Handling Failures in MapReduce

- If Master fails, can we start up a new Master and resume execution of job?
  - No!
    - Lost all state about execution status of tasks

- Must replicate Master state to tolerate failures
Replicating MapReduce Master

Master M1
Map tasks:
1 2 3 4

Worker W1

Task 3 assigned to W1

Master M2
Map tasks:
1 2 3 4

Worker W2

Done with task 1

Run task 3

Done with task 2

Run task 3

Task 3 assigned to W1

Run task 3
Replicating Bank Database

- One copy of account in SF, another copy in NY
- Clients send **updates to both** copies

---

"Deposit $100"

$1,000
$1,100
$1,111

"Pay 1% interest"

$1,000
$1,010
$1,110

*Inconsistent replicas!*
Synchronizing Replicas

- **How to ensure replicas are in sync?**
  - Apply updates in same order at all replicas

- **Model every replica as a state machine**
  - Given identical initial states, applying updates in same sequence results in same final state
  
→ Replicated state machine (RSM)
Replicated State Machine

- **Replicate over time**
  - Start a new replica after one goes down
  - Crash failures

- **Replicate over space**
  - Run multiple replicas simultaneously
  - Fail-stop failures (unable to handle exceptions)
Implementing RSM

- Order updates based on time of receipt

- Challenge: Clocks not in sync across replicas
Clock Syncing: Challenges

- **Leverage GPS broadcasts?**
  - Time from GPS accurate to about 1 microsecond
  - Power hungry and does not work indoors

- **Correct for skew based on reference clock?**
  - Clock drift
  - Unbounded network delay
Cristian’s algorithm

1. Client sends a request packet, timestamped with its local clock $T_1$

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

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

4. How can client use these timestamps to synchronize its local clock to server’s clock?
Cristian’s algorithm

Goal: At $T_4$, client sets clock $\leftarrow T_3 + \delta_{\text{resp}}$

- But client knows $\delta$, not $\delta_{\text{resp}}$

- Client measures round trip time $\delta = \delta_{\text{req}} + \delta_{\text{resp}} = (T_4 - T_1) - (T_3 - T_2)$

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

Client sets clock $\leftarrow T_3 + \frac{1}{2}\delta$
Further refinements

- Limitation of Cristian algorithm
  - Relies on timeserver availability

- NTP
  - Distributed ecosystem for time synchronization
  - Tolerates failures of network and time servers
  - Leverages heterogeneous clocks
- All replicas must apply updates in same order

- Ordering updates based on time of receipt is hard because of clock skew and drift

- Challenge in synchronizing clocks:
  - Unbounded network delay
Idea: Logical clocks

- Landmark 1978 paper by Leslie Lamport

- **Insight**: Disregard precise clock time
  - Only relationships between events matter

- Associate every event with “logical time”
  - Preserve “happens before” relationships
  - If a happens before b, then clock(a) < clock(b)
Defining “happens-before”

1. In same process, if $a$ occurs before $b$, then $a \rightarrow b$
2. If $c$ is a message receipt of $b$, then $b \rightarrow c$
3. If $a \rightarrow b$ and $b \rightarrow c$, then $a \rightarrow c$
How do we use “happens before” to ensure consistent ordering of updates in an RSM?
Lamport clock: Objective

- Associate any event \( a \) with a *clock time* \( C(a) \)

- Clock condition: If \( a \rightarrow b \), then \( C(a) < C(b) \)

- Order events at all nodes based on clock time
Each process $P_i$ maintains a local clock $C_i$.
The Lamport Clock algorithm

1. Before executing an event, \( C_i \leftarrow C_i + 1 \)
The Lamport Clock algorithm

1. Before executing an event, $C_i \leftarrow C_i + 1$
1. Before executing an event, \( C_i \leftarrow C_i + 1 \)

2. Send the local clock in the message \( m \)

\[
\begin{align*}
P1 & : C_1 = 2 \\
P2 & : C_2 = 0 \\
P3 & : C_3 = 0
\end{align*}
\]

Physical time ↓
3. On process $P_i$ receiving a message $m$:

- What time to set for receive event? $C(m) + 1$?
3. On process $P_i$ receiving a message $m$:

What time to set for receive event? $C(m) + 1$?
The Lamport Clock algorithm

3. On process $P_i$ receiving a message $m$:

- Set $C_i$ and time of receive event to $1 + \max\{ C_i, C(m) \}$
Lamport clock

- Associate any event $a$ with a *clock time* $C(a)$

- Clock condition: If $a \rightarrow b$, then $C(a) < C(b)$

- If $a$ causally leads to $b$, then $C(a) < C(b)$
Ordering all events

C = 1  2  3  4

a → b → c

P1
C_1 = 2

P2
C_2 = 4

d → e → f

P3
C_3 = 2
Not all events are related by “happens before”

- a, d not related by \( \rightarrow \) so concurrent, written as \( a \mid \mid d \)
Ordering all events

- How to ensure total ordering at all replicas?

- Can two events at same process have same clock value?
Ordering all events

- Break ties by appending the process number to each event:

  1. Process $P_i$ timestamps event $e$ with $C_i(e).i$
  2. $C(a).i < C(b).j$ when:
     - $C(a) < C(b)$, or $C(a) = C(b)$ and $i < j$

- Now, for any two events $a$ and $b$, $C(a) < C(b)$ or $C(b) < C(a)$
  - Total ordering of events
Recall multi-site database replication:

- San Francisco (P1) deposited $100:
- New York (P2) paid 1% interest:

Applying updates in different order made replicas inconsistent

All replicas apply updates in Lamport clock order
RSMs with Lamport Clocks

3.1
8.2
2.1
7.2

P1

P2

9.1
9.2
10.1
10.2
11.2
Consistent Ordering of Updates

- **Key idea:** Place events into a local queue
  - Sorted by increasing $C(x)$

- **How to tell if update at head of queue can be applied?**
  - Ping other nodes and check if they have earlier updates
RSMs with Lamport Clocks
Reducing Waiting

- How to reduce time between receiving an update and applying it?

- Periodically ping all other nodes to sync clock

- Tolerate temporary inconsistency
  - Apply update immediately upon receipt
  - But, maintain log of updates in order to rollback
Causal ordering

- Ordering all updates at all replicas may be unnecessary
  - Example: Replication of Facebook posts

- Causal ordering suffices
  - Show comments only if post visible to user

- With Lamport clock, \( a \rightarrow b \) implies \( C(a) < C(b) \)
  - But, converse is not necessarily true
  - \( C(a) < C(b) \) does not imply \( a \rightarrow b \) (possibly, \( a \parallel b \))
Vector clock (VC)

- Label each event $e$ with a vector 
  \[ V(e) = [c_1, c_2 \ldots, c_n] \]
  \[ \text{No. of components} = \text{No. of processes} \]

- Semantic interpretation:
  \[ c_i \text{ is a count of events in process } i \text{ that causally precede } e \]
Updating vector clocks

- Initially, all vectors are \([0, 0, \ldots, 0]\)
- Two rules for updating

1. For each **local event** on process \(i\), increment local entry \(c_i\)
2. If process \(j\) **receives** message with vector \([d_1, d_2, \ldots, d_n]\):
   - Set each local entry \(c_k = \max\{c_k, d_k\}\)
   - Increment local entry \(c_i\)
Vector clock: Example

- All counters start at [0, 0, 0]
- Applying local update rule
- Applying message rule
  - Local vector clock \textbf{piggybacks} on interprocess messages
Vector clocks can establish causality

- **Rule for comparing vector clocks:**
  - \( V(a) = V(b) \) when \( a_k = b_k \) for all \( k \)
  - \( V(a) < V(b) \) when \( a_k \leq b_k \) for all \( k \) and \( V(a) \neq V(b) \)

- **How to tell if two events are concurrent?**
  - \( a \parallel b \) if \( a_i < b_i \) and \( a_j > b_j \), some \( i, j \)

- \( V(a) < V(z) \) if and only if there is a chain of events linked by \( \rightarrow \) between \( a \) and \( z \)