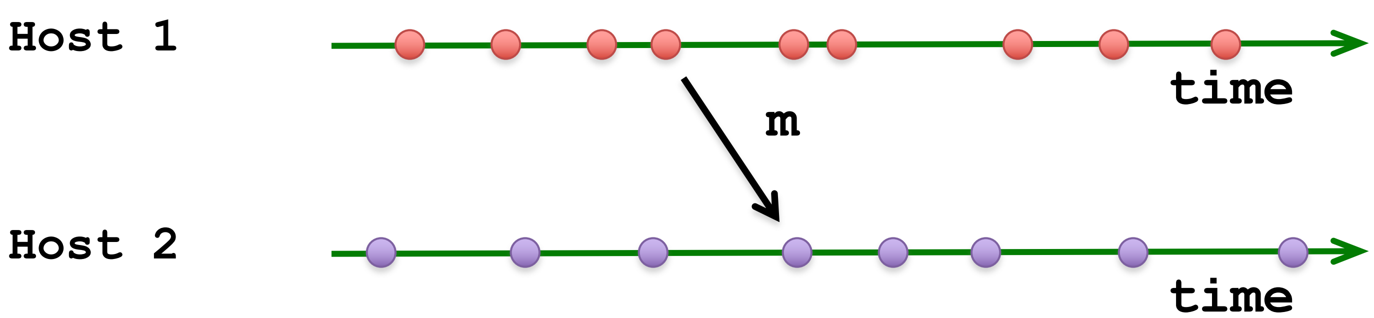
Source: Freedman

# Ordering Events

- **Observation 1:**
  - Events in a local history are <u>totally ordered</u>




- **Observation 2:**
  - For every message m, send(m) precedes receive(m)




Source: Freedman

# Happens-Before (Lamport [1978])

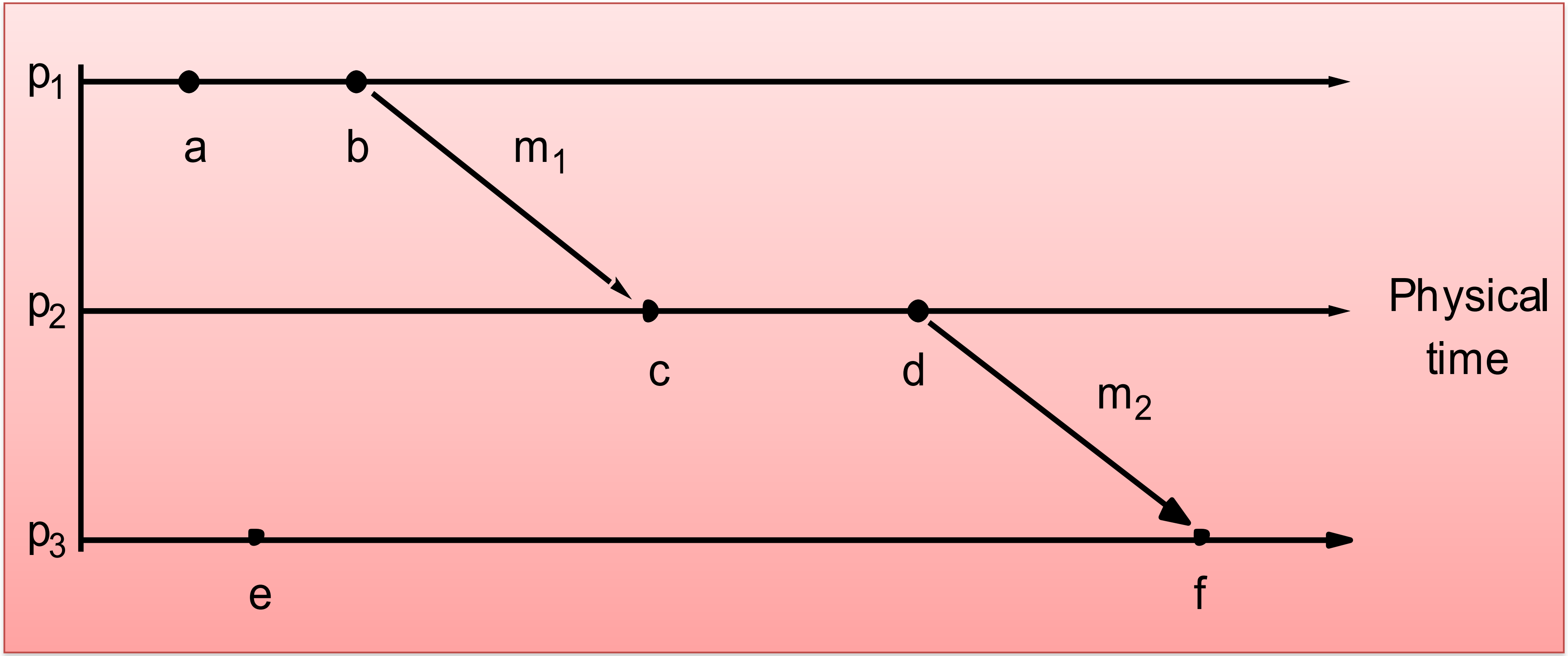
- **Relative time? Define Happens-Before ($\rightarrow$) :**
  - On the same process: $a \rightarrow b$, if time(a) < time(b)
  - If p1 sends m to p2: send(m) $\rightarrow$ receive(m)
  - Transitivity: If $a \rightarrow b$ and $b \rightarrow c$ then $a \prec c$

