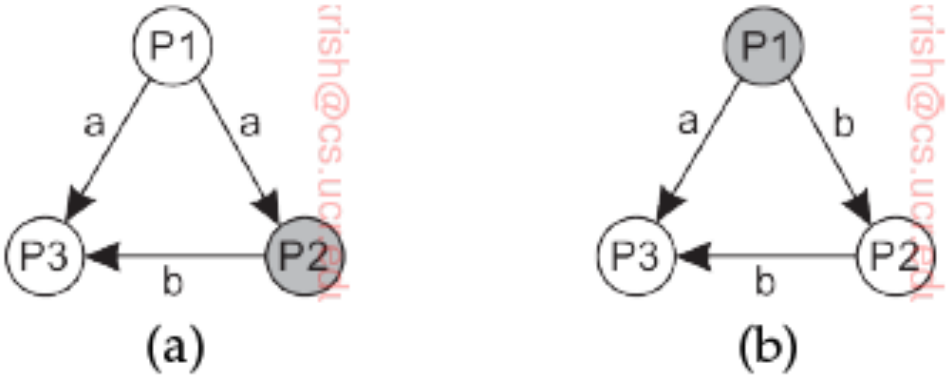
LECTURE 9

Byzantine Failures and CAP Theorem

Byzantine Failures

- Arbitrary patterns of failures – not just crash failures.
- Specifically → inconsistent messages.
  - Primary exhibiting Byzantine behavior → sending different messages to different replicas.
  - Backup exhibiting Byzantine behavior → sending message inconsistent from the input from primary.



Byzantine agreement

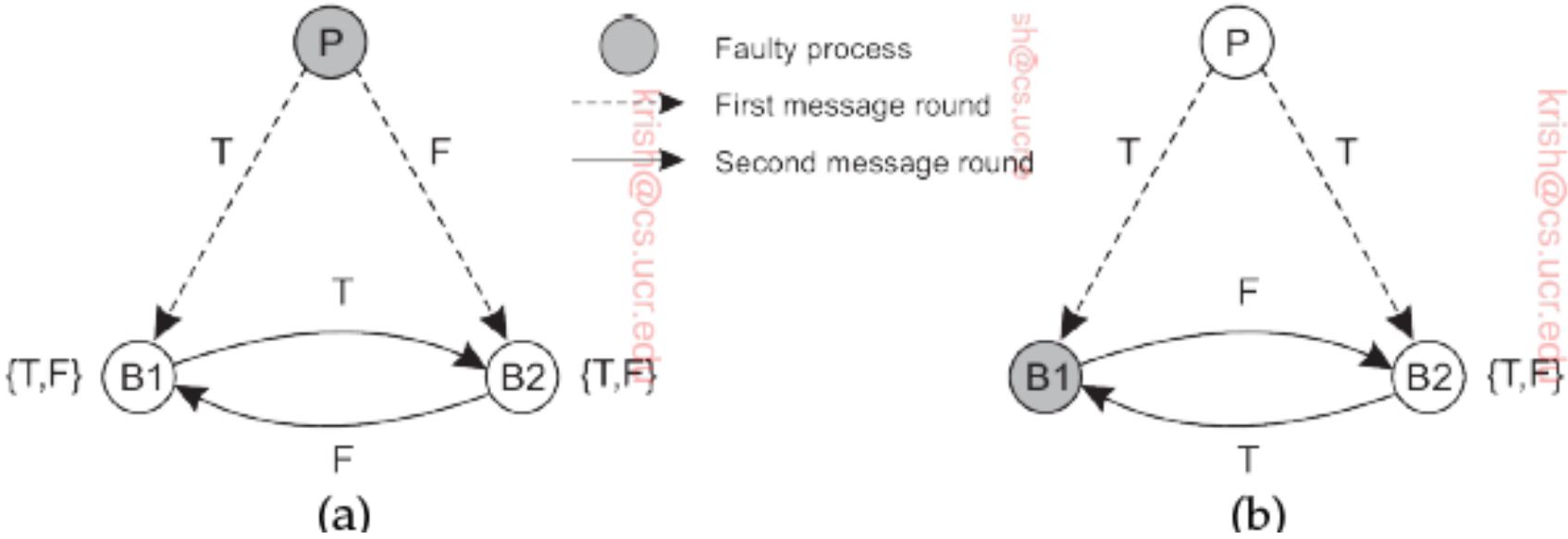
- BA1: Every non-faulty backup process stores the same value.
- BA2: If primary is non-faulty, then every non-faulty backup process stores exactly what the primary had sent.
  - If primary is non-faulty BA1 → BA2

What is needed ?

- Under these assumptions, at least 3k + 1 members are needed to reach consensus if k members can fail.
  - Our assumptions are there is a primary P and backups B_1 B_2 …. B_{n-1}

Having 3k processes is not enough

- Let us assume k = 1
- Consider the following:
  - Case (a) primary fails; Case (b) back up fails



Having 3k cases is not enough

- In Case (a) primary sends T to one backup and F to one backup.
  - Each backup forwards what is received to the other.
  - Thus, each has {T,F} –conclusion cannot be drawn.
- In Case (b), B1 flips the primary's message and relays a wrong (F) message to B2. B2 relays the correct message to B1. Again easy to see – no conclusion.

Extension to general case

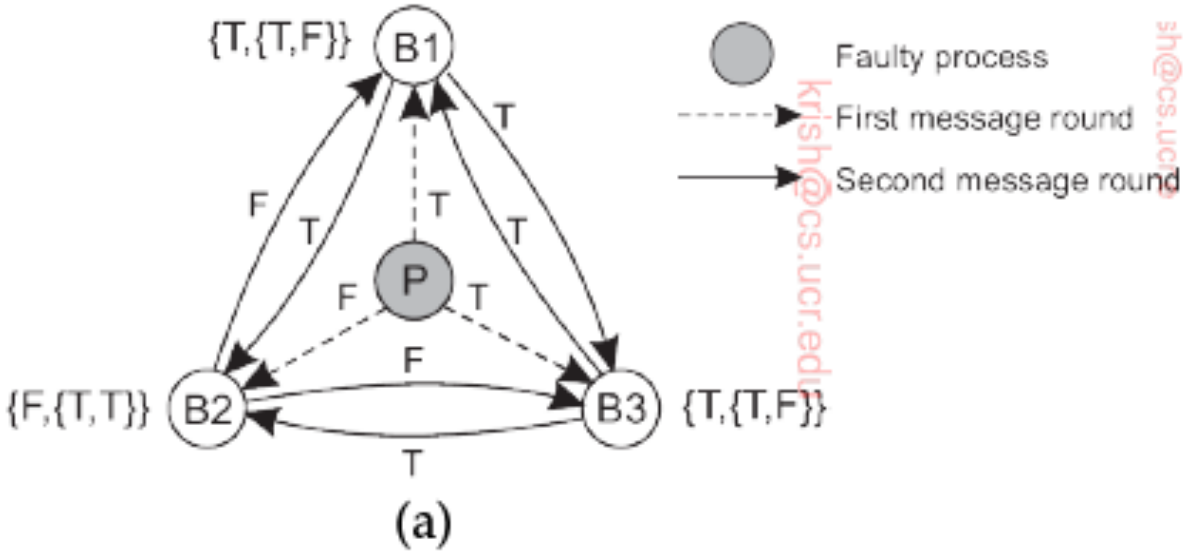
- For k > 1, use a simple reduction method.
- Group the processes into three disjoint sets each containing at most n/3 members.
- Simulate actions – each set S_i represents all members of its group.
  - Thus, all members of a group are faulty or not faulty.
  - Easy to see that this is similar to the example with k=1.

Having 3k+1 processes is enough

- We will only show this for the case k = 1.
- A similar but more cumbersome analysis possible for k = 2.
- We will consider two cases :
  - Primary is faulty
  - Backup is faulty

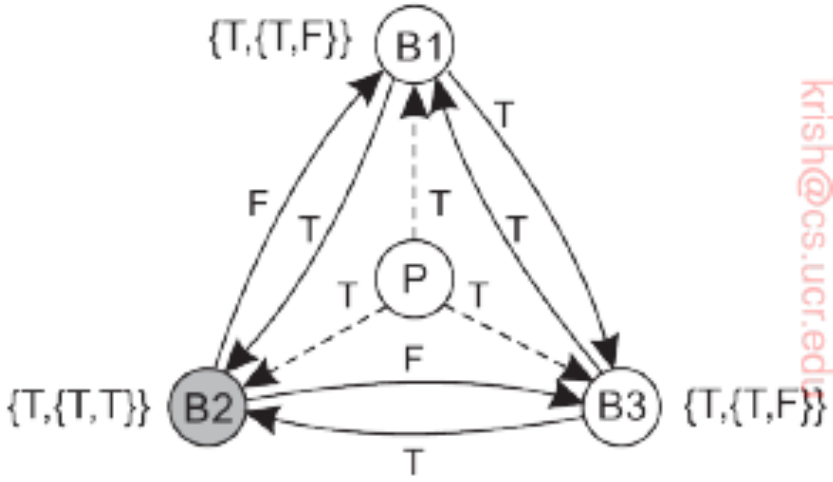
Consensus with 4 processes: Faulty primary

- Processes forward what they receive to others.
- In first round, P sends T to B1 and F to B2 and T to B3.
- Each backup sends what they have to others.
- Easy to see that at the end of the second round, each backup has {T, T, F}.
  - Thus, a consensus of T is reached.



Consensus with 4 processes: Faulty backup

- Non faulty primary sends T to all backups.
- Faulty B2 sends F, while non-faulty B1 and B3 send T.
- We see that the two non-faulty backups have {T,T,F}.

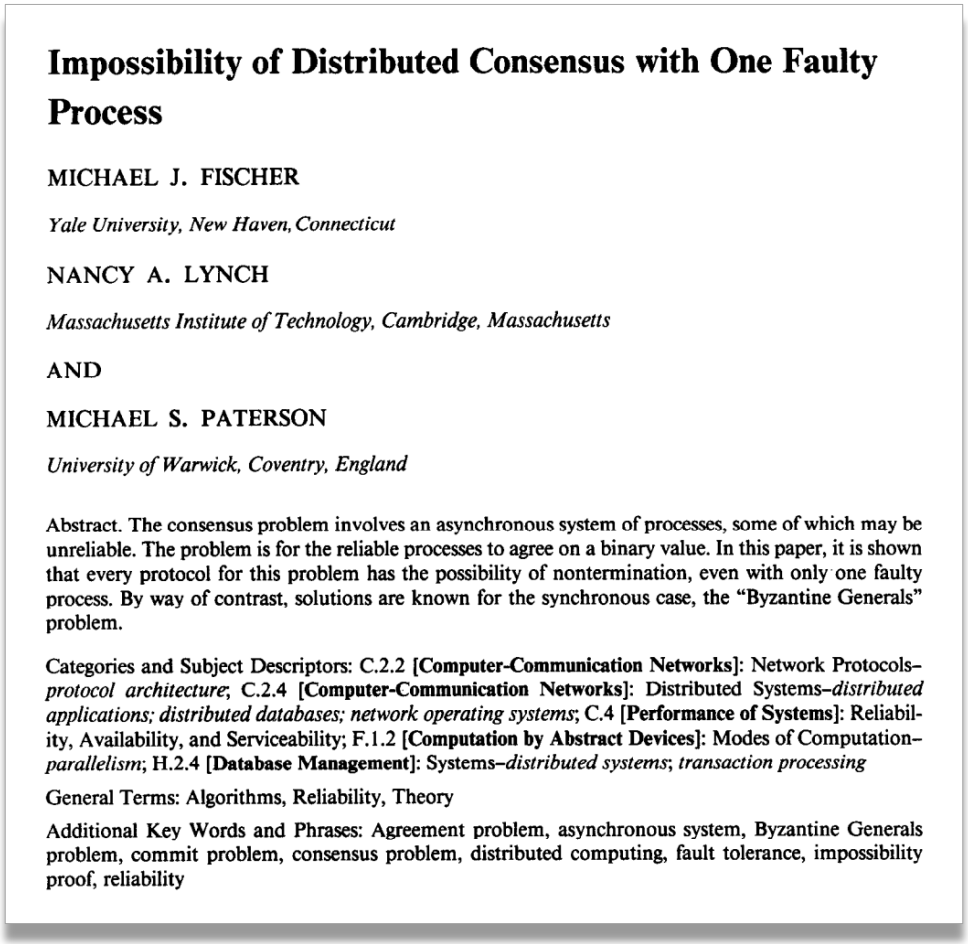


Asssumptions

- For crash failures (2k+1) processes to come to consensus in presence of k failures.
- However, the assumption is that the delay is bounded – message is received within some finite time.
  - But what is this finite time ?
- If processes do not operate in a lock step mode i.e., they are asynchronous, then hard to say what latency is incurred in receiving messages.

FLP Impossibility Result

- In asynchronous model, distributed consensus impossible if even one process *may* fail
- Holds even for "weak" consensus (i.e., only some process needs to learn, not all)
- Holds even for only two states: 0 and 1



Impossibility of Distributed Consensus with One Faulty Process

MICHAEL J. FISCHER

Yale University, New Haven, Connecticut

NANCY A. LYNCH

Massachusetts Institute of Technology, Cambridge, Massachusetts

AND

MICHAEL S. PATERSON

University of Warwick, Coventry, England

Abstract. The consensus problem involves an asynchronous system of processes, some of which may be unreliable. The problem is for the reliable processes to agree on a binary value. In this paper, it is shown that every protocol for this problem has the possibility of nontermination, even with only one faulty process. By way of contrast, solutions are known for the synchronous case, the "Byzantine Generals" problem.

Categories and Subject Descriptors: C.2.2 [**Computer-Communication Networks**]: Network Protocols–*protocol architecture*; C.2.4 [**Computer-Communication Networks**]: Distributed Systems–*distributed applications; distributed databases; network operating systems*; C.4 [**Performance of Systems**]: Reliability, Availability, and Serviceability; F.1.2 [**Computation by Abstract Devices**]: Modes of Computation–*parallelism*; H.2.4 [**Database Management**]: Systems–*distributed systems*; *transaction processing*

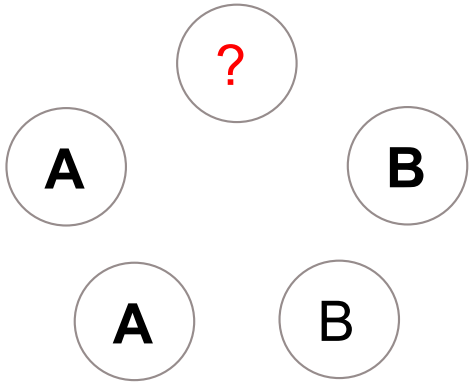
General Terms: Algorithms, Reliability, Theory

Additional Key Words and Phrases: Agreement problem, asynchronous system, Byzantine Generals problem, commit problem, consensus problem, distributed computing, fault tolerance, impossibility proof, reliability

FLP Impossibility Result

- Intuition: Cannot distinguish failures from slowness
  - May not hear from process that has deciding vote

