Fault-Tolerance II: Replication, Time and Consistency

COS 518: *Advanced Computer Systems*
Lecture 4

Kyle Jamieson

[Credits: Selected content from Mike Freedman, Jinyang Li, Robert Morris]

Widely-separated replicas

- Goal: Provide a **highly-durable** service
  - If the application or OS damages the data, all replicas will **suffer the same damage**
  - Co-located replicas are **vulnerable to the environment**
- Principle: Multiple copies, **widely separated** and **independently-administered**
  - **Goal:** Provide service identical to the non-replicated version
    - Except more reliable, and perhaps slower
- Separate the replicas! **Problem: High-latency, fundamentally-unreliable communication** between distant points

Today

1. **Two-Phase Commit**

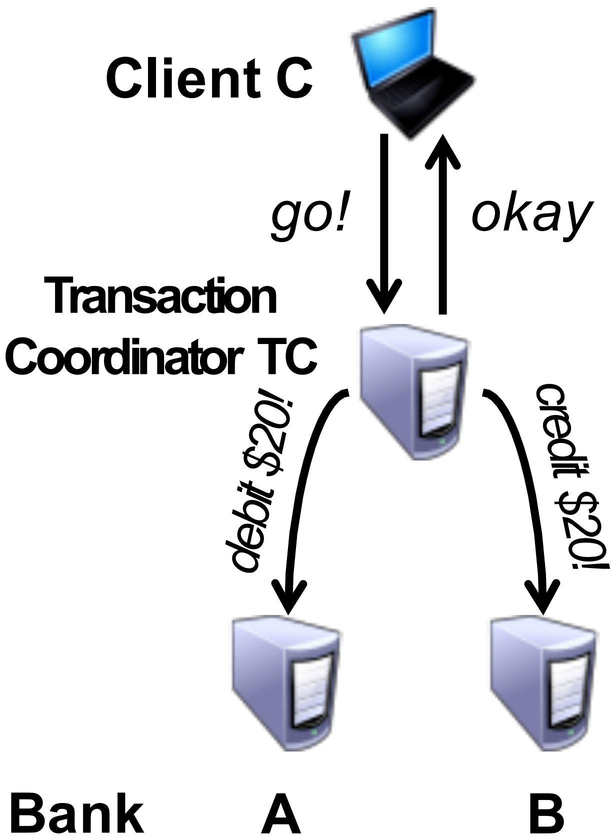
2. Replication and agreement with Paxos

3. Time, events, and consistency

Two-Phase Commit (2PC)

- **Goal:** General purpose, distributed agreement on some action, with failures
  - Different entities play different roles in the action
- **Running example:** Transfer money from Bank A to Bank B
  - Debit at A, credit at B, tell the client "okay"
  - Require **both** banks to do it, or **neither**
  - Require that **one bank never act alone**
- This is an **all-or-nothing** atomic commit protocol
  - Later will discuss how to make it **before-or-after** atomic

Straw Man protocol



1. **C → TC**: *"go!"*

2. **TC → A**: *"debit $20!"*
   TC → B: *"credit $20!"*
   TC → C: *"okay"*

- **A, B** perform actions on receipt of messages

Reasoning about the Straw Man protocol

- What could **possibly** go wrong?

1. Not enough money in **A's** bank account?
2. **B's** bank account no longer exists?
3. **A** or **B** **crashes** before receiving message?
4. The best-effort network to **B** **fails**?
5. **TC** **crashes** after it sends *debit* to **A** but before sending to **B**?

Correctness versus liveness

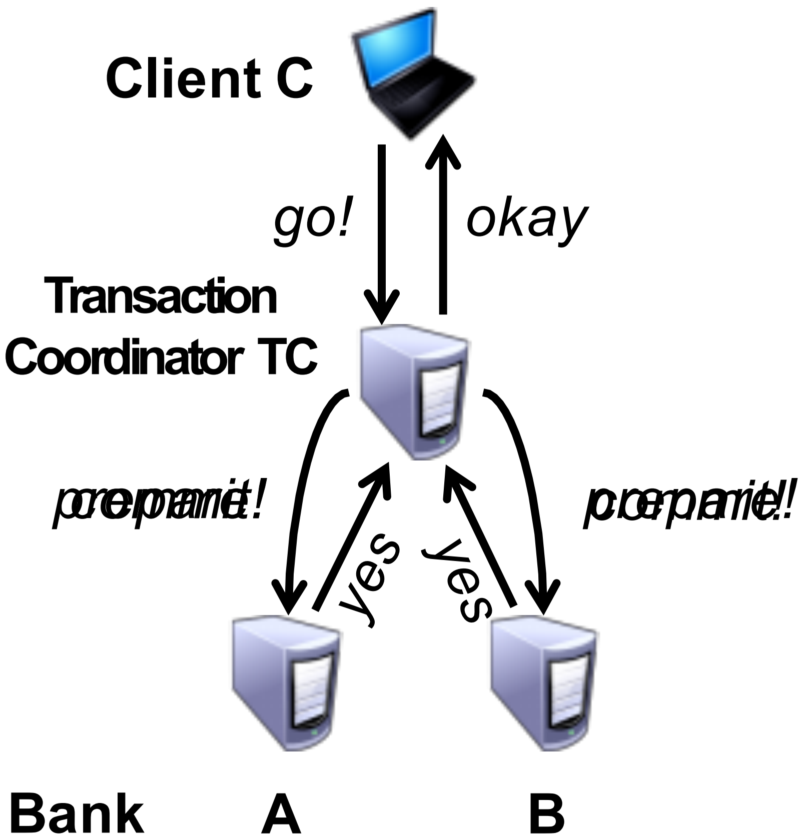
- Note that TC, A, and B each have a notion of committing

- We want two properties:

1. Correctness
   - If **one commits, no one aborts**
   - If **one aborts, no one commits**

2. Liveness
   - If **no failures** and **A** and **B** can commit, **action commits**
   - If **failures,** reach a conclusion ASAP

A *correct* atomic commit protocol



1. **C → TC:** *"go!"*
2. **TC → A, B:** *"prepare!"*
3. **A, B → P:** *"yes"* or *"no"*
4. **TC → A, B:** *"commit!"* or *"abort!"*
   - TC sends *commit* if **both** say *yes*
   - TC sends *abort* if **either** say *no*

   TC → C: *"okay" or "failed"*

- As before, **A, B** commit on receipt of commit message

Reasoning about atomic commit

- *Why is this correct?*
  – Neither can commit unless both agreed to commit

- *What about performance?*
  – Crashes or message loss can still prevent completion

  – Two types of problems:

1. **Timeout:** I'm up, but didn't receive a message I expected
   - Maybe other node crashed, maybe network broken
2. **Reboot:** Node crashed, is rebooting, must clean up

Timeouts in atomic commit

- Where do hosts **wait** for messages?

1. **TC** waits for "yes" or "no" from **A** and **B**
   - **TC** hasn't yet sent any commit messages, so **can safely abort** after a timeout
   - But this is **conservative:** might be the net
     - We've preserved correctness, sacrificed performance

2. **A** and **B** wait for "commit" or "abort" from **TC**
   - If it sent a *no*, it **can safely abort** *(why?)*
   - If it sent a *yes*, can it unilaterally abort?
   - Can it unilaterally commit?

