DISTRIBUTED SYSTEMS: ORDERING AND CONSISTENT CUTS

by Maofan (Ted) Yin
my428@cornell.edu
Time, Clocks, and the Ordering of Events in a Distributed System

- The original author of LaTeX
- Sequential consistency
- Atomic register hierarchy
- Lamport’s bakery algorithm
- Byzantine fault tolerance
- Paxos
- Lamport signature

Leslie B. Lamport (1941–)
Time, Clocks and the Ordering of Events

Leslie B. Lamport (1941–)

- B.S. in mathematics from MIT
- M.A. and Ph.D. in mathematics from Brandeis University
- Dijkstra Prize (2000, because of this paper, and 2005)
- ACM A.M. Turing Award (2013)
- ACM Fellow (2014)
“Jim Gray once told me that he had heard two different opinions of this paper: that it’s trivial and that it’s brilliant. I can’t argue with the former, and I am disinclined to argue with the latter.”

Leslie B. Lamport (1941–)
“This is my most often cited paper. Many computer scientists claim to have read it. But I have rarely encountered anyone who was aware that the paper said anything about state machines … People have insisted that there is nothing about state machines in the paper. I’ve even had to go back and reread it to convince myself that I really did remember what I had written.”
“The only reason of time is so that everything does not happen at once.”

— Albert Einstein

- Why time is so important? Air ticket reservation, online shopping, etc.
“The only reason of time is so that everything does not happen at once.”

— Albert Einstein

- Systems: an interesting definition of “distributed”: msg. transmission delay is NOT negligible compared to the time between events in a single process.

- Sometimes impossible to say any one of two occurred first: partial ordering.
“The only reason of time is so that everything does not happen at once.”

— Albert Einstein

- “Everything does not happen at once” means ordering.
- An ordering can give a happened-before relation of events in the system.
- Clocks can map events to numbers, so as to give the relation.
Clocks
In this paper, two clock implementations are introduced

- **Logical clocks:**
  - works without the help of any physical equipment,
  - causes anomaly with external happened-before relation (the clock is confined within the system).

- **Physical clocks:**
  - works when physical clocks have certain precision,
  - but provides with strong relation.
We have

- A priori: total ordering of events in the same process
- Msgs. can carry time info

We want to achieve

- A relation $a \rightarrow b$ that
  1. $a, b \in$ same process, $a$ comes before $b \implies a \rightarrow b$,
  2. $a$ sends a msg. to $b \implies a \rightarrow b$,
  3. $a \rightarrow b \land b \rightarrow c \implies a \rightarrow c$.

Remarks:

- $a$ and $b$ are concurrent if $a \not\rightarrow b \land b \not\rightarrow a$.
- $a \not\rightarrow a$ (irreflexivity),
- $a \rightarrow b \land b \rightarrow c \implies a \rightarrow c$ (transitivity),
- $a \rightarrow b \implies b \not\rightarrow a$ (asymmetry).
Logical Clocks: Space-Time Diagram

Past 1965
sending and receiving msgs. are also events,

happened-before relation can be deduced by checking whether there is a directed path from a to b.
Let the clock be $C\langle e \rangle$, where $e$ stands for an event.

$C\langle e \rangle := C_i\langle e \rangle$, $e$ is an event of process $i$.

To satisfy "→" relation, we want $\forall a, b$

$$a \rightarrow b \implies C\langle a \rangle < C\langle b \rangle \quad \text{(clock cond.)}$$

not vice versa: $a \rightarrow b \iff C\langle a \rangle < C\langle b \rangle$

otherwise,

$$e \leftrightarrow e' \land e' \leftrightarrow e \implies C\langle e \rangle \not< C\langle e' \rangle \land C\langle e \rangle \not> C\langle e' \rangle$$

$$\implies C\langle e \rangle = C\langle e' \rangle$$
Logical Clocks: Design

- Clock condition is held if
  - C1: \(a, b \in \text{proc. } i: a \text{ is before } b \implies C_i(a) < C_i(b)\).
  - C2: \(i\) sends msg. as event \(a\) to \(j\) as event \(b\): \(C_i(a) < C_j(b)\).

- Therefore, we can impose the following implementation rules
  - IR1: proc. \(i\) increases \(C_i\) between any two successive events.
  - IR2:
    - when \(i\) sends msg. \(m\) as an event \(a\): \(m\) contains a timestamp \(T_m = C_i(a)\).
    - when \(j\) receives as an event \(b\), it sets \(C_j := \max\{C_j, T_m + 1\}\).
Extend the minimum partial ordering obtained above to one possible total ordering.

Trick: use process identity ordering to give order to all concurrent relation.

Example: define \( a \triangleright b \) ("\( \Rightarrow \)" in the paper)

\[ C_i \langle a \rangle < C_j \langle b \rangle, \]
\[ C_i \langle a \rangle = C_j \langle b \rangle \land P_i < P_j. \]

"\( \prec \)" fairness: \( C_i \langle a \rangle = C_j \langle b \rangle \land j < i \Rightarrow a \triangleright b \) if \( j < C_i \langle a \rangle \) mod \( N \leq i. \)
A unified protocol for each of processes

Compete to acquire the lock & no pre-coordination

1. mutex lock semantics (safety),
2. ordered requests,
3. eventual release of every processes \(\implies\) every request will be granted. (liveness)
The ordering constraint makes the design non-trival! Imagine a plausible solution using a central scheduling process $P_0$

- $P_1$ sends a request to $P_0$,
- $P_1$ sends a msg. to $P_2$,
- $P_2$ sends a request to $P_0$.

$P_1$ should be granted because of the causal order.
The solution makes use of logical clocks to reorder the requests

- assume FIFO and reliable channels
- each process has a local queue that can buffer the reorder the requests
Logical Clocks: Case Study

- **Request:** $P_i$ sends "$T_m: P_i$ requests the resource" to every other proc. and puts onto its local queue.

- **Receive (req.):** on receiving "$T_m: P_i$ req. the res.", $P_j$ puts it into local queue and send ACK to $P_i$ (not needed if it has sent a msg. to $P_i$ with higher $T'_m$).

- **Release:** $P_i$ removes any corresponding request msgs. from local queue and sends "$T_m: P_i$ releases the res." to others.

- **Receive (rel.):** on receiving "$T_m: P_i$ release the res.", $P_j$ removes any corresponding request msgs. from $P_i$.

- **When granted:** (TBC).
When granted

- $T_m$: $P_i$ req. res.” in queue and **ordered first** (by “$\triangleright$” relation),
- $P_i$ received a msg. from every other procs. later than $T_m$ (all others know about the request).
Request or release the resource \(\implies\) operations on a global state.

State machine:
- states: \(s \in S\),
- commands: \(c \in C\),
- events that cause state transition: \(e : C \times S \rightarrow S, e(c, s) = s'\).

In the previous case: \(C = \{P_i \text{ requests}\} \cup \{P_i \text{ releases}\}\)

Each process has a local running instance of the state machine.

The order of executing commands is consistent.

State machine replication without fault tolerance.
How to address the issue?

- Give the user the responsibility for avoiding anomalous behavior (to express the external causality with manual timestamp).

- Introduce stronger clock condition:
  - Let “→” denote the happened-before relation for the set of all systems events (including “external” events).
  - ∀a, b : a→b \implies C(a) < C(b).
Physical Clocks

- $C_i(t)$ is differentiable function of $t$ except for isolated jump discontinuities where the clock is reset.

- True physical clock: $dC_i(t)/dt \approx 1$. 

![Graph showing $C_i(t)$ as a function of $t$ with a reset point.](image)
Physical Clocks

- PC1: \( \exists \) constant \( \kappa \ll 1 : \forall i, |dC_i(t)/dt - 1| < \kappa \). (physical property of a specific clock \( C_i \))
- PC2: \( \forall i, j : |C_i(t) - C_j(t)| < \epsilon \). (guaranteed by a carefully chosen protocol)
Let $\mu$ be a number: $\forall i, j, a \rightarrow b \implies$

- $a \in \text{process } i$,
- $b \in \text{process } j$,
- $a$ occurs at $t$,
- $b$ occurs later than $t + \mu$.

$\mu$ is less than the shortest transmission time for interprocess messaging.

To avoid anomalous behavior: $\forall i, j, t : C_i(t + \mu) - C_j(t) > 0$. 
Physical Clocks

- To avoid anomalous behavior: $\forall i, j, t : C_i(t + \mu) - C_j(t) > 0$.
- Resetting clocks: clocks are always reset forward. (why?)
- If PC1 and PC2 are guaranteed
  - From PC1, we have for same process $i$:
    $$C_i(t + \mu) - C_i(t) > (1 - \kappa)\mu.$$  
  - Combining with PC2, we have:
    $$\epsilon \leq \mu(1 - \kappa) \implies \mu \geq \frac{\epsilon}{1 - \kappa}$$
Combining with PC2, we have:

\[ \epsilon \leq \mu(1 - \kappa) \implies \mu \geq \frac{\epsilon}{1 - \kappa} \]

- How to guarantee PC2?
- What \( \epsilon \) can we get when ensuring PC2?
Physical Clocks

- Define total delay: $v_m = t' - t$.
- Minimum delay: $\mu_m \geq 0 : \mu_m \leq v_m$.
- Define unpredicatable delay: $\xi_m = v_m - \mu_m$. 


Physical Clocks

- IR1': \( \forall i, P_i \) does not receive msg. at \( t \) \( \implies \) \( C_i \) is differentiable at \( t \) and \( \frac{dC_i(t)}{dt} > 0 \) (> 0 is trivial because clocks never go backward).

- IR2':
  - \( P_i \) sends msg. at \( t \) that contains \( T_m = C_i(t) \),
  - Upon receiving \( m \) at \( t' \), \( P_j \) sets \( C_j(t') \) equal to
    \[
    \max \left\{ \lim_{\delta \to 0} C_j(t' - |\delta|), T_m + \mu_m \right\}
    \]
Theorem (proof is in Appendix A of the paper):

$$\epsilon \approx d \cdot (2\kappa \tau + \xi) \quad \forall t \geq t_0 + \tau d \quad (\text{assuming } \mu + \xi \ll \tau)$$

- $d$: the diameter of the communication graph among the processes.
- $\tau$: at least 1 msg. sent between $(t, t + \tau)$.
- Recall: given

$$\mu \geq \frac{\epsilon}{1 - \kappa}$$

then the anomalous behavior cannot happen.
Distributed Snapshots

- Distributed Snapshots: Determining Global States of Distributed Systems

- Dining philosophers problem.
- Chandy-Lamport algorithm.
- Three books and over a hundred papers on distributed computing, verification of concurrent programs, parallel programming languages and performance models of computing & communication systems.

K. Mani Chandy (1944–)
Distributed Snapshots

K. Mani Chandy (1944–)

- B.Tech. from Indian Institute of Technology.
- M.S. from Polytechnic Institute of Brooklyn.
- Ph.D. in Electrical Engineering from MIT.
- Simon Ramo Professor of Computer Science at Caltech.
- Member of National Academy of Engineering.
Distributed Snapshots

K. Mani Chandy (1944–)

- Worked for Honeywell and IBM.
- Was in CS department of UT Austin, serving as chair in 1978–79 and 1983–85.
- Story of the Chandy-Lamport algorithm according to Lamport’s website.
Assumption: a process can
- record its own state and the msgs. it sends and receives,
- nothing else!

A process \( p \) must enlist the cooperation of other procs. that must record their local states and send the recorded states to \( p \).

What makes a “snapshot”: a global state is a set of
- process states
- channel states: the buffered messages
How to make snapshot: analogy to taking a panorama photo.
Taking snapshots: How?

- How to make snapshot: analogy to taking a panorama photo.
- Different moments in different pieces, but together make a reasonable photo.
- Define “making sense” for distributed snapshots?
Taking snapshots: Why?

- Detect stable property of a predicate $y$ in the system $D$.
- Stable: $y(S) \rightarrow y(S')$, $\forall S'$ of $D$ reachable from $S$.
- $y$ is true $\implies y$ is always true.
Model

- Processes.
- Channels with
  - infinite buffer,
  - no error,
  - FIFO.
- Delay is arbitrary but finite.
- Events are
  - Atomic
  - \( e = \langle p, s, s', M, c \rangle \)
- Global state \( S \) consist of
  - Process states: \( s_1, s_2, \ldots \)
  - Channel states: a sequence of msgs. \( M_1, M_2, \ldots \)
Model: Example

\[
\begin{array}{c}
\text{p} \quad\xrightarrow{c} \quad\text{q} \\
\text{q} \quad\xleftarrow{c'} \quad\text{p}
\end{array}
\]

\[
\begin{array}{c}
\text{s0} \quad\xrightarrow{\text{send token}} \quad\text{s1} \\
\text{s1} \quad\xleftarrow{\text{receive token}} \quad\text{s0}
\end{array}
\]

\[
\begin{array}{c}
\text{s0} \quad\xrightarrow{\text{empty}} \quad\text{s0} \\
\text{s0} \quad\xrightarrow{\text{token}} \quad\text{s0}
\end{array}
\]

\[
\begin{array}{c}
\text{s0} \quad\xrightarrow{\text{empty}} \quad\text{s0} \\
\text{s0} \quad\xrightarrow{\text{empty}} \quad\text{s0}
\end{array}
\]

\[
\begin{array}{c}
\text{s0} \quad\xrightarrow{\text{empty}} \quad\text{s1} \\
\text{s1} \quad\xleftarrow{\text{empty}} \quad\text{s0}
\end{array}
\]
Motivation: see 3.1 of the paper.

Some processes spontaneously start to record their states.

For each process \( p \): sends one marker along \( c \) (the channel directed away from \( p \)) after recoding its state and before it sends further msgs.

For each process \( q \) receiving a marker from channel \( c \)

- if \( q \) has not recorded its state
  - \( q \) records its state,
  - \( q \) records the state of \( c \) as empty;
- otherwise, \( q \) records the state of \( c \) as the sequence of msgs. received along \( c \)
  - after \( q \)’s state was recoreded,
  - before \( q \) received the marker along \( c \).
○ Termination?

○ Has the recorded global state ever happened in the system?
Algorithm: Discuss

- Has the recorded global state ever happened in the system? (Not always)
- Locally “consistent” ≠ globally “consistent”.
Algorithm: Discuss

Diagram of state transitions and actions.

States:
- s0: receive token
- s1: send token

Actions:
- empty
- token

Transitions:
- p to q: c, c'
- s0 to s1: receive token
- s1 to s0: send token
○ Define “happened”?
Algorithm: Properties and Proof

Let seq = (e_i, 0 ≤ i) be a distributed computation.

S_{i-1} \xrightarrow{e_{i-1}} S_i.

Initiated in S_ι, terminated in S_φ.

Show that for the captured snapshot S^*
  ◦ S^* is reachable from S_ι,
  ◦ S_φ is reachable from S^*.
Show that for the captured snapshot $S^*$
- $S^*$ is reachable from $S_\ell$,
- $S_\phi$ is reachable from $S^*$.

$\exists seq'$
- $seq'$ is a permutation of seq,
- $S_\ell = S^*$ or $S_\ell$ occurs earlier than $S^*$,
- $S_\phi = S^*$ or $S^*$ occurs earlier than $S_\phi$. 
○ Define $e_i$ is
  ○ “prerecording” (pre.) iff. $e_i$ is in proc. $p$ and $p$ records its state after $e_i$ (somewhere) in seq.
  ○ “postrecording” (post.) o.w.

○ If not ALL pre. preceds post. $\exists j$
  \[ \underbrace{\ldots, e_{j-1}, e_j, \ldots}_{\text{post. \ pre.}} \]
  ○ then $\ldots, e_j, e_{j-1}, \ldots$ is also a computation.
Algorithm: Proof

- If not ALL pre. precedes post. \( \exists j \)
  - \( \ldots, e_{j-1}, e_j, \ldots \)
  - then \( \ldots, e_j, e_{j-1}, \ldots \) is also a computation.

- \( e_{j-1} \) and \( e_j \) must be on different procs. (because \( e_{j-1} \) is post., \( j - 1 < j \)).

- Assume \( e_{j-1} \) occurs at \( p \), \( e_j \) occurs at \( q \), and \( p \neq q \).

- There CANNOT be a msg. sent at \( e_{j-1} \) and received at \( e_j \)
  - a msg. sent along \( c \) when \( e_{j-1} \) occurs \( \implies \) a marker must have been sent long \( c \) before \( e_{j-1} \) (by definition of post. events).
  - a msg. received along \( c \) when \( e_j \) occurs \( \implies \) a marker must have been received long \( c \) before \( e_j \) (FIFO) \( \implies e_j \) is post. too (on receiving a marker, a process records its state). Contradiction!
Algorithm: Proof

- Assume $e_{j-1}$ occurs at $p$, $e_j$ occurs at $q$, and $p \neq q$.
- There CANNOT be a msg. sent at $e_{j-1}$ and received at $e_j$. (proved, channel state is unchanged)
- State of $q$ is not altered by the occurrence of $e_{j-1}$: because of different procs.
  - If $e_j$ at $q$ receives $M$ along $c$, then $M$ must have been the msg. at the head of $c$ before $e_{j-1} \implies e_j$ can occur in $S_{j-1}$.
- State of $p$ is not altered by the occurrence of $e_j$
  - $e_j$ happens after $p$ and at a different process $\implies e_{j-1}$ can occur after $e_j$. 
Therefore

- \( \ldots, e_{j-2}, e_j, e_{j-1}, \ldots \) is a valid computation,
- the global state after \( e_1, \ldots, e_{j-2}, e_j, e_{j-1} \) is the same as \( e_1, \ldots, e_{j-2}, e_{j-1}, e_j \).

With the invariants held, such swapping can be done repetitively, until

- all pre. events precede post. events,
- seq is a computation,
- \( \forall i, i < \iota \text{ or } i \geq \phi : e'_i = e_i \), and
- \( \forall i, i \leq \iota \text{ or } i \geq \phi : S'_i = S_i \).
With the invariants held, such swapping can be done repetitively, until

- all pre. events precede post. events,
- seq is a computation,
- $\forall i, i < \iota$ or $i \geq \phi : e'_i = e_i$, and
- $\forall i, i \leq \iota$ or $i \geq \phi : S'_i = S_i$.

Finally, we need to show the state $\tilde{S}$ in the middle (after all pre. before all post.) is $S^*$ (recorded snapshot).

Equivalently

- the state of $\forall p$ is the same,
- the state of $\forall c$ is the same.
Equivalently

- the state of \( \forall p \) is the same,
  - by noticing the state of a process can only be changed by events,
  - all posts. events are after the state \( \bar{S} \);
- the state of \( \forall c \) is the same:

  \[
  (\text{msgs. of pre. send of } c) - (\text{msgs. of pre. receive of } c) = \text{msgs. taken in the snapshot of } c
  \]

- msgs. of pre. send of \( c \) =
  - (i) msgs. sent by \( p \) before sending a marker,
- msgs. of pre. receive of \( c \) =
  - (ii) msgs. received by \( q \) before recording,
- (i) \(-\) (ii) = msgs. in the snapshot.
Distributed Snapshot: Stability Detection

- **Input:** a stable property \( y \)
- **Output:** A boolean value definite with the property
  - \( y(S_t) \rightarrow \text{definite} \)
  - \( \text{definite} \rightarrow y(S_\phi) \)

- **Implementation**
  - record a global state \( S^* \),
  - definite := \( y(S^*) \).

- **Correctness**
  - \( S^* \) is reachable from \( S_t \),
  - \( S_\phi \) is reachable from \( S^* \), and
  - \( y(S) \rightarrow y(S') \) \( \forall S' \) reachable from \( S \) (definition of a stable property).
Thank you!

Q & A