- **Lamport Algorithm establishes partial ordering:**
  - All processes use counter (clock) with initial value of 0
  - Counter incremented / assigned to each event as timestamp
  - A send (msg) event carries its timestamp
  - For receive (msg) event, counter is updated by

```
max (receiver-counter, message-timestamp) + 1
```

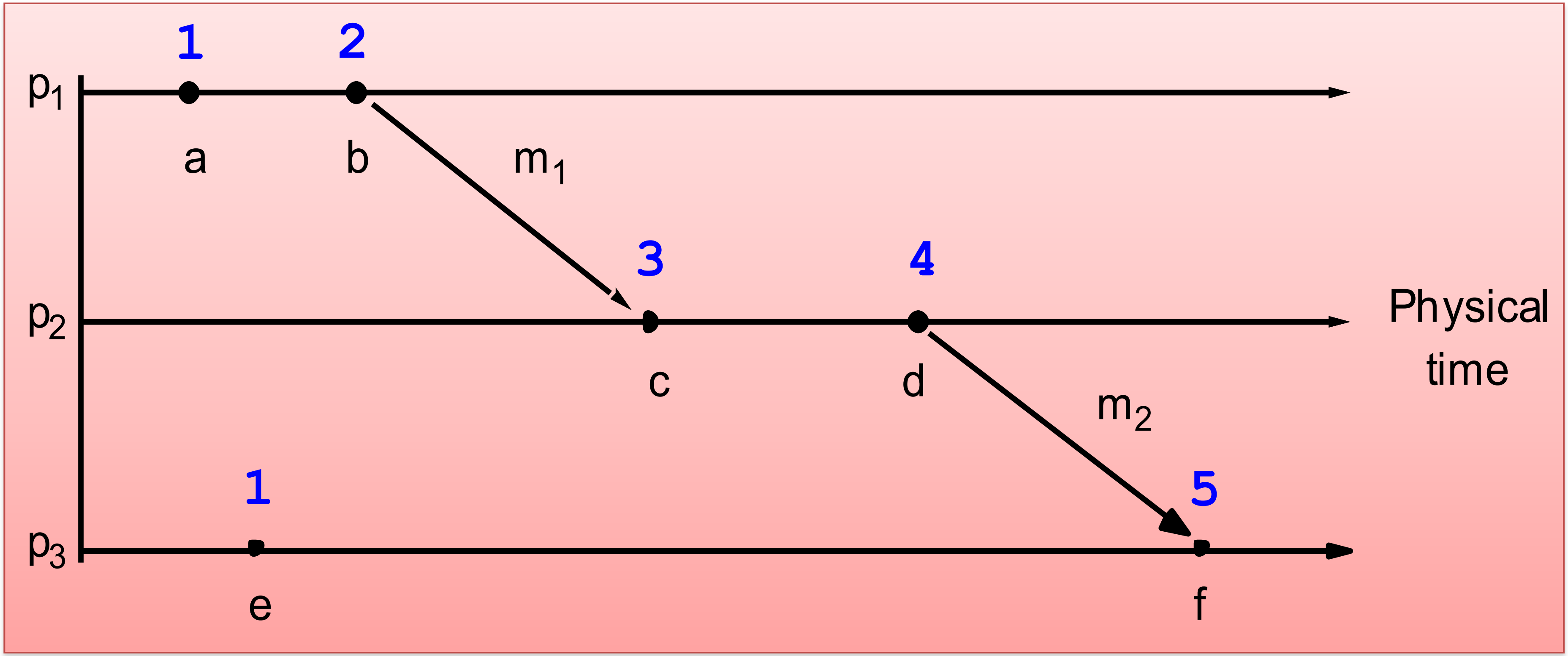
Source: Freedman

# Events Occurring at Three Processes




Source: Freedman

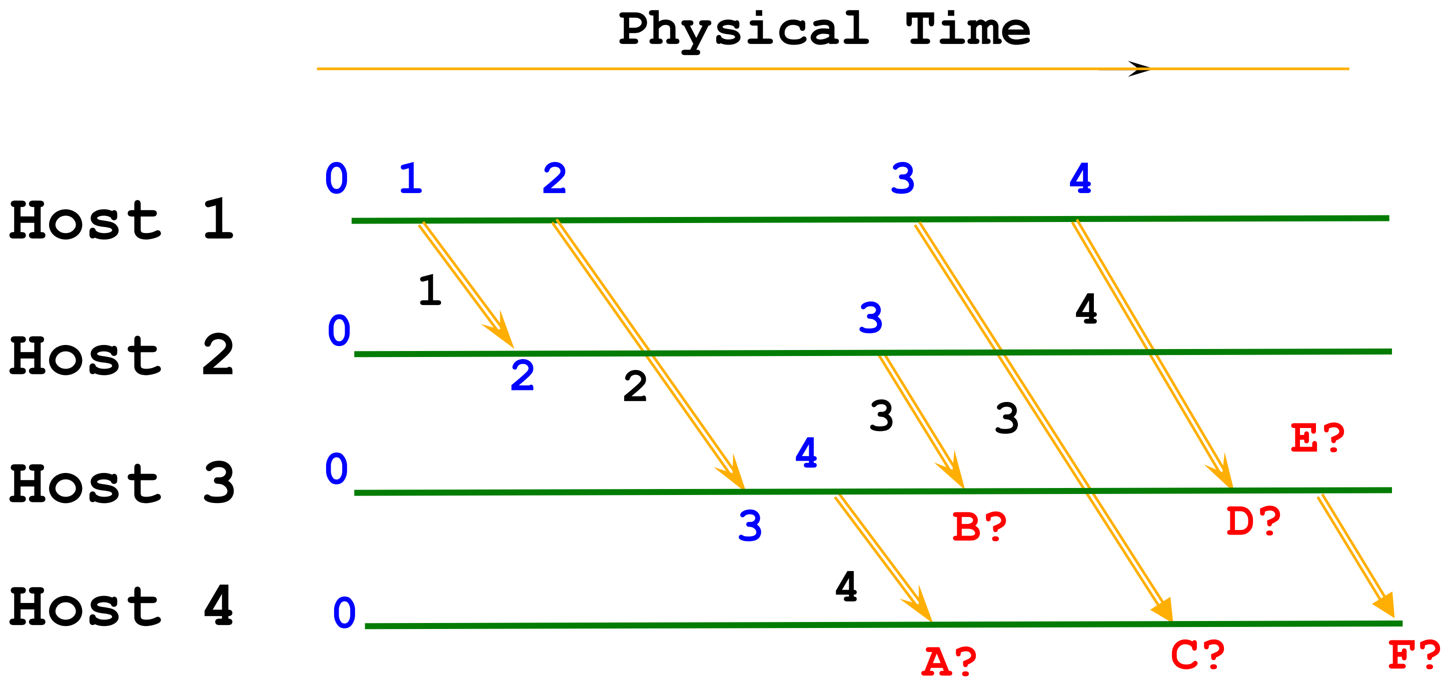
# Lamport Timestamps




Source: Freedman

# Lamport Logical Time

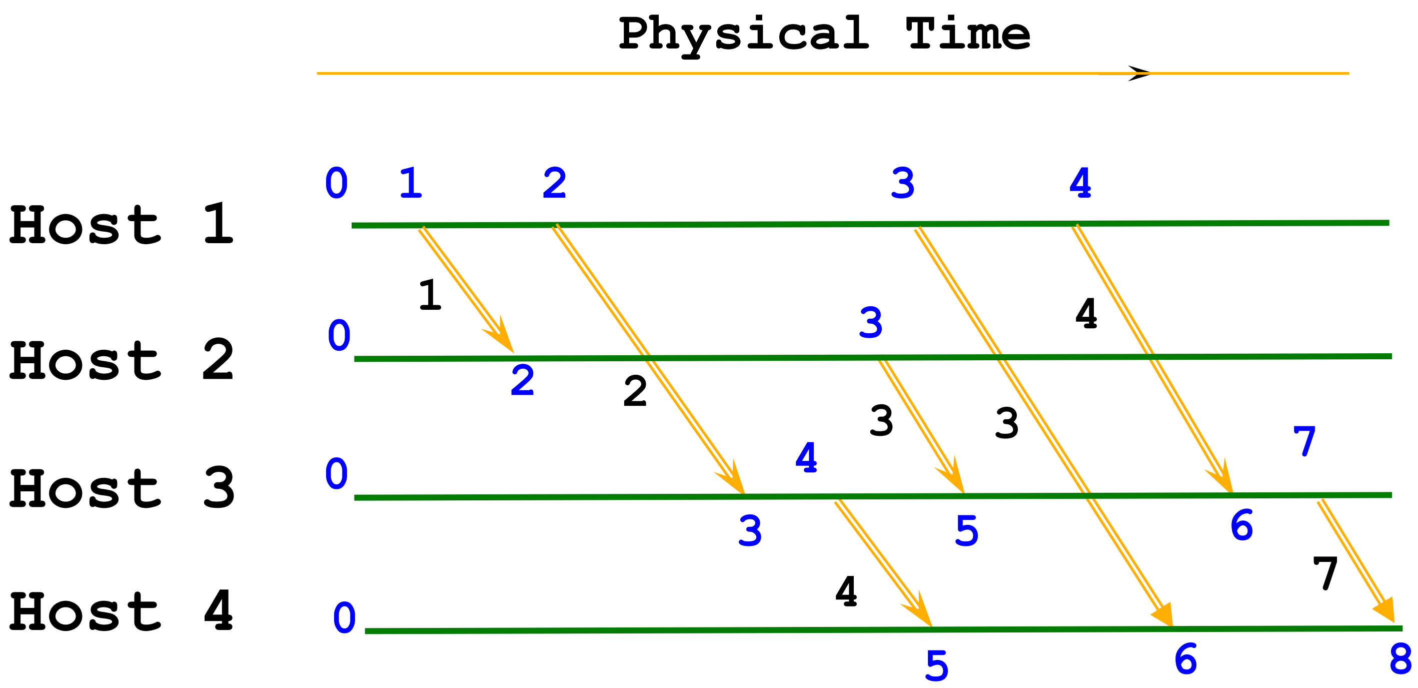
- Fill in the missing values, A-F:




Source: Freedman (partial)

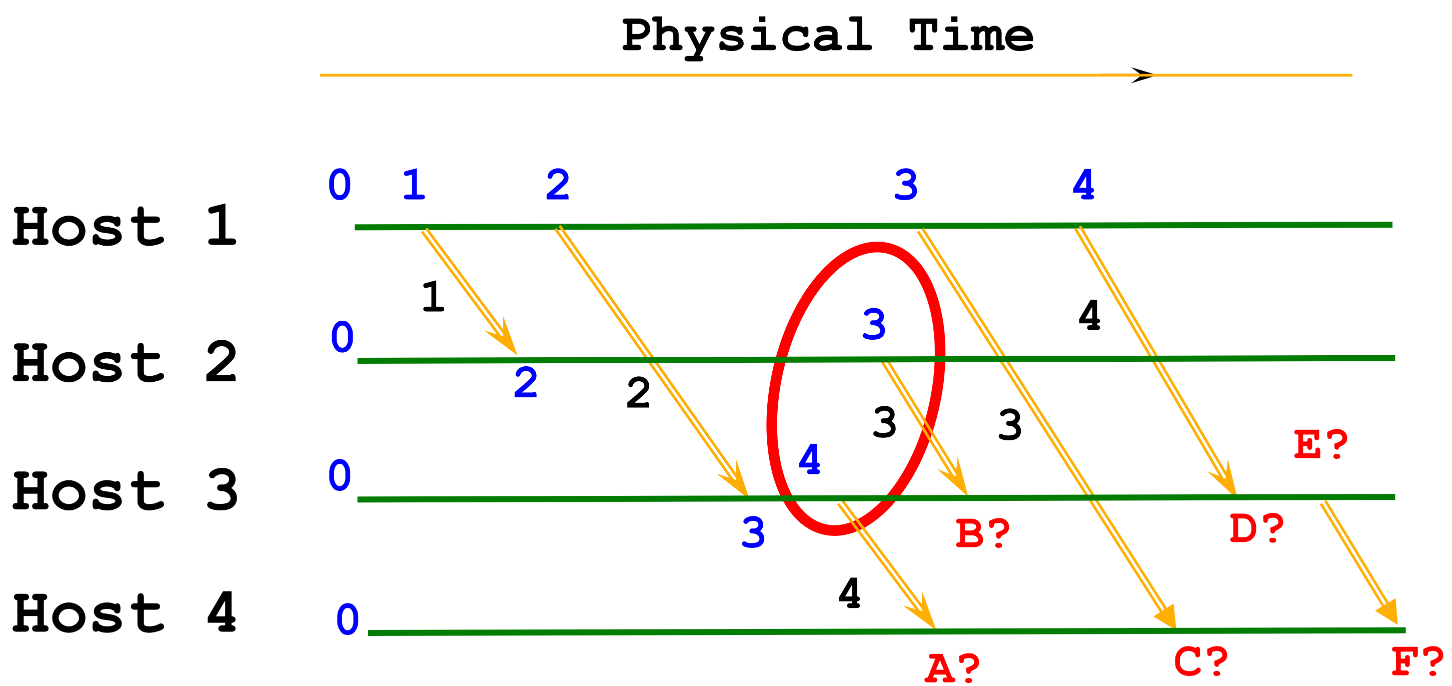
# Lamport Logical Time

- Fill in the missing values, A-F:




Source: Freedman (partial)

# Lamport Logical Time




Can we say: if timestamp(e) < timestamp (f) then e precedes f ?

**Logically concurrent events!**

Source: Freedman

# Vector Logical Clocks

- **With Lamport Logical Time**
  - e precedes f $\Rightarrow$ timestamp(e) < timestamp (f), but
  - timestamp(e) < timestamp (f) $\cancel{\Rightarrow}$ e precedes f

Source: Freedman

# Vector Logical Clocks

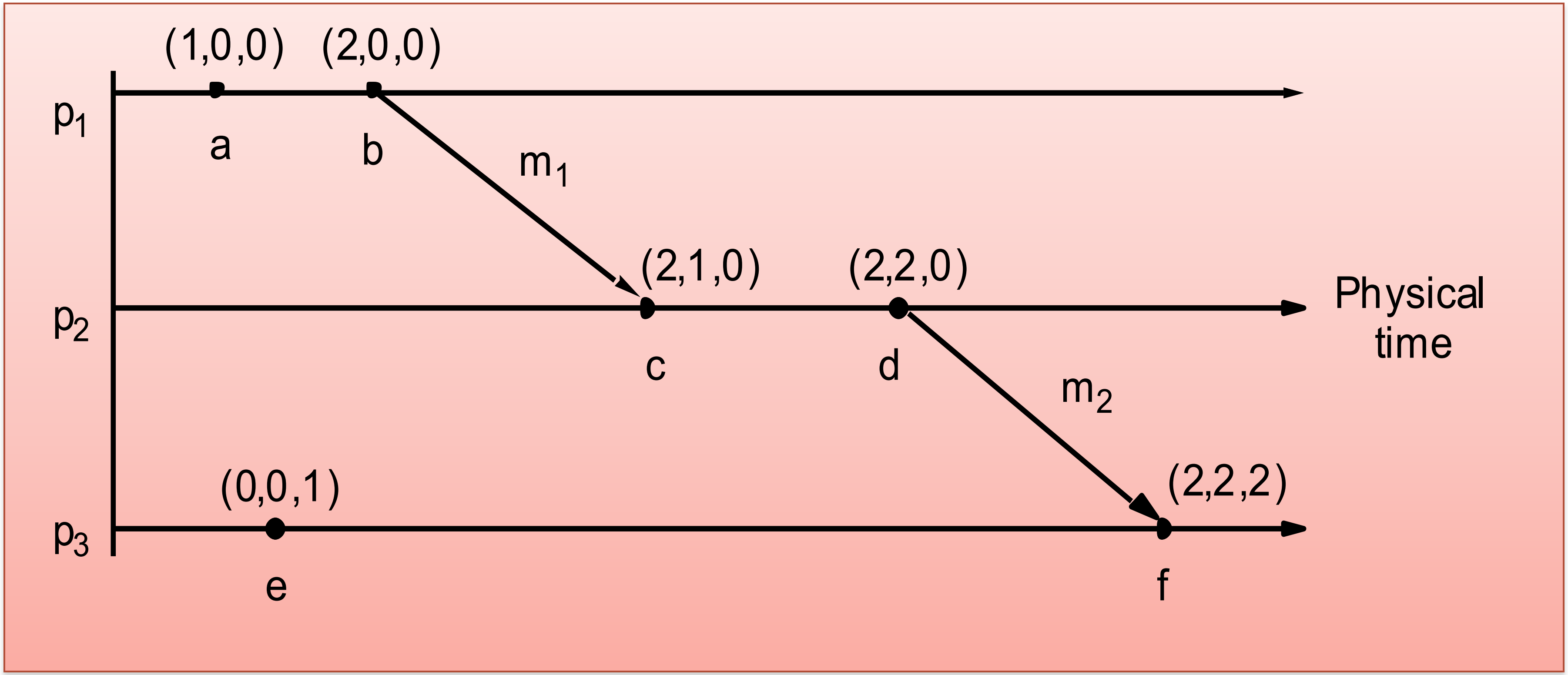
- **With Lamport Logical Time**
  - e precedes f $\Rightarrow$ timestamp(e) < timestamp (f), but
  - timestamp(e) < timestamp (f) $\cancel{\Rightarrow}$ e precedes f

- Vector Logical time guarantees this:
  - All hosts use a vector of counters (logical clocks),
  - ith element is the clock value for host i, initially 0
  - Each host i, increments the ith element of its vector upon an event, assigns the vector to the event.
  - A send(msg) event carries vector timestamp
  - For receive(msg) event,

$$\mathbf{V_{receiver}[j]} = \begin{cases} \text{Max } (V_{receiver}[j]\,,\, V_{msg}[j]), & \text{if j is not self} \\ V_{receiver}[j] + 1 & \text{otherwise} \end{cases}$$

Source: Freedman

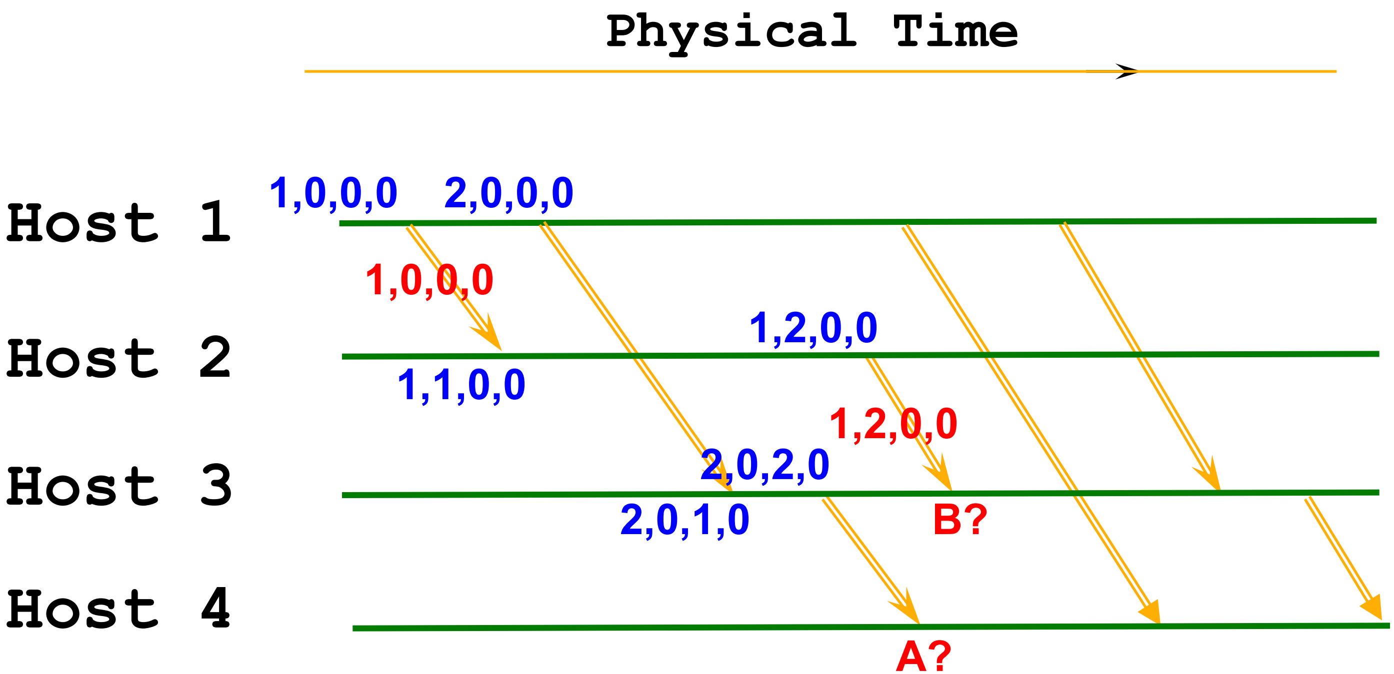
# Vector Timestamps




Source: Freedman

# Vector Logical Time

- Fill in the missing values, A and B:

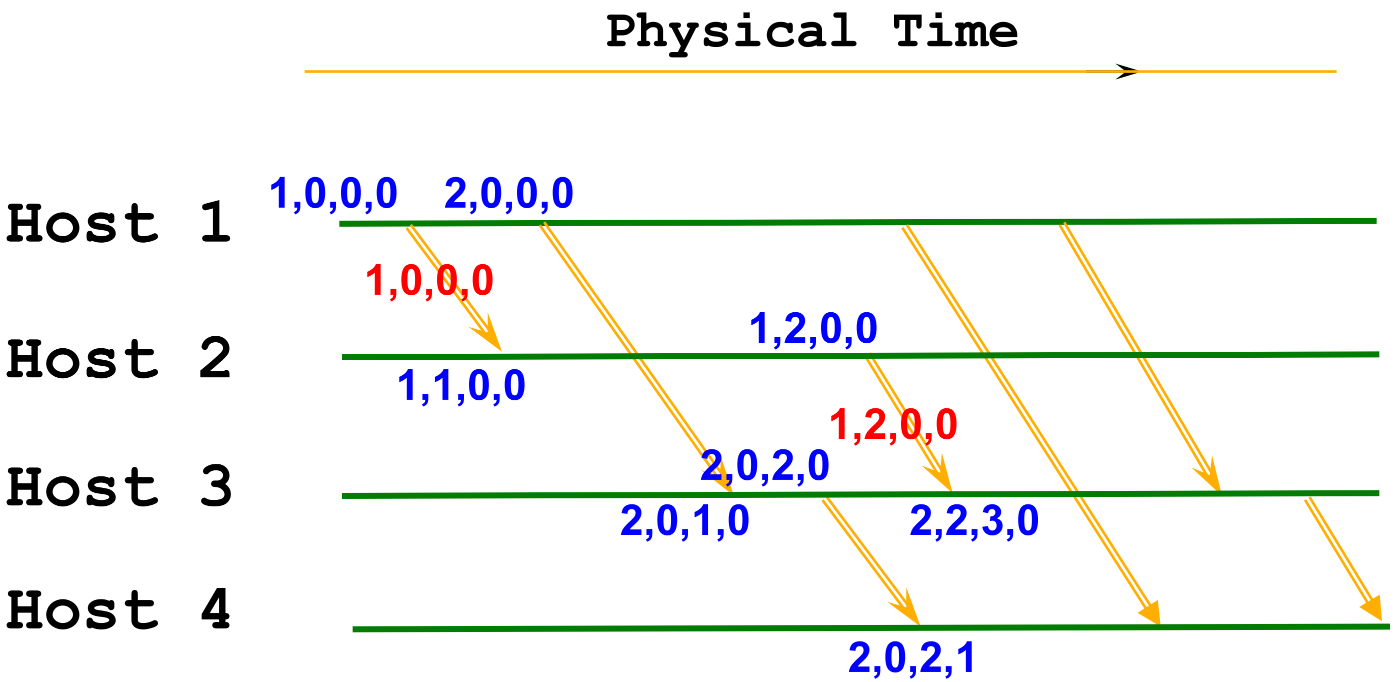



$$V_{receiver}[j] = \begin{cases} \text{Max } (V_{receiver}[j]\ ,\ V_{msg}[j]), & \text{if } j \text{ is not self} \\ V_{receiver}[j] + 1 & \text{otherwise} \end{cases}$$

Source: Freedman

# Vector Logical Time

- Fill in the missing values, A and B:




$$V_{receiver}[j] = \begin{cases} \text{Max } (V_{receiver}[j]\,,\, V_{msg}[j]), \text{ if j is not self} \\ V_{receiver}[j] + 1 \qquad \text{otherwise} \end{cases}$$

Source: Freedman

# Comparing Vector Timestamps

- a = b if they agree at every element
- a < b if a[i] <= b[i] for every i, but !(a = b)
- a > b if a[i] >= b[i] for every i, but !(a = b)
- a || b if a[i] < b[i], a[j] > b[j], for some i,j (conflict!)

- If one history is prefix of other, then one vector timestamp < other
- If one history is not a prefix of the other, then (at least by example) VTs will not be comparable.

Source: Freedman

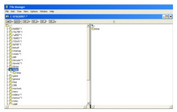
# Eventual is not the only choice

- Host of other properties available
  - Beyond our scope!
- **Examples**
  - Strong consistency
  - Weak consistency
  - Causal consistency
  - Read-your-writes consistency
  - Session consistency
  - Monotonic read consistency
  - Monotonic write consistency
- See Werner Vogels' entry http://www.allthingsdistributed.com/2007/12/eventually_consistent.html for informal overview, or a good distributed systems book for algorithms ☺

# Synchronous Replication

# Synchronous Replication





- Hide replication behind `READ`/`WRITE` memory abstraction

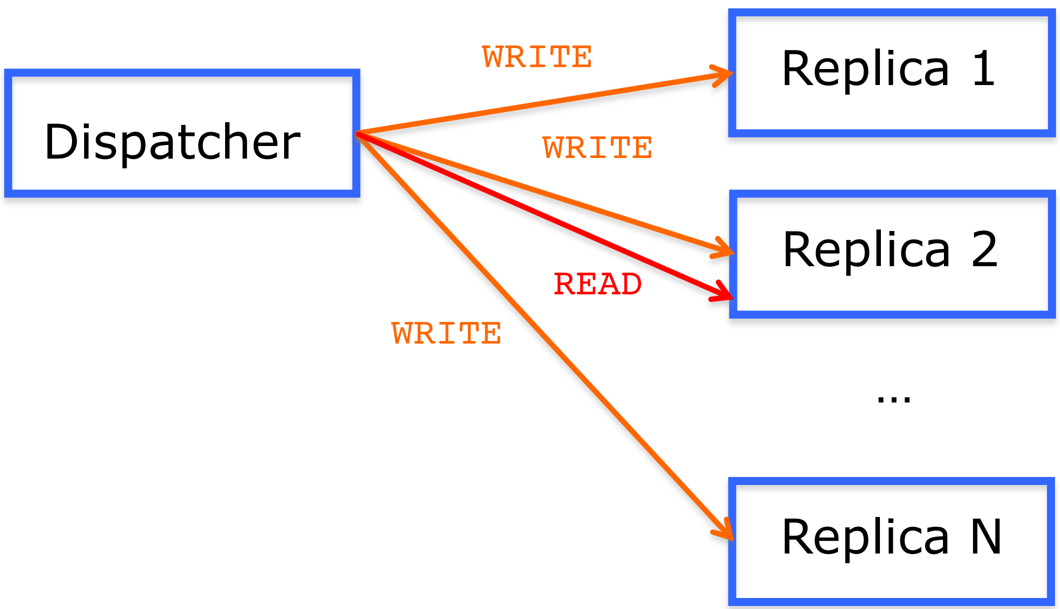
- Program operates against memory

- Memory makes sure `READ`s and `WRITE`s are **atomic**
  - **All-or-nothing:** either in all correct replicas or none
  - **Before-or-after:** Equivalent to a total order

- Memory replicates data for fault tolerance

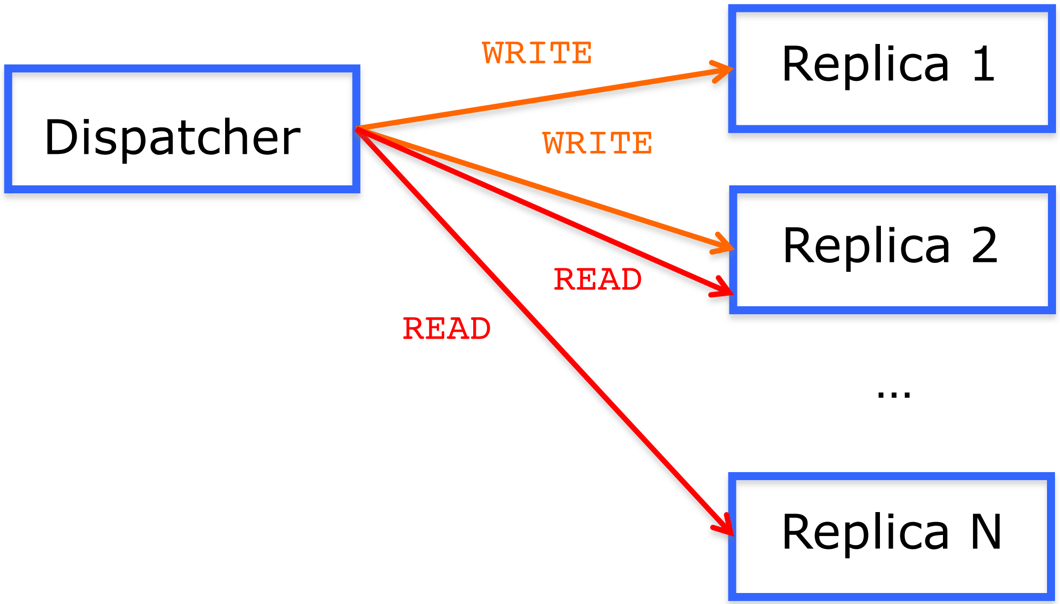
# Synchronous Replication

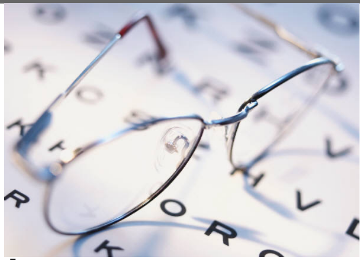
- Read Any, Write-All
  - For now assume we have a centralized Dispatcher → state-machine replication algorithms drop that assumption!
- WRITEs synchronously sent everywhere
- But READs can be answered by any replica



# Synchronous Replication

- Quorums
  - Read Quorum ($Q_r$) / Write Quorum ($Q_w$)
  - $Q_r + Q_w > N_{replicas}$
- Reads or writes only succeed if same response is given by respective quorum
  - Read any, Write all case is $Q_w = N_{replicas}$, $Q_r = 1$





# What should we learn today?

- Explain and apply common fault-tolerance strategies such as error detection, containment, and masking
- Explain techniques for redundancy, such as n-version programming, error coding, duplicated components, replication
- Categorize main variants of replication techniques and implement simple replication protocols
- Explain the difficulties of guaranteeing atomicity in a replicated distributed system
- Discuss consistency properties in a replicated system and the notion of eventual consistency