Theoretical Aspects

- Logical Clocks
- Causal Ordering
- Global State Recording
- Termination Detection
Lamport’s Clock

- Happened before relation:
  - a -> b : Event a occurred before event b. Events in the same process.
  - a -> b : If a is the event of sending a message m in a process and b is the event of receipt of the same message m by another process.
  - a -> b, b -> c, then a -> c. “->” is transitive.

- Causally Ordered Events
  - a -> b : Event a “causally” affects event b

- Concurrent Events
  - a || b: if a !-> b and b !-> