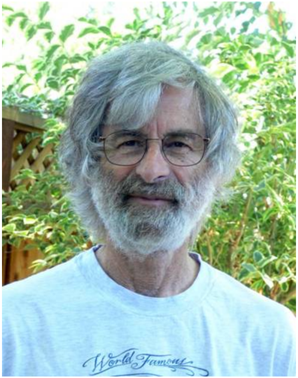
Distributed Systems: Ordering and Consistent Cuts

by Maofan (Ted) Yin
my428@cornell.edu

Time, Clocks and the Ordering of Events

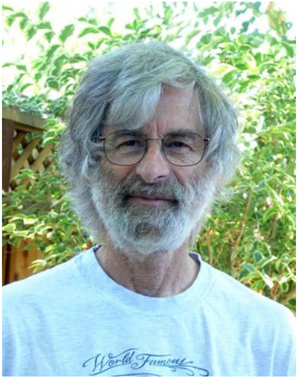
- *Time, Clocks, and the Ordering of Events in a Distributed System*



Leslie B. Lamport (1941–)

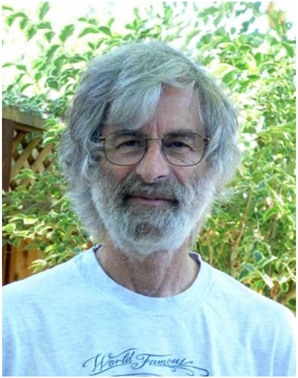
- The original author of LaTeX
- Sequential consistency
- Atomic register hierarchy
- Lamport's bakery algorithm
- Byzantine fault tolerance
- Paxos
- Lamport signature

Time, Clocks and the Ordering of Events



Leslie B. Lamport (1941–)

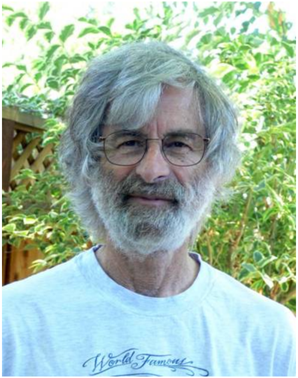
- B.S. in mathematics from MIT
- M.A. and Ph.D. in mathematics from Brandeis University
- Dijkstra Prize (2000, because of this paper, and 2005)
- IEEE Emanuel R. Piore Award (2004)
- IEEE John von Neumann Medal (2008)
- ACM A.M. Turing Award (2013)
- ACM Fellow (2014)



Leslie B. Lamport (1941–)

- “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. ”

Time, Clocks and the Ordering of Events



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.”

Time and Systems

> *"The only reason of time is so that everything does not happen at once."*
>
> — Albert Einstein

- Something happened at 3:15: ocurred within $[3:15, 3:16)$.
- Why time is so important? Air ticket reservation, online shopping, etc.

Time and Systems

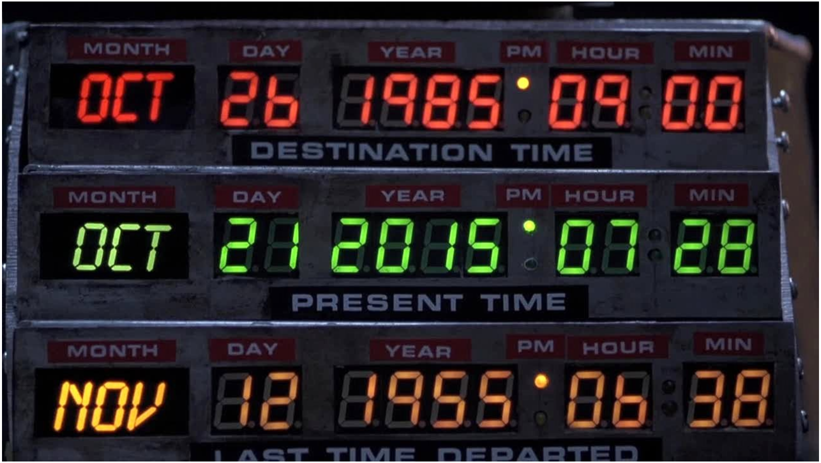
> *"The only reason of time is so that everything does not happen at once."*
>
> — Albert Einstein

- 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 occured first: *partial ordering.*

Time and Systems

> *"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.


MONTH
DAY
YEAR
PM
HOUR
MIN
OCT 26 1985 09 00
DESTINATION TIME
MONTH
DAY
YEAR
PM
HOUR
MIN
OCT 21 2015 07 28
PRESENT TIME
MONTH
DAY
YEAR
PM
HOUR
MIN
NOV 12 1955 06 38

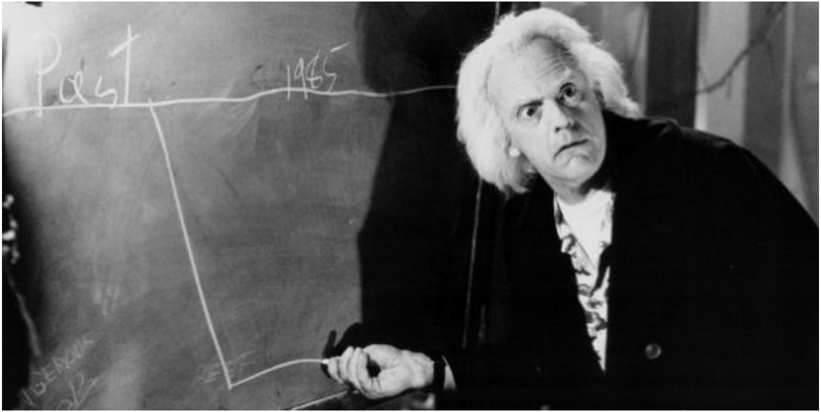
Clocks

In this paper, two clock implementations are introduced

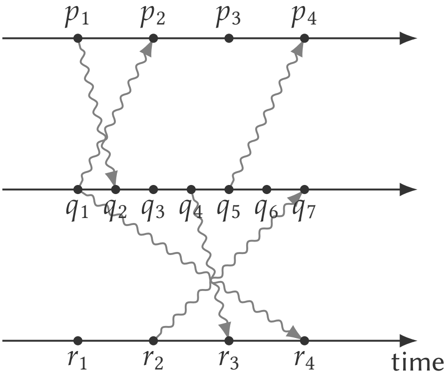
- **Logical clocks:**
  - works without the help of any physical equipment,
  - causes anomaly with external happened-before relation (the clock is confined within the system).
- **Physical clocks:**
  - works when physical clocks have certain precision,
  - but provides with strong relation.

- We have
  - A priori: total ordering of events in the same process
  - Msgs. can carry time info
- We want to archieve
  - A relation $a \to b$ that
    1. $a, b \in$ same process, a comes before $b \Longrightarrow a \to b$,
    2. a sends a msg. to $b \Longrightarrow a \to b$,
    3. $a \to b \land b \to c \Longrightarrow a \to c$.
  - Remarks:
    - a and b are *concurrent* if $a \not\to b \land b \not\to a$.
    - $a \not\to a$ (irreflexivity),
    - $a \to b \land b \to c \Longrightarrow a \to c$ (transitivity),
    - $a \to b \Longrightarrow b \not\to a$ (asymmetry).

Logical Clocks: Space-Time Diagram



Logical Clocks: Space-Time Diagram



- 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.

Logical Clocks: Design

- 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 "$\rightarrow$" relation, we want $\forall a, b$

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

-

$$\text{not vice versa: } a \rightarrow b \Leftrightarrow C\langle a\rangle < C\langle b\rangle$$

otherwise,

$$e \not\rightarrow e' \wedge e' \not\rightarrow e \implies C\langle e\rangle \not< C\langle e'\rangle \wedge C\langle e\rangle \not> C\langle e'\rangle$$
$$\implies C\langle e\rangle = C\langle e'\rangle$$

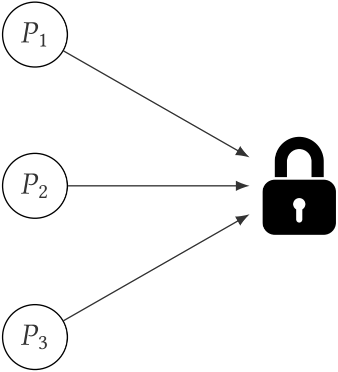
Logical Clocks: Design

- Clock condition is held if
  - C1: $a, b \in$ proc. i: a is before $b \implies C_i\langle a\rangle < C_i\langle b\rangle$.
  - C2: i sends msg. as event a to j as event b: $C_i\langle a\rangle < C_j\langle b\rangle$.
- 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 $T_m = C_i\langle a\rangle$,
    - when j receives as an event b, it sets $C_j := \max\left\{C_j, T_m + 1\right\}$.

Logical Clocks: Partial to Total Ordering

- 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)
  - $C_i\langle a\rangle < C_j\langle b\rangle$,
  - $C_i\langle a\rangle = C_j\langle b\rangle \wedge P_i \prec P_j$.
- "$\prec$" fairness: $C_i\langle a\rangle = C_j\langle b\rangle \wedge j < i \Longrightarrow a \triangleright b$ if $j < C_i\langle a\rangle \bmod N \leq i$.

Logical Clocks: Case Study



- 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)

Logical Clocks: Case Study

The ordering constaint 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.

P1 should be granted because of the causal order.

Logical Clocks: Case Study

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 resouce" to every other procs. 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).

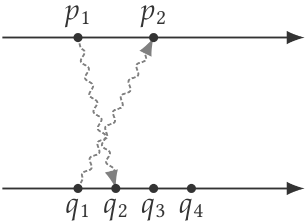
Logical Clocks: Case Study

- 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).

Logical Clocks: Case Study Generalization

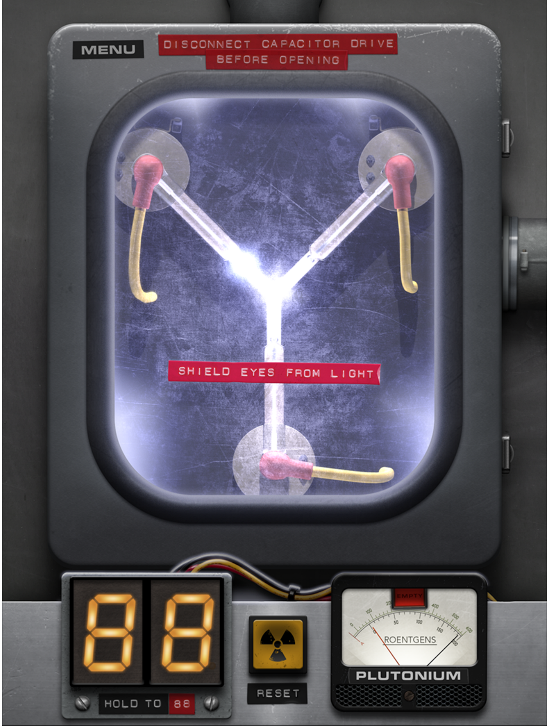
- Request or release the resource $\Longrightarrow$ operations on a global state.
- State machine:
  - states: $s \in S$,
  - commands: $c \in C$,
  - events that cause state transition: $e : C \times S \rightarrow S$, $e(c, s) = s'$.
- In the previous case: $C = \{P_i \text{ requests}\} \cup \{P_i \text{ releases}\}$
- Each process has a local running instance of the state machine.
- The order of executing commands is consistent.
- State machine replication without fault tolerance.

Logical Clocks: Anomalous Behavior

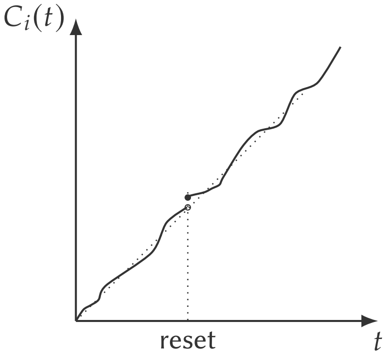


How to address the issue?

- Give the user the responsibility for avoiding anomalous behavior (to express the external causality with manual timestamp).
- Introduce stronger clock condition:
  - Let "$\boldsymbol{\rightarrow}$" denote the happened-before relation for the set of *all* systems events (including "external" events).
  - $\forall a, b : a \boldsymbol{\rightarrow} b \implies C\langle a \rangle < C\langle b \rangle$.


MENU
DISCONNECT CAPACITOR DRIVE
BEFORE OPENING
SHIELD EYES FROM LIGHT
88
HOLD TO 88
RESET
EMPTY
ROENTGENS
PLUTONIUM

Physical Clocks



- $C_i(t)$ is differentiable function of t except for isolated jump discontinuities where the clock is reset.
- True physical clock: $\mathrm{d}C_i(t)/\mathrm{d}t \approx 1$.

Physical Clocks

- PC1: $\exists$ constant $\kappa \ll 1 : \forall i, |\mathrm{d}C_i(t)/\mathrm{d}t - 1| < \kappa$. (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)

Physical Clocks

- Let μ be a number: $\forall i, j, a \to b \implies$
  - $a \in$ process i,
  - $b \in$ process j,
  - a occurs at t,
  - b occurs later than $t + \mu$.
- μ 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:

$$\epsilon \leq \mu(1 - \kappa) \Longrightarrow \mu \geq \frac{\epsilon}{1 - \kappa}$$

Physical Clocks

- Combining with PC2, we have:

$$\epsilon \leq \mu(1-\kappa) \Longrightarrow \mu \geq \frac{\epsilon}{1-\kappa}$$

- How to guarantee PC2?
- What ϵ 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 $\mathrm{d}C_i(t)/\mathrm{d}t > 0$ (> 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

$$\max\left\{\lim_{\delta\to 0} C_j(t' - |\delta|), T_m + \mu_m\right\}$$

Physical Clocks

- Theorem (proof is in Appendix A of the paper):

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

- d: the diameter of the communication graph among the processes.
- τ: 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*

K. Mani Chandy (1944–)

- 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–)

- 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.
- Memeber of National Academy of Engineering.
- A. A. Michelson Award (1985).
- IEEE Koji Kobayashi Award (1987).

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.

Taking snapshots: What?

- 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

Taking snapshots: How?

- How to make snapshot: anology to taking a panorama photo.

Taking snapshots: How?

- How to make snapshot: anology to taking a panorama photo.
- Different moments in different pieces, but together make a reasonable photo.
- Define "making sense" for distributed snapshots?

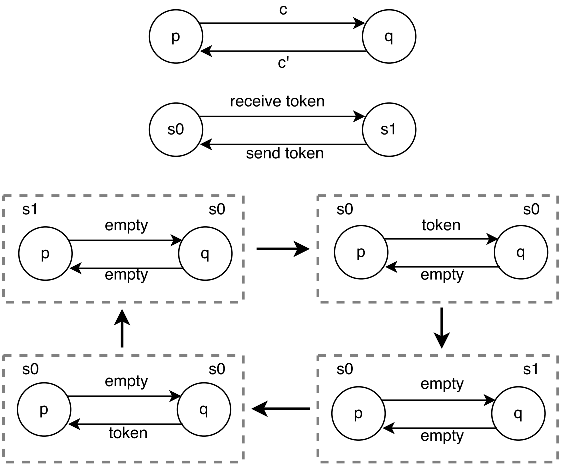
Taking snapshots: Why?

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

Model

- Processes.
- Channels with
  - infinite buffer,
  - no error,
  - FIFO.
- Delay is arbitrary but finte.
- 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



Algorithm

- 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.

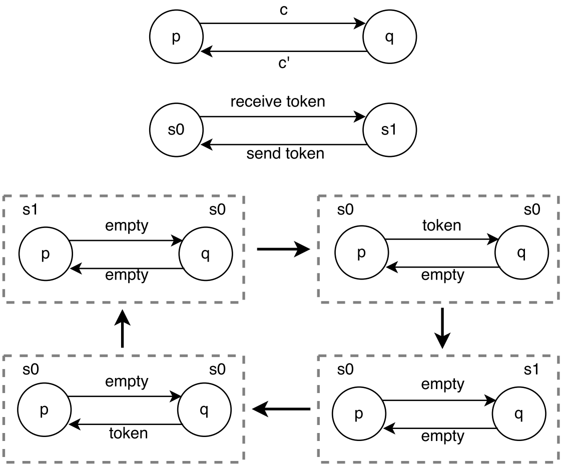
Algorithm: Discuss

- 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” $\neq$ globally “consistent”.

Algorithm: Discuss



Algorithm: Discuss

- Define “happened”?

Algorithm: Properties and Proof

- Let seq $= (e_i, 0 \leq i)$ 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^*.

Algorithm: Proof

- Show that for the captured snapshot S^*
  - S^* is reachable from S_ι,
  - S_ϕ is reachable from S^*.
- $\exists\,\mathrm{seq}'$
  - seq' is a permutation of seq,
  - $S_\iota = S^*$ or S_ι occurs earlier than S^*,
  - $S_\phi = S^*$ or S^* occurs ealier than S_ϕ.

Algorithm: Proof

- 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. $\exists j$
  - $\ldots, \overbrace{e_{j-1}}^{\text{post.}}, \overbrace{e_j}^{\text{pre.}}, \ldots$
  - then $\ldots, e_j, e_{j-1}, \ldots$ is also a computation.

Algorithm: Proof

- If not ALL pre. precedes post. $\exists j$
  - $\ldots, \overbrace{e_{j-1}}^{\text{post.}}, \overbrace{e_j}^{\text{pre.}}, \ldots$
  - 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 $\Longrightarrow$ 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 $\Longrightarrow$ a marker must have been received long c before e_j (FIFO) $\Longrightarrow$ 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 occurence 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 occurence of e_j
  - e_j happens after p and at a different process $\implies e_{j-1}$ can occur after e_j.

Algorithm: Proof

- Therefore
  - $\ldots, e_{j-2}, \boldsymbol{e}_j, \boldsymbol{e}_{j-1}, \ldots$ is a valid computation,
  - the global state after $e_1, \ldots, e_{j-2}, \boldsymbol{e}_j, \boldsymbol{e}_{j-1}$ is the same as $e_1, \ldots, e_{j-2}, \boldsymbol{e}_{j-1}, \boldsymbol{e}_j$.
- With the invariants held, such swapping can be done repetitively, until
  - all pre. events precede post. events,
  - seq is a computation,
  - $\forall i, i < \iota$ or $i \geq \phi : e'_i = e_i$, and
  - $\forall i, i \leq \iota$ or $i \geq \phi : S'_i = S_i$.

Algorithm: Proof

- 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 $\bar{S}$ 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.

Algorithm: Proof

- 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 booleam value definite with the property
  - $y(S_\iota) \longrightarrow \text{definite}$
  - $\text{definite} \longrightarrow y(S_\phi)$
- Implementation
  - record a global state S^*,
  - $\text{definite} \coloneqq y(S^*)$.
- Correctness
  - $S*$ is reachable from S_ι,
  - S_ϕ is reachable from S^*, and
  - $y(S) \longrightarrow y(S') \quad \forall S'$ reachable from S (definition of a stable property).

Thank you!

Q & A