   - A, B could wait forever, but there is an alternative…

Atomic commit: Server termination protocol

- Consider Server **B** (Server **A** case is symmetric) waiting for *commit* or *abort* from **TC**
  – Assume **B** voted *yes* (else, unilateral abort possible)

- **B** → **A:** "status?" **A** then replies back to **B.** Four cases:

1. (No reply from **A**): no decision, **B** waits for **TC**
2. Server **A** received commit or abort from **TC:** Agree with the **TC's** decision
3. Server **A** hasn't voted yet or voted *no:* both **abort**
   – **TC** **can't have** decided to commit
4. Server A voted *yes:* both must **wait** for the **TC**
   – **TC** decided to *commit* if both replies received
   – **TC** decided to *abort* if it timed out

Reasoning about the server termination protocol

- *What are the correctness and liveness properties?*

- Can resolve **some** timeout situations with guaranteed correctness

- Sometimes however **A** and **B** must block
  - Due to failure of the **TC** or network to the **TC**

- But what will happen if **TC, A,** or **B** **crash and reboot?**

How to handle crash and reboot?

- Can't back out of commit if already decided
  - **TC** crashes just after sending *"commit!"*

  - **A** or **B** crash just after sending *"yes"*
    - Big Trouble if they reboot and don't remember saying "yes"
      - They might change their minds after reboot

- If all nodes knew their state before crash, we could use the termination protocol…
  - We already know how to solve this problem: **write-ahead log** *"commit!" and "yes"* to disk
    - **Review:** *Why write log before sending* "commit!" *or* "yes"?

Recovery protocol with non-volatile state

- If everyone rebooted and is reachable, TC can just check for **commit** record on disk and **resend** action
- **TC:** If no **commit** record on disk, **abort**
  - You didn't send any *"commit!"* messages
- **A, B:** If no **yes** record on disk, **abort**
  - You didn't vote *"yes"* so **TC couldn't have** committed
- A, B: If **yes** record on disk, execute termination protocol
  - This might block

Two-Phase Commit

- This recovery protocol with non-volatile logging is called ***Two-Phase Commit (2PC)***
- *What properties does it have?*
- **Correct:** All hosts that decide reach the same decision
  - No commit unless everyone says "yes"
- **Live:** If no failures and all say "yes" then commit
  - **But if failures then 2PC might block**
  - **TC must be up to decide**
- **Doesn't tolerate faults well: must wait for repair**

Today

1. Two-Phase Commit

2. **Replication and agreement with Paxos**

3. Time, events, and consistency

State machine replication

- Idea: **Replicate data** on multiple servers
  - If servers fail, hopefully others can step in

- **Any server** is essentially a **state machine**
  - Disk, RAM, CPU registers are **state**
  - Instructions **transition** among states
  - User requests cause instructions to be executed, so cause transitions among 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

Primary-Backup (P-B) approach

- Nominate one "special" server: the ***primary***
  - Call all other servers ***backups***

- Clients send all operations to current primary

- The primary's role:
  - Chooses **order** for clients' operations
  - **Sends** clients' operations to backups
  - **Replies** to clients' operation requests

- What if the primary fails?

Primary failure

- Last operation received by primary might not be complete
  - Need to pick a new primary

- *Couldn't we just add a spare primary?*
  - But what if original node **wasn't dead** but just **slow?**
  - Now backups get mixed messages from "primaries"

- Define lowest-numbered server as the primary
  - After failure, everyone pings everyone
  - Does everyone now know who the new primary is?
    - **Not necessarily:** pings lost, delayed, or network partition result in **two primaries**

Agreement is hard

- Fundamentally, the issue revolves around membership
  - In asynchronous environment, can't detect failures reliably

- Suppose Servers 1, 2 agree on a primary; 3, 4 don't respond

- Are we done?
  - Agreement must complete even with failed servers

  - Can't distinguish failed server from network partition
    - So 3, 4 may be partitioned, agreed on **different primary!**

Key idea: Majority consensus

- Require a **majority** of nodes to agree on a primary
  - At most one network partition can contain a majority
  - If pings lost and thus two potential primaries, the majorities must overlap
    - Node(s) in the **overlap** see both potential primaries and **raise the alarm** about the disagreement

Paxos: High-level outline

1. Elect a replica to be ***leader***
   - Nodes send numbered ***prepare*** messages to everyone
   - Respond with ***promise*** messages, promising to **reject** a **lower-numbered** proposal in the **next** step

2. On receiving a **majority** of nodes' "promises," ***a* leader** sends a numbered ***accept*** message to propose a value
   - Respond with ***accept-ok*** messages **if it is the highest-numbered** proposal they've seen

3. If a **majority** of nodes "accept-ok," leader sends ***commit*** message to notify replicas

Paxos: Functionality and state

- Both "proposer" and "acceptor" run at all replicas
- Acceptor state at **each** node running Paxos:
  – Must persist across reboots

1. n_p: Greatest proposal number seen in a **prepare (init: -)**
2. n_a: Greatest proposal number seen in an **accept (init: -)**
   – v_a: Value seen in that **accept** message (init: -)

Paxos: In more detail

1. **Elect a replica to be a leader**

- Leader broadcasts ***prepare***(*n*) message
  - – Choose *n*, unique and higher than any *n* seen so far
  - – Broadcast **prepare**(*n*) to everyone including self

- Acceptor's **prepare**(*n*) handler:
  - – if $n > n_p$: $n_p \leftarrow n$ and reply ***promise***(n, n_a, v_a)
    - else reply ***prepare-reject***

Paxos: In more detail

1. Elect a replica to be a leader

- Leader broadcasts ***prepare***(n) message
  - Choose n, unique and higher than any n seen so far
  - Broadcast **prepare**(n) to everyone including self
- Acceptor's **prep**…) handler:
  - if $n > n_p$: $n_p \leftarrow$ … ply ***promise***(n, n_a, v_a)
    - else reply ***pr***… ***ject***

> - n = ⟨counter, unique node ID⟩
> - Leader increments counter
> - Generates higher n than any seen before
> - Node ID makes *n unique*

Paxos: In more detail

1. Elect a replica to be a **leader**

2. **Two-phase commit**

- If leader received promise from majority it:
  - $v^* \leftarrow v_a$ of **promise** with highest n_a (else, its own v)
  - Broadcasts ***accept***(n, v^*) to everyone