- Implication:
  - Choose safety or liveness
    - Liveness → availability



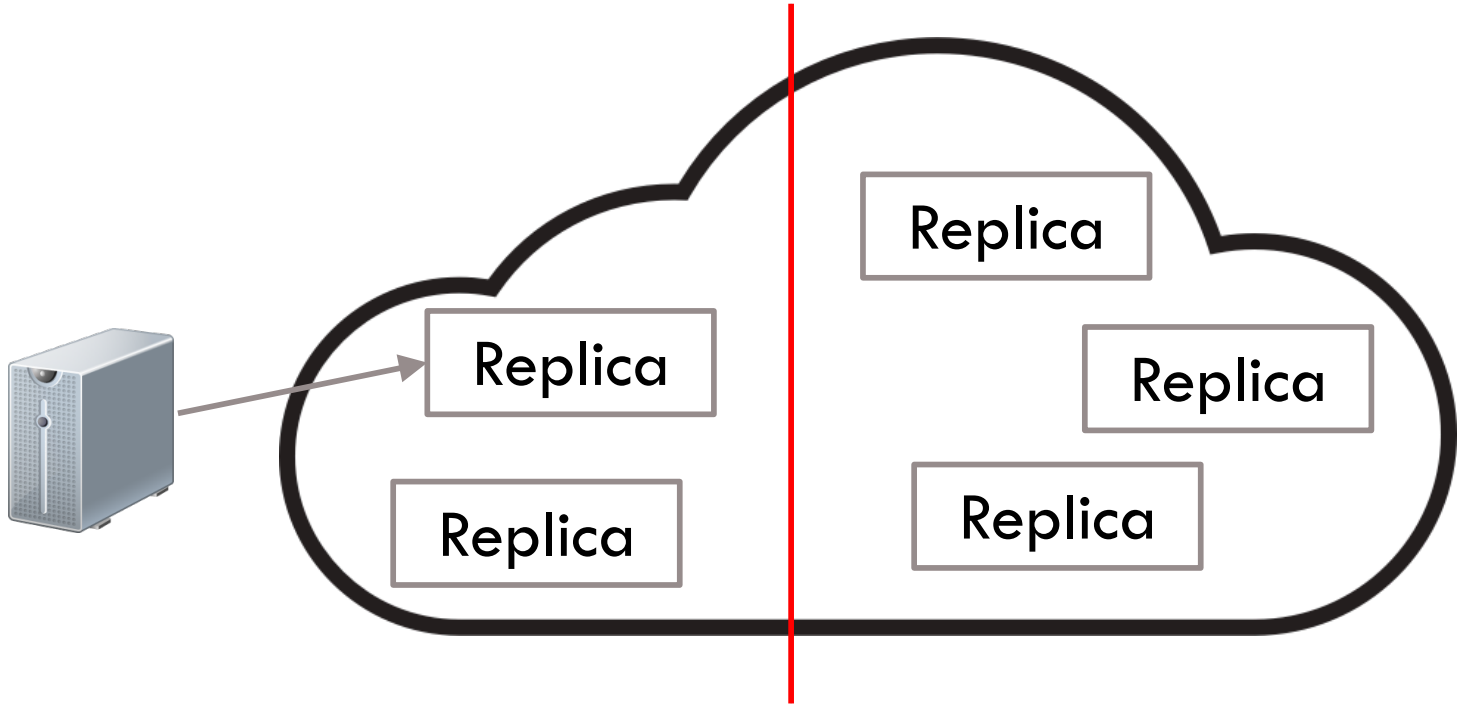
- How to get both safety and liveness?
  - Need failure detectors (partial synchrony)

Recap: some terms

- Consistency: Every read receives the most recent write or an error
- Availability : Every read receives a response (not error) but no guarantee it is most recent.
- Partition tolerance : System operates in spite of arbitrary number of message losses or delays between the replicas.

CAP Theorem

- Pick any two: Consistency (C), Availability (A), and Partition-Tolerance (P)
  - In practice, choose between CP and AP

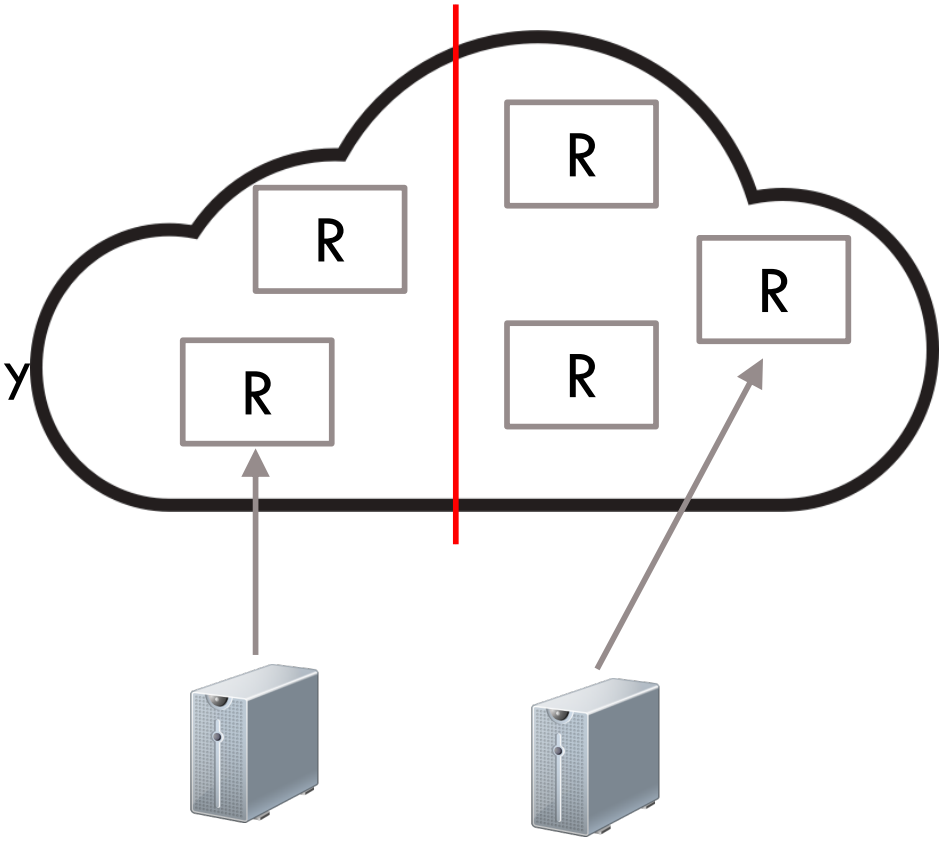


What does this mean?

- When partition occurs:
- Cancel the operation
  - Decreases availability but ensures consistency
- Proceed with the operation
  - Ensures availability at the cost of inconsistency

PACELC

- Extension of CAP to include the impact of latency.
- If partition,
  - Choose availability vs. consistency
- Else,
  - Choose latency vs. consistency
- Unifies two separate tradeoffs that we have talked about



Why should you care?

- Can identify when system designers over-claim
- Explicitly reason about tradeoffs when designing systems
- Example: Should you choose AP or CP?

Impact on Consistency

- When a replica receives a read or write, when can it respond without violating linearizability?
  - If it is in a majority partition
- If we want any replica to always serve clients
  - Can we provide any consistency guarantees?
- Example of such a RSM?

Example Scenario

- Calendar application running on smartphones
  - Each entry has time and set of participants
- Local copy of calendar on every phone
  - No master copy
- Phone has only intermittent connectivity
  - Cellular data expensive while roaming
  - WiFi not available everywhere
  - Bluetooth has very short range

Format of Updates

- Goal: Automatic conflict resolution when replicas sync with each other

- What would work?
  - "10AM meeting, 4901 BBB, EECS 498 staff"
  - "1 hour meeting at 10AM if room and participants free, else 11AM, 4901 BBB, EECS 498 staff"

Example Execution

- Node **A** asks for meeting **M1** at 10 AM, else 11 AM
- Node **B** asks for meeting **M2** at 10 AM, else 11 AM

- **X** syncs with **A**, then **B**
- **Y** syncs with **B**, then **A**

- **X** will put meeting **M1** at **10:00**
- **Y** will put meeting **M1** at **11:00**

Replicas can't apply updates in order received

Ordering of Updates

- All replicas must apply updates in same order
- How to achieve consistent ordering despite intermittent connectivity?
- Lamport clock!

Ordering of Updates

- Recap of Lamport clocks:
  - Every update associated with timestamp of the form (local timestamp **T**, originating node **ID**)
  - a < b if a.T < b.T, or (a.T = b.T and a.ID < b.ID)

- Updates with timestamps in our example:
  - ⟨701, A⟩: A asks for meeting **M1** at 10 AM, else 11 AM
  - ⟨770, B⟩: B asks for meeting **M2** at 10 AM, else 11 AM

Another Example Execution

- ⟨701, A⟩: **A** asks for meeting **M1** at 10 AM, else 11 AM
- ⟨700, B⟩: **Delete** meeting **M1**
  - B's clock was slow

- Now, delete will be ordered before add
- How to prevent this?
  - Lamport clocks preserve causality

Example Execution

- ⟨701, A⟩: A asks for meeting **M1** at 10 AM, else 11 AM
- ⟨770, B⟩: B asks for meeting **M2** at 10 AM, else 11 AM

- Pre-sync database state:
  - A has M1 at 10 AM
  - B has M2 at 10 AM

- After A receives and applies update from B:
  - A has M1 at 10AM and M2 at 11AM

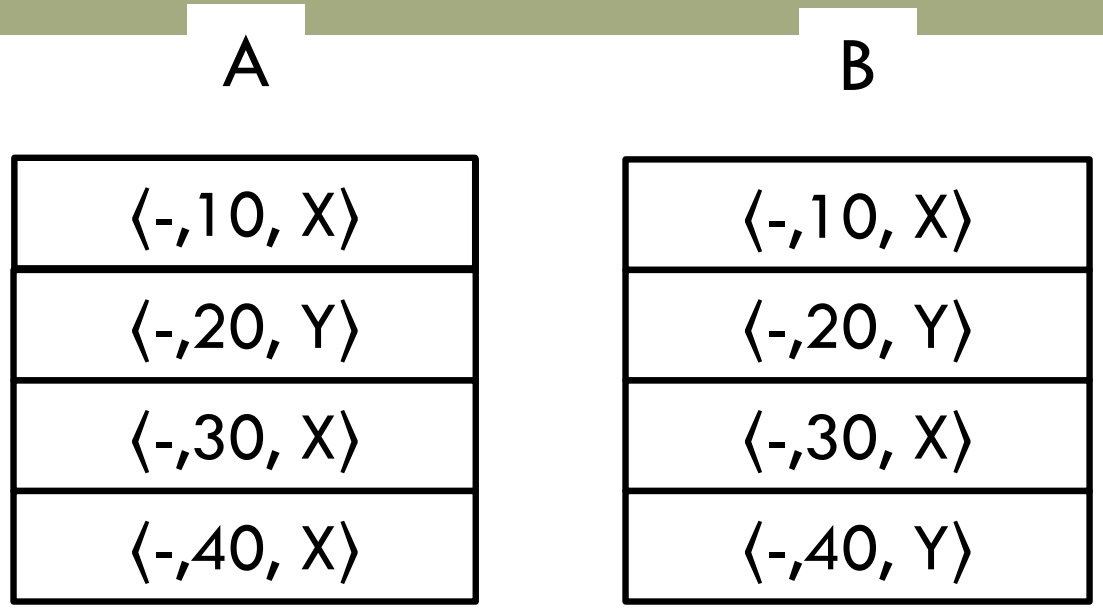
- How can B apply update from A?
  - B already has M2 at 10AM

Solution: Roll back and replay

- B needs to "roll back" its state, and re-run ops in the correct order
- So, in the user interface, displayed calendar entries are tentative at first
  - B's user saw M2 at 10 AM, then it moved to 11 AM
- Takeaways:
  - Need to maintain log of updates
  - Sync updates between replicas not state

How to sync, quickly?

How to sync without state exchange proportional to size of log?



- B tells A: highest timestamp for every node ← Version vector
  - e.g., "X 30, Y 40"
  - In response, A sends all X's updates after ⟨-,30,X⟩, and all Y's updates after ⟨-,40,Y⟩

Consistency semantics

- Can a calendar entry ever be considered no longer tentative?

- Eventual consistency:
  - If no new updates, all replicas eventually converge to same state

Committing Updates

- Implications of ordering updates using timestamp:
  - Never know whether some write from the past may yet reach your node
  - So all entries in log must be tentative forever
  - All nodes must store entire log forever

- How to mark calendar entries as committed?
- How to garbage collect updates to prune the log?

Committing Updates

- Update with timestamp (T, ID) is stable if higher timestamp update received from every node
- Problem?
  - Disconnected replica prevents others from declaring updates stable
- Solution:
  - Pick one of the replicas as primary
  - Primary determines order of updates
  - Desirable properties of primary?

Committing Updates

- At any replica:
  - Stable state
  - Log of tentatively ordered updates (order based on Lamport clock timestamps)
- Upon sync with primary
  - Receive updates in order
  - Apply updates to stable state and prune log
- Any constraints in order chosen by primary?
  - Must respect causality