Table of Contents

- Introduction and Motivation
- Theoretical Foundations
- Distributed Programming Languages
- Distributed Operating Systems
- Distributed Communication
- Distributed Data Management
- Reliability
- Applications
- Conclusions
- Appendix
State Model

- A process executes three types of events: internal actions, send actions, and receive actions.
- A global state (also configuration): a collection of local states and the state of all the communication channels.
- Global state evolves by means of transitions
- Initiator: first event
- Distributed algorithm: multiple initiators

System structure from logical point of view.
Thread

- lightweight process (maintain minimum information in its context)
- multiple threads of control per process
- multithreaded servers (vs. single-threaded process)

A multithreaded server in a dispatcher/worker model.
Preliminary

**Assertions:** a predicate on the configurations of an algorithm

Invariant, such as loop invariant, is an assertion

e.g., \{I\} while c body \{\neg c \land I\} (under Floyd-Hoare logic)

calculate sum: 1+2+…+n, two assertions I: 1+2+…+k and c: k < n

**Safety property:** if it is true in each reachable configuration

i.e., something bad will never happen (e.g., absence of deadlock, mutual exclusion, partial correctness)

**Liveness property:** if executions, from some point on, contain a configuration in which the assertion holds

i.e., something good will eventually happen (e.g., fairness, termination)

**Fair:** if every event that can happen in infinitely many times is performed infinitely often

**Complexity:** time, space, message (bit) complexity
The happened-before relation (denoted by $\rightarrow$) is defined as follows:

- **Rule 1**: If $a$ and $b$ are events in the same process and $a$ was executed before $b$, then $a \rightarrow b$.
- **Rule 2**: If $a$ is the event of sending a message by one process and $b$ is the event of receiving that message by another process, then $a \rightarrow b$.
- **Rule 3**: If $a \rightarrow b$ and $b \rightarrow c$, then $a \rightarrow c$. 
Relationship Between Two Events

- Two events $a$ and $b$ are **causally related** if $a \rightarrow b$ or $b \rightarrow a$.

- Two distinct events $a$ and $b$ are said to be **concurrent** if $a \not\rightarrow b$ and $b \not\rightarrow a$ (denoted as $a \parallel b$).
Example 2

A time-space view of a distributed system.
Example 2 (Cont’d.)

- **Rule 1:**
  \[
  a_0 \rightarrow a_1 \rightarrow a_2 \rightarrow a_3 \\
  b_0 \rightarrow b_1 \rightarrow b_2 \rightarrow b_3 \\
  c_0 \rightarrow c_1 \rightarrow c_2 \rightarrow c_3 
  \]

- **Rule 2:**
  \[
  a_0 \rightarrow b_3 \\
  b_1 \rightarrow a_3, b_2 \rightarrow c_1, b_0 \rightarrow c_2 
  \]
Example 3

An example of a network of a bank system.
Example 3 (Cont’d.)

A sequence of global states.
Consistent Global State

Four types of cut that cross a message transmission line.
A cut is consistent iff no two cut events are causally related.

- **Strongly consistent**: no (c) and (d).
- **Consistent**: no (d) (orphan message).
- **Inconsistent**: with (d).
Focus 3: Snapshot of Global States

A simple distribute algorithm to capture a consistent global state.

A system with three processes $P_i$, $P_j$, and $P_k$.

Many key concepts: asynchronous computation, global state, information propagation and gathering, …
Chandy and Lamport's Solution

- **Rule for sender $P$**:
  - $P$ records its local state
  - $P$ sends a marker along all the channels on which a marker has not been sent.

- **Rule for receiver $Q$**:
  /* on receipt of a marker along a channel $chan$ */
  - $Q$ has not recorded its state $\rightarrow$
    - record the state of $chan$ as an empty sequence and
      follow the "Rule for sender"
  - $Q$ has recorded its state $\rightarrow$
    - record the state of $chan$ as the sequence of messages received along $chan$
      after the latest state recording but before receiving the marker
Chandy and Lamport's Solution (Cont’d.)

- It can be applied in any system with FIFO channels (but with variable communication delays).
- The initiator for each process becomes the parent of the process, forming a spanning tree for result collection.
- It can be applied when more than one process initiates the process at the same time.
Chandy and Lamport's Solution (Cont’d.)

- Distributed algorithm: message-passing
- Distributed snapshot
- Dynamic spanning tree
- Asynchronous systems
- Message dissemination
- Progress termination
- Program debugging
  - Breakpoint
- Simulation
  - Physical and logical processes (event-driven)
Synchronous vs. Asynchronous Systems

Asynchronous Systems:
- Each node is driven by its own (independent) local clock.
- The transmission delay is finite but unpredictable.

Synchronous Systems:
- All nodes are driven by the global clock, which generates intervals (also rounds) of fixed, nonzero duration.
- The transmission delay is nonzero, but strictly less than the duration of an interval.
Distributed Algorithms: Traversal

Tarry’s algorithm:

- A process forwards the token through the same channel once.
- A process forwards the token to its parent only when there is no other option.

Complexity: 2E messages and at most 2E time units.
Extensions to avoid visited nodes:

- Include the IDs of visited nodes
  Complexity: 2(N-1) in time and in messages, but O(N log N) in bit complexity

- Awerbuch’s extension: the first-time process with the token informs its neighbors
  Complexity: 4N-2 in time and 4E in messages

- Cidon’s extension: improves on Awerbuch’s extension
  Complexity: 2(N-1) in time and 4E in messages
Distributed Algorithms: Wave-and-Echo

Wave-and-Echo algorithm (also for counting connected nodes)
- **Initiator** starts by sending a token to all its neighbors.
- When a node receives a token for the first time, it makes the sender its parent, and sends the token to all its neighbors.
- When a node has received messages from all its neighbors, it sends a message to its parent.
- When the **initiator** has received messages from all its neighbors, it stops.

General wave (-and-echo) algorithm (also for information propagation)
- A process often needs to gather information from all other processes.
- Usually the process starts with an initiator and ends with the same imitator (after collecting all data/results from all other processes).
- When the wave algorithm is issued at multiple nodes. Many waves, except one, will fail
Distributed Algorithms: Termination

Dijkstra-Scholten (tree-based):

- The initiator of the root of the tree.
- Upon receiving a message:
  - If the receiving process is currently not in the tree: the process joins the tree by becoming a child of the sender.
  - If the receiving process is already in the computation: the process immediately sends an acknowledgment message to the sender.
- When a process has no more children and has become idle, the process detaches itself from the tree by sending an acknowledgment to its tree parent.
- Termination occurs when the initiator has no children and has become idle.

Example: global snapshot (with one king)
Distributed Algorithms: Termination (cont’d)

Shavit-Francez (forest-based):

- Same as Dijkstra-Scholten, except with multiple initiators.
- Each non-initiator joining one tree.
- Termination detection initiated by multiple initiators through a wave algorithm.
  Example: global snapshot (with multiple kings)

Other termination algorithms:

- **Weight-throwing** algorithm: dividing a fixed weight over the active processes
- **Rana’s** algorithm: waves tagged with logical clocks
- **Safra’s** algorithm: token-based traversal
Other Algorithms: Parallel Algorithms

**PRAM model**
- Parallel random access memory
- EREW, ERCW, CREW, CRCW models
- Chap. 2 of JaJa’s “an introduction to parallel algorithms”

**BSP model by L. Valiant (1990)**
- Bulk synchronous parallel (BSP)
- Sequential composition of “supersteps”
  - Local computation
  - Process communication
  - Barrier synchronization
Parallel Algorithm: Bitonic sorter by K. Batcher

- Sorting network based on **Bitonic sequence**
  - Up-then-Down or Down-then-Up
  - $O(n \log^2(n))$ comparators
  - $O(\log^2(n))$ latency
- Also Batcher’s **odd-even sort** (small $\rightarrow$ large)
Barrier Synchronization

- **Sequential**: One process $p$ (leader, through leader election if needed)
  - Process $p$ issues wave-and-echo to all nodes
  - Process $p$ indicates next round to all nodes

- **Parallel**: Processes $p_0$, $p_1$, ..., $p_{N-1}$, $n$ starts from 0 until $\log_2 N - 1$
  - Notifies process $p_{(i+2^n) \mod N}$
  - Waits for notification by process $p_{(i-2^n) \mod N}$, and
  - Processes to round $n+1$
Focus 4: Lamport's Logical Clocks

Based on a “happen-before” relation that defines a partial order on events

- **Rule**₁. Before producing an event (an external send or internal event), we update $LC$:
  \[ LC_i = LC_i + d \quad (d > 0) \]
  ($d$ can have a different value at each application of **Rule**₁)

- **Rule**₂. When it receives the time-stamped message $(m, LC_j, j)$, $P_i$ executes the update
  \[ LC_i = \max \{ Lc_i, LC_j \} + d \quad (d > 0) \]
A total order based on the partial order derived from the happen-before relation

\[ a \ ( \text{in } P_i ) \Rightarrow b \ ( \text{in } P_j ) \]

to

iff

(1) \( LC(a) < LC(b) \) or (2) \( LC(a) = LC(b) \) and \( P_i < P_j \)

where < is an arbitrary total ordering of the process set, e.g., <can be defined as \( P_i < P_j \) iff \( i < j \).

A total order of events in the table for Example 2:

\[ a_0 \ b_0 \ c_0 \ a_1 \ b_1 \ a_2 \ b_2 \ a_3 \ b_3 \ c_1 \ c_2 \ c_3 \]
Vector and Matrix Logical Clock

Linear clock: if $a \rightarrow b$ then $LC_a < LC_b$

Vector clock: $a \rightarrow b$ iff $LC_a < LC_b$

Each $P_i$ is associated with a vector $LC_i[1..n]$, where
- $LC_i[i]$ describes the progress of $P_i$, i.e., its own process.
- $LC_i[j]$ represents $P_i$’s knowledge of $P_j$'s progress.
- The $LC_i[1..n]$ constitutes $P_i$’s local view of the logical global time.
Vector and Matrix Logical Clock (Cont’d.)

When \( d = 1 \) and \( init = 0 \)

- \( LC_i[i] \) counts the number of internal events
- \( LC_i[j] \) corresponds to the number of events produced by \( P_j \) that causally precede the current event at \( P_i \).

Knowledge and implicitly knowledge
Vector and Matrix Logical Clock (Cont’d.)

- **Rule 1.** Before producing an event (an external send or internal event), we update $LC_i[i]$:

\[
LC_i[i] := LC_i[i] + d \quad (d > 0)
\]

- **Rule 2.** Each message piggybacks the vector clock of the sender at sending time. When receiving a message $(m, LC_j, j)$, $P_i$ executes the update.

\[
LC_i[k] := \max (LC_i[k]; LC_j[k]), \quad 1 \leq k \leq n
\]

\[
LC_i[i] := LC_i[i] + d
\]
Example 4

An example of vector clocks.
Example 5: Totally-Ordered Multicasting

- Two copies of the account at A and B (with balance of $10,000).
- Update 1: add $1,000 at A.
- Update 2: add interests (based on 1% interest rate) at B.
- Update 1 followed by Update 2: $11,110.
- Update 2 followed by Update 1: $11,100.
Example 6: Application of Vector Clock

Internet electronic bulletin board service

When receiving $m$ with vector clock $LC_j$ from process $j$, $P_i$ inspects timestamp $LC_j$ and will postpone delivery until all messages that causally precede $m$ have been received.
Matrix Logical Clock

Each $P_i$ is associated with a matrix $LC_i[1..n, 1..n]$ where

- $LC_i[i, i]$ is the local logical clock.
- $LC_i[k, l]$ represents the view (or knowledge) $P_i$ has about $P_k$'s knowledge about the local logical clock of $P_l$.

If

$$\min(LC_i[k, i]) \geq t$$

then $P_i$ knows that every other process knows its progress until its local time $t$. 
Physical Clock

- Correct rate condition:
  \[ \forall_i \left| \frac{dPC_i(t)}{dt} - 1 \right| < \alpha \]

- Clock synchronization condition:
  \[ \forall_i \forall_j |PC_i(t) - PC_j(t)| < \beta \]
Lamport's Logical Clock Rules for Physical Clock

- For each $i$, if $P_i$ does not receive a message at physical time $t$, then $PC_i$ is differentiable at $t$ and $dPC(t)/dt > 0$.
- If $P_i$ sends a message $m$ at physical time $t$, then $m$ contains $PC_i(t)$.
- Upon receiving a message $(m, PC_j)$ at time $t$, process $P_i$ sets $PC_i$ to maximum $(PC_i(t - 0), PC_j + \mu_m)$ where $\mu_m$ is a predetermined minimum delay to send message $m$ from one process to another process.
Focus 5: Clock Synchronization

- UNIX make program:
  - Re-compile when *file.c*'s time is large than *file.o*'s.
  - Problem occurs when source and object files are generated at different machines with no global agreement on time.

- Maximum drift rate $\rho$: $1-\rho \leq \frac{dPC}{dt} \leq 1+\rho$
  - Two clocks (with opposite drift rate $\rho$) may be $2\rho\Delta t$ apart at a time $\Delta$ after last synchronization.
  - Clocks must be resynchronized at least every $\delta/2\rho$ seconds in order to guarantee that they will be differ by no more than $\delta$. 
Cristian's Algorithm

- Each machine sends a request every $\delta/2\rho$ seconds.
- Time server returns its current time $PC_{UTC}$ (UTC: Universal Coordinate Time).
- Each machines changes its clock (normally set forward or slow down its rate).
- Delay estimation: $(T_r - T_s - I)/2$, where $T_r$ is receive time, $T_s$ send time, and $I$ interrupt handling time.
Cristian's Algorithm (Cont’d.)

Getting correct time from a time server.
Exercise 2

1. Consider a system where processes can be dynamically created or terminated. A process can generate a new process. For example, $P_1$ generates both $P_2$ and $P_3$. Modify the happened-before relation and the linear logical clock scheme for events in such a dynamic set of processes.

2. For the distributed system shown in the figure below.
Exercise 2 (Cont’d)

- Provide all the pairs of events that are related.
- Provide logical time for all the events using
  - linear time, and
  - vector time
- Assume that each $LC_i$ is initialized to zero and $d = 1$.

3. Provide linear logical clocks for all the events in the system given in Problem 2. Assume that all $LC$'s are initialized to zero and the $d$'s for $P_a$, $P_b$, and $P_c$ are 1, 2, 3, respectively. Does condition $a \rightarrow b \Rightarrow LC(a) < LC(b)$ still hold? For any other set of $d$'s? and why?

4. Traversal on graph $\{(a, b), (b, c), (b, d), (c, e), (d, e), (e, f)\}$ using Terry’s solution and Awerbuch’s extension.

5. Show details of sorting $(4, 6, 1, 3, 5, 8, 7, 2)$ and $(1, 4, 8, 7, 2, 6, 5, 3)$ on an 8-input-and-8-output Batcher’s Even-Odd sorting network.