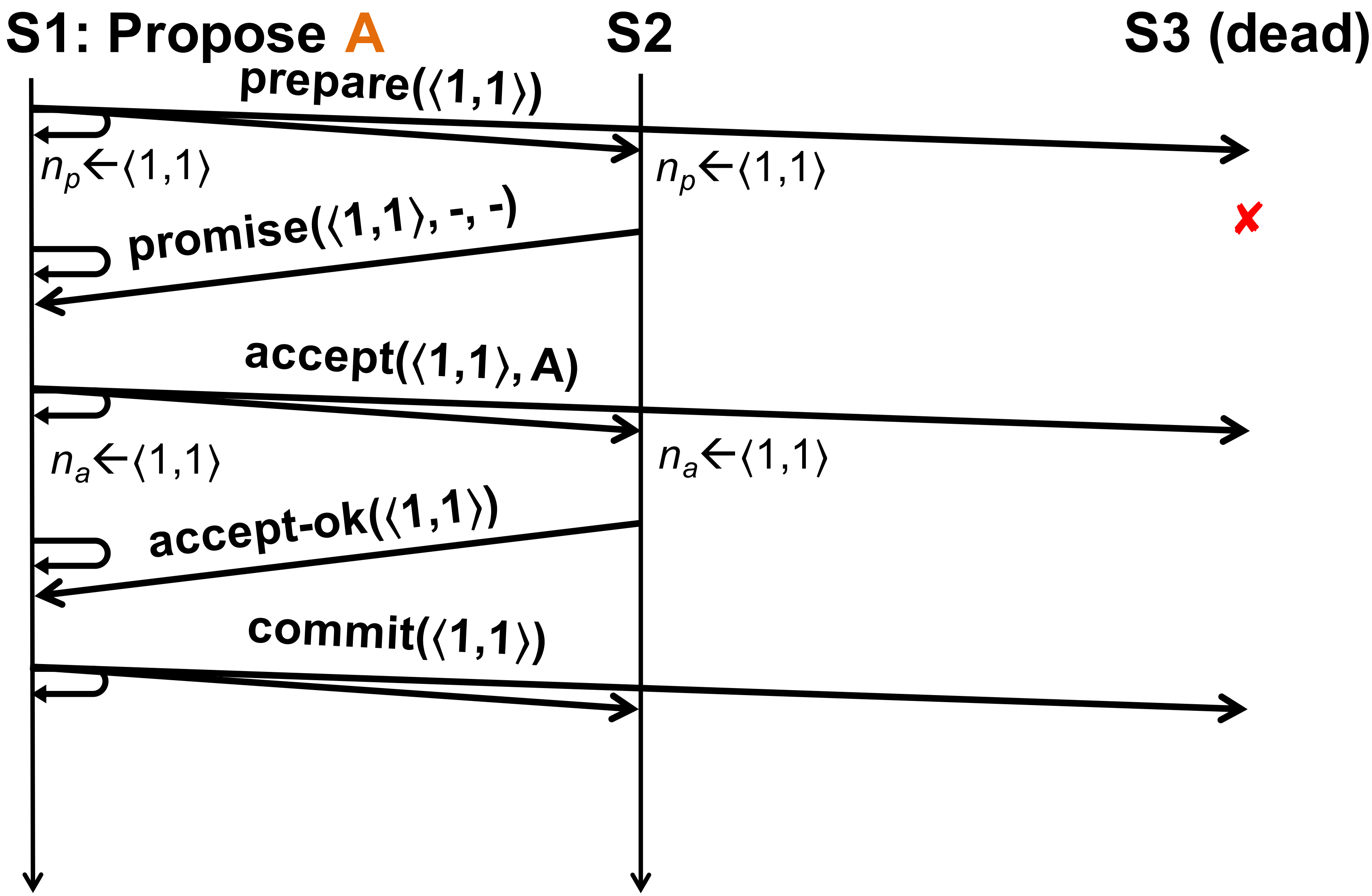
- Acceptor's **accept**(n, v) handler:
  - if $n \geq n_p$: $(n_p, n_a, v_a) \leftarrow (n, n, v)$ and reply **accept-ok**(n)
  - else, reply **accept-reject**

Paxos: In more detail

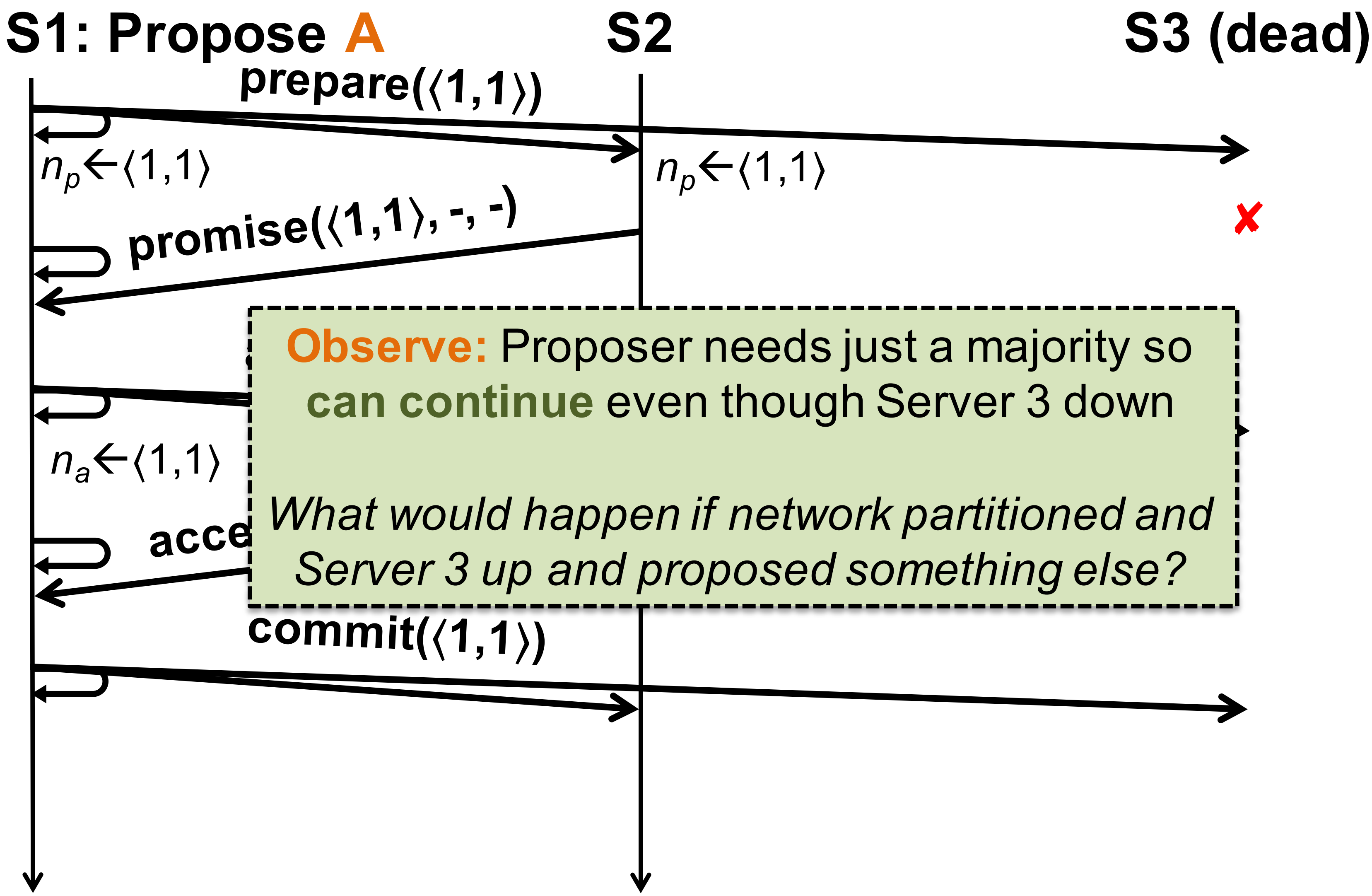
1. Elect a replica to be a **leader**
2. Two-phase commit
3. **Notify replicas**

- If leader received accept-ok from a **majority** of nodes:
  - Broadcasts **commit(*n*)** message to everyone

Example 1: Normal operation, one failure



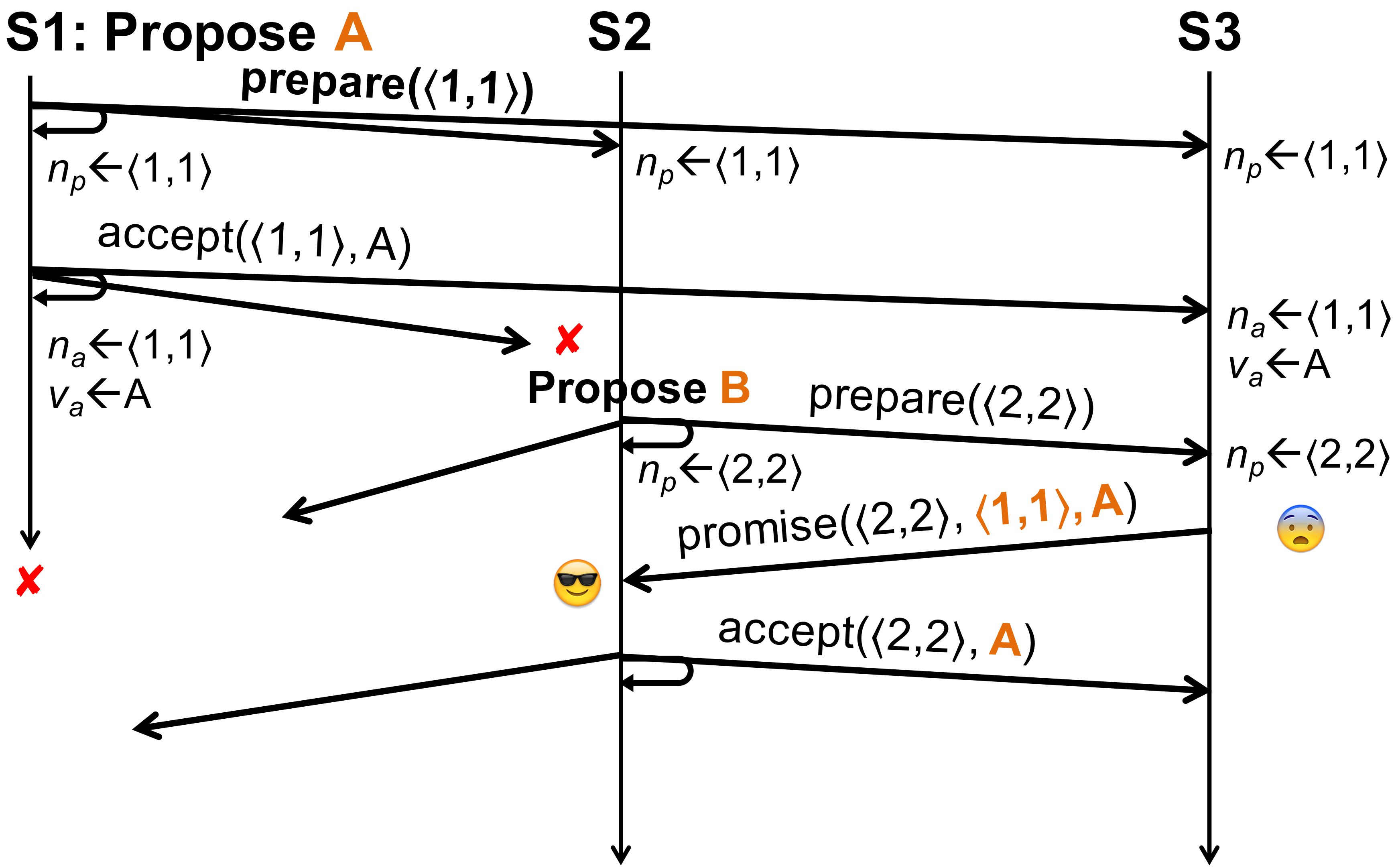
Example 1: Normal operation, one failure



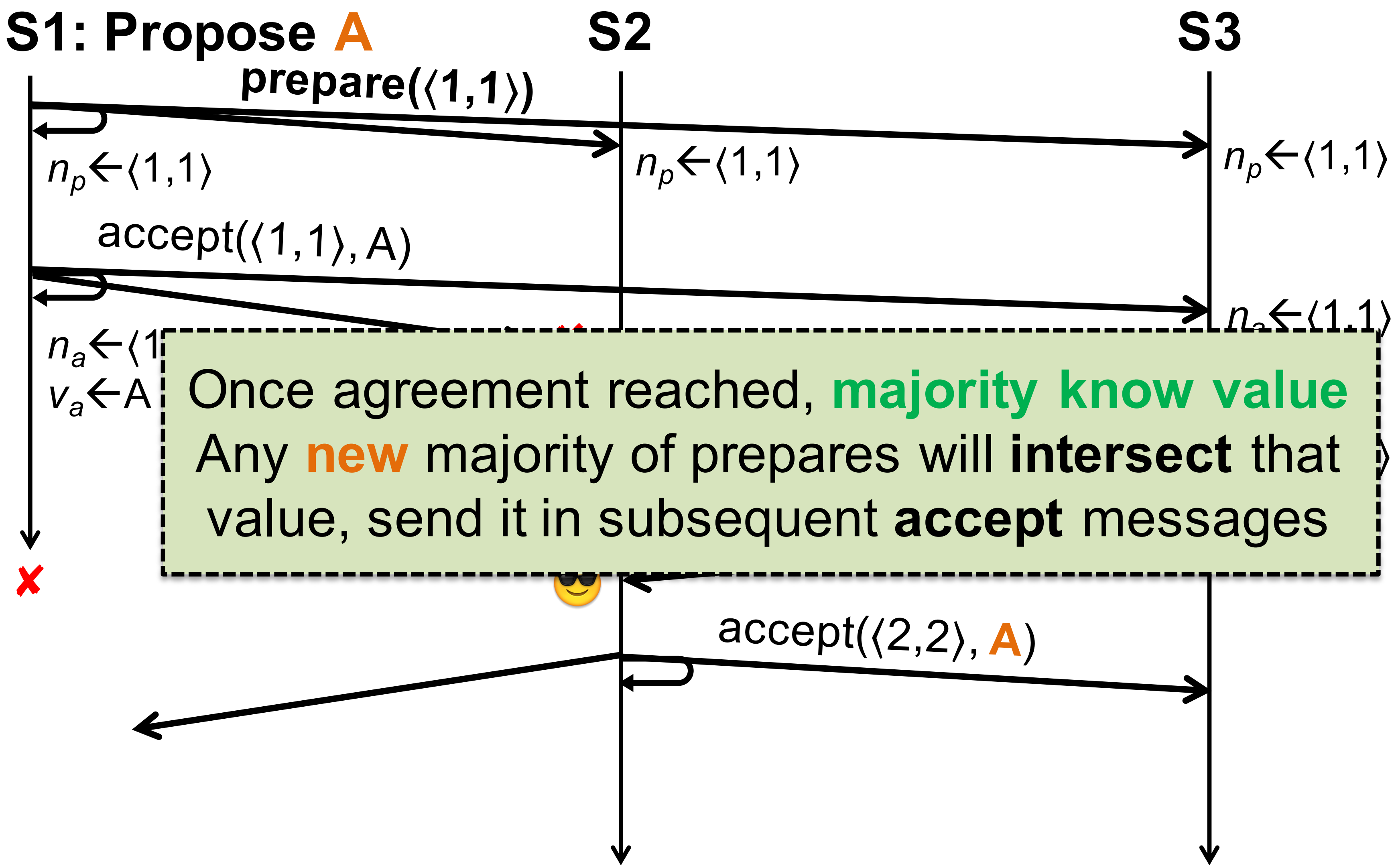
Observe: Proposer needs just a majority so **can continue** even though Server 3 down

What would happen if network partitioned and Server 3 up and proposed something else?

Example 2: Propagating agreement in the presence of failures



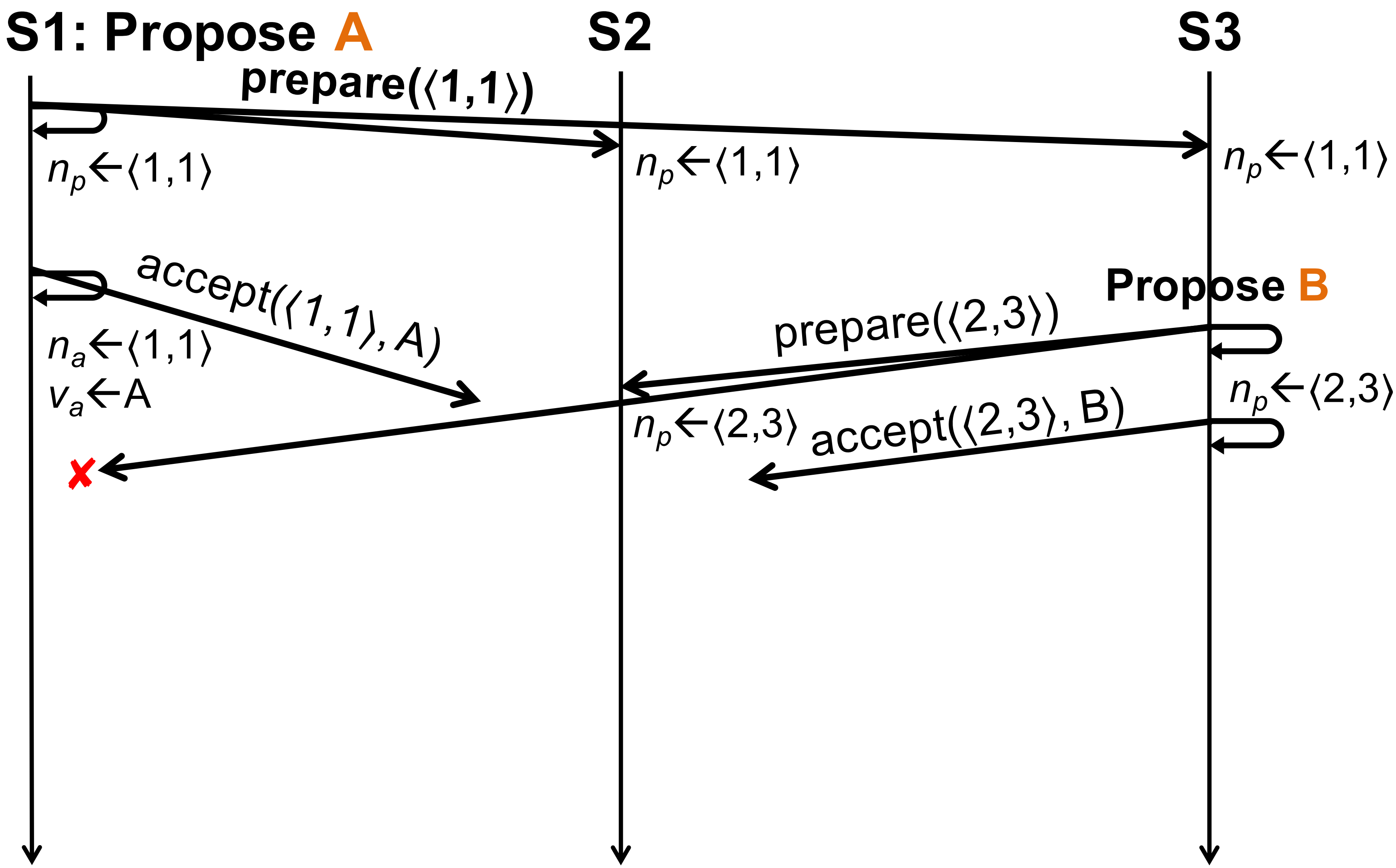
Example 2: Propagating agreement in the presence of failures



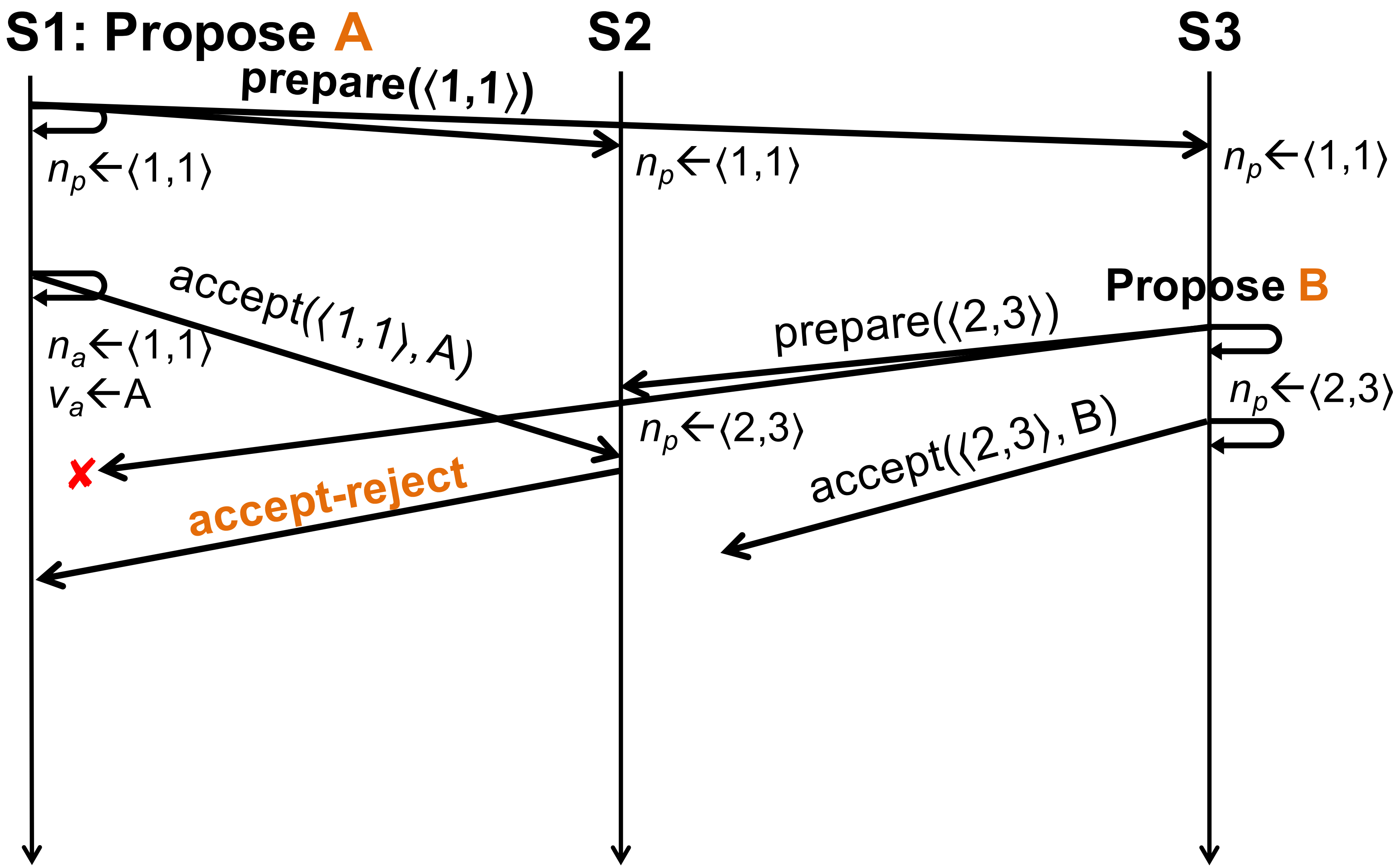
Once agreement reached, **majority know value**

Any **new** majority of prepares will **intersect** that value, send it in subsequent **accept** messages

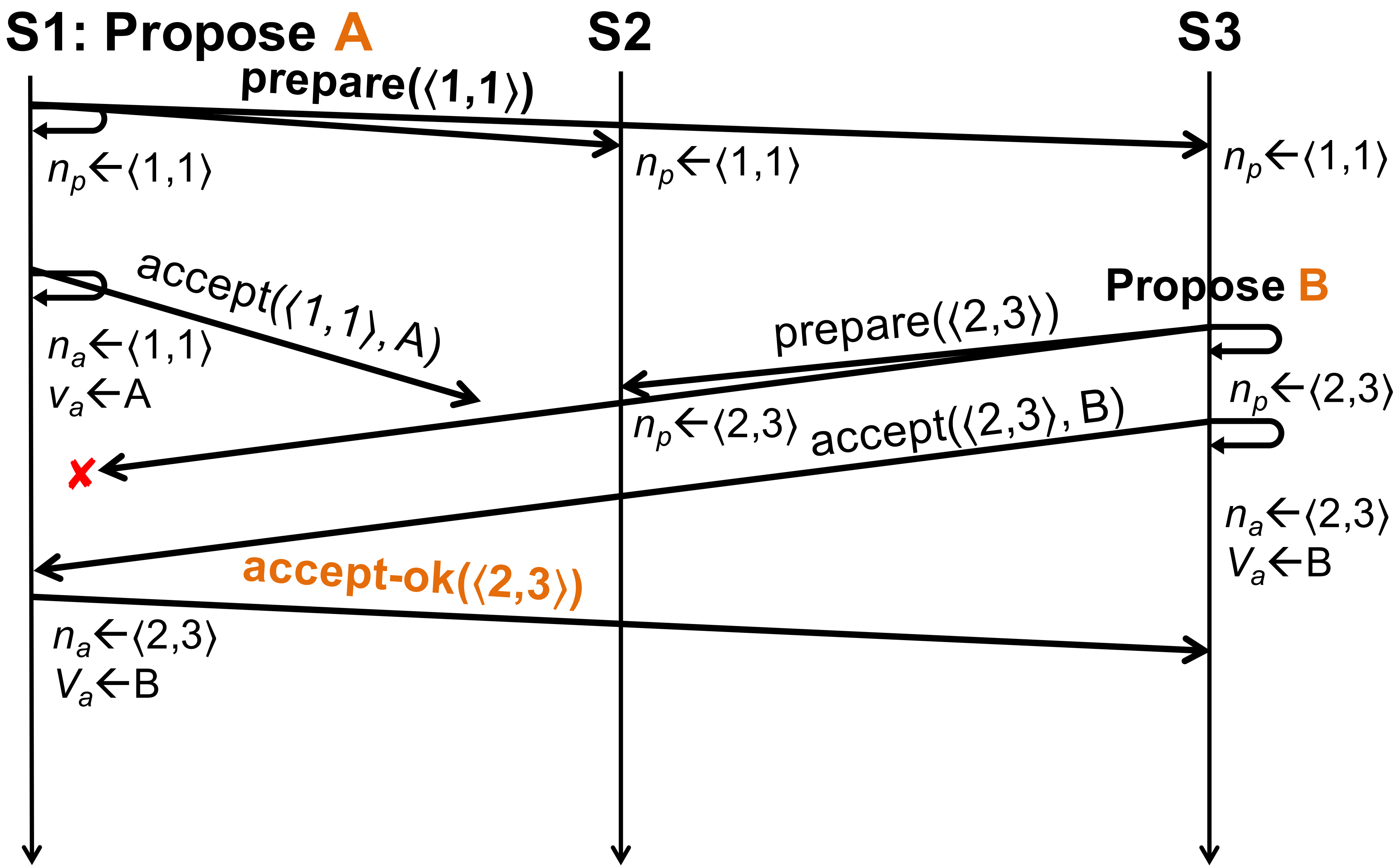
Example 3: Concurrent proposals



Example 3: Concurrent proposals
Case 1: S1's accept message wins



Example 3: Concurrent proposals
Case 2: S3's accept message wins



View-change algorithm

- Uses Paxos for state machine replication

- System operates in a series of ***views***
  - ***view*** = { View #, servers belonging to view }
    - Leader is lowest numbered server in that view

- To choose new leader, use Paxos to agree on new view
  - View $i + 1$ chosen by members of View i
  - To handle f failures, must have $2f + 1$ replicas
    - So that a majority is still alive
    - So that majority agree on the new view

Today

1. Two-Phase Commit
2. Replication and agreement with Paxos
3. **Time, events, and consistency**

Time and distributed systems

- With multiple events, what happens first?

A shoots B

A dies

B shoots A

B dies

Strict consistency

Definition: Any read on a data item X returns value corresponding to result of most recent write on X

- Strongest consistency model we'll consider
- Need an **absolute global time**
  - "Most recent" needs to be unambiguous
  - Impractical to implement

P1:	W(x)a		
P2:			R(x)a

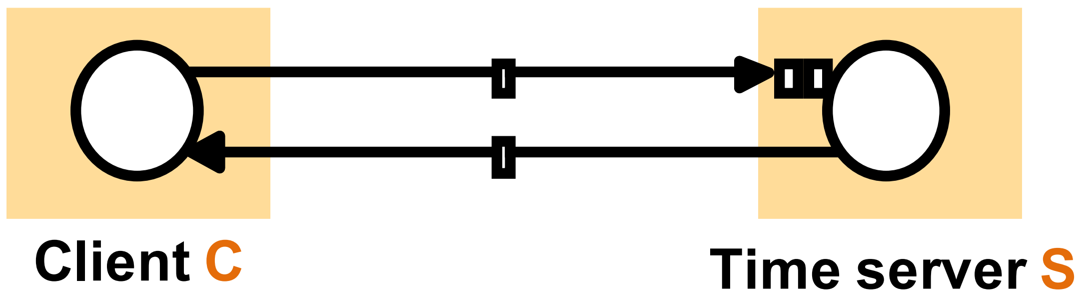
(a)

P1:	W(x)a		
P2:		R(x)NIL	R(x)a

(b)

Strictly Consistent **Not Strictly Consistent**

Cristian's clock synchronization algorithm



- Uses a **time server** to keep a **reference time**
- C asks S for time, adjusts local clock based on response
  - But different network latency → clock skew?
- Correct for this? For links with symmetrical latency:

RTT = response-received-time – request-sent-time

adjusted-local-time = server-timestamp t + (RTT / 2)

local-clock-error = adjusted-local-time – local-time

Is this sufficient?

- Server latency due to load?
  - If can measure:
    - adjusted-local-time = server-time t + (RTT+ lag / 2)
- But what about asymmetric latency?
  - RTT / 2 not sufficient!
- What do we need to measure RTT?
  - Requires no clock drift!
- What about "almost" concurrent events?
  - Clocks have just micro/millisecond precision

Sequential consistency (Lamport, 1979)

Definition: Result of any execution is same as if all ops were executed in some sequential order, and ops of each individual process appear in this sequence in order specified by its program

- When processes are running concurrently:
  - Interleaving of read and write ops is acceptable, but all processes see the same interleaving of ops

- Differences from strict consistency:
  - No reference to most recent time
    - **Absolute global time does not play a role**

Valid sequential consistency?

P1:	W(x)a				
P2:		W(x)b			
P3:			R(x)b		R(x)a
P4:				R(x)b	R(x)a

(a)

P1:	W(x)a				
P2:		W(x)b			
P3:			R(x)b		R(x)a
P4:				R(x)a	R(x)b

(b)

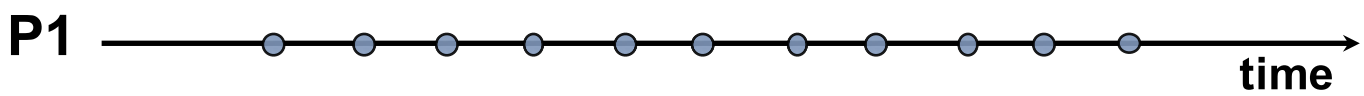
- Why? Because P3 and P4 don't agree on order of ops. Doesn't matter when events happened on different machines so long as processes **agree** on order
- *What if P1 instead of P2 did both W(x)a and W(x)b?*

Events and histories

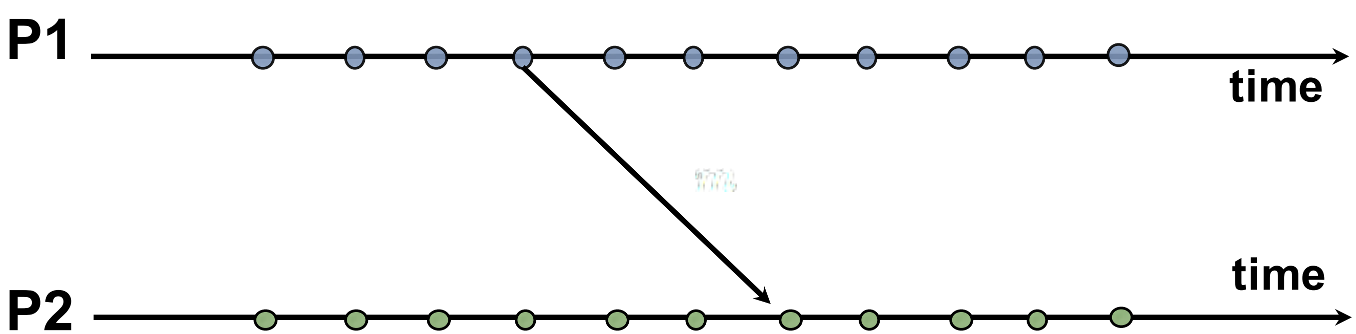
- Processes execute sequences of events
- Events can be of 3 types:
  - local, send, and receive
- e_p^i is the i-th event of process p
- The local history h_p of process p is the sequence of events executed by process

Ordering events

- Observation 1:
  – Events in a local history are totally ordered



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

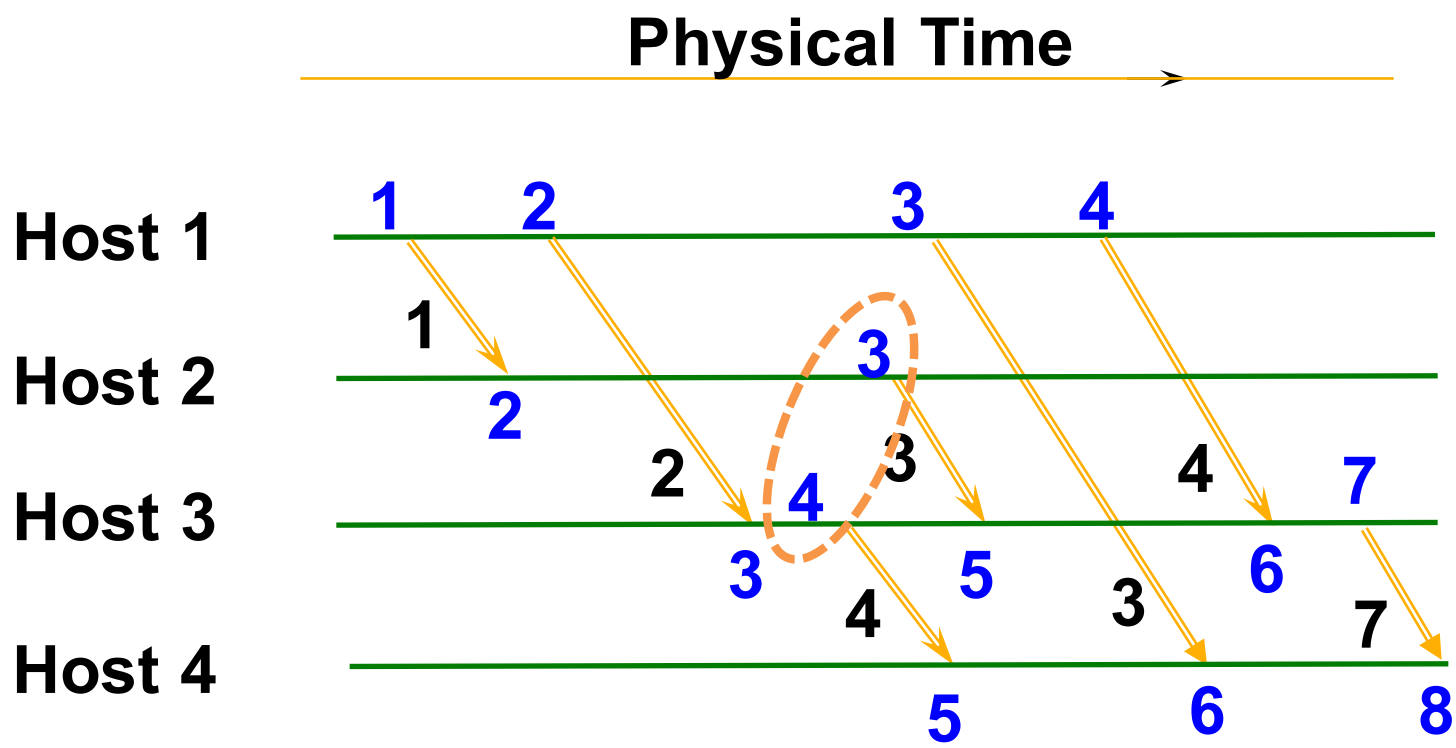


Happened-before relation → (Lamport, 1978)

Definition: In one process, if time(a) < time(b) then a ⟶ b.
If p1 sends m to p2 then send(m) ⟶ receive(m).
If a ⟶ b and b ⟶ c then a ⟶ c.

- Lamport Algorithm uses → for partial ordering
  - All processes have a local ***counter*** (initially 0)
  - Counter incremented by and assigned to each event as its timestamp
  - A message carries its sender's timestamp
    - On receive(msg) event, counter is updated to: max {receiver-counter, message-timestamp } + 1

Lamport logical time



Logically concurrent events!

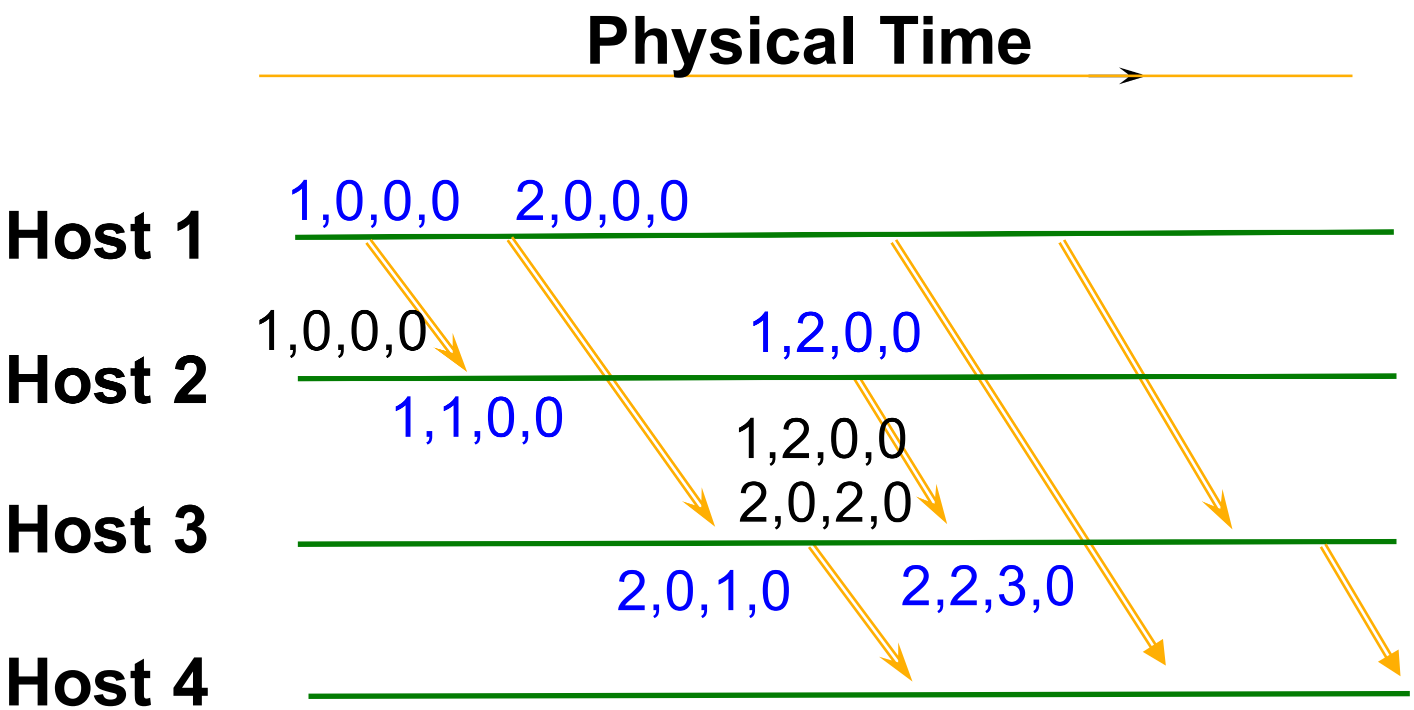
Vector logical clocks

- With Lamport logical time:
  - e precedes f $\Rightarrow$ timestamp(e) < timestamp (f), but
  - timestamp(e) < timestamp (f) **doesn't imply** e precedes f

- Vector logical time guarantees this:
  - Each server ***s*** has a **vector v_s[ ]** of counters
    - $v_s[k]$ is the clock value at server s, for server k (initially 0)
  - On event: $v_k[k] \leftarrow v_k[k] + 1$, assign v_k to the event

  - A send(msg) event carries vector timestamp
  - For receive(msg) event at server ***s***, $v_s[k] \leftarrow$
    - max{ $v_s[k]$, msg[k] }, **if** k ≠ s
    - $v_s[k]$+ 1 **if** k = s

Vector logical time



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
  - (a and b conflict)

- If one history is prefix of other, then one vector timestamp < other
- If one history is not a prefix of other, then vector timestamps won't be comparable.

Weakening consistency: “Eventual” Consistency

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

- Useful where concurrency appears only in a restricted form
- Assumption: write conflicts, if any, will be easy to resolve
  - Even easier if whole-”object” updates only

Eventual consistency in practice

- Distributed, inconsistent state
  - Writes only go to some subset of storage nodes
    - By design (for higher throughput)
    - Due to transmission failures (bimodal multicast)

- Anti-entropy protocol fixes inconsistencies
  - If automatic reconciliation, requires some way to handle write conflicts
  - CMU's **Coda** file system for disconnected operation
  - PARC's **Bayou** project for distributed state
  - Amazon's use in **Dynamo** system

Middle ground: Causal consistency

- Potentially ***causally related*** operations:
  - R(x) then W(x)
  - R(x) then W(y), $x \neq y$

- Necessary condition: Writes that are ***potentially*** causally related must be seen by all processes in the same order
  - Concurrent writes may be seen in a different order on different machines

- Hutto and Ahamad, 1990 → Eiger, NSDI '13
- Weaker than sequential consistency, stronger than eventual consistency

Causal consistency

P1:	W(x)a			W(x)c		
P2:		R(x)a	W(x)b			
P3:		R(x)a			R(x)c	R(x)b
P4:		R(x)a			R(x)b	R(x)c

- Allowed with causal consistency, but not with sequential consistency
- W(*x*)*b* and W(*x*)*c* are **concurrent**
  - So all processes don't see them in the same order
- P3 and P4 read the values 'a' and 'b' in order as potentially causally related. No 'causality' for 'c'.

Causal consistency

P1:	W(x)a			W(x)c	
P2:		R(x)a	W(x)b		
P3:		R(x)a		R(x)c	R(x)b
P4:		R(x)a		R(x)b	R(x)c

- Why not sequentially consistent?
  - P3 and P4 see W(*x*)*b* and W(*x*)*c* in different order.

- But fine for causal consistency
  - Writes W(*x*)*b* and W(*x*)*c* are **not causally dependent**
    - Write after write has no dependencies

Causal consistency

P1: W(x)a				
P2:	R(x)a	W(x)b		
P3:			R(x)b	R(x)a
P4:			R(x)a	R(x)b

(a)

P1: W(x)a				
P2:		W(x)b		
P3:			R(x)b	R(x)a
P4:			R(x)a	R(x)b

(b)

- A: Violation: W(x)b potentially dependent on W(x)a
- B: Correct. P2 doesn't read value of a before W

Causal consistency

- Requires keeping track of which processes have seen which writes
  - Needs a dependency graph of which op is dependent on which other ops
  - …or use vector timestamps!

CAP

- Conjectured by Eric Brewer (2000), proven by Nancy Lynch and Seth Gilbert (2002)
- CAP: Distributed systems can have 2 of 3:
  - Consistency (strong)
  - Availability
  - Partition Tolerance: Liveness despite arbitrary failures
- Really: You get P, you choose A or C
- Eric was using CAP to justify BASE: Basically Available, Soft-state services with Eventual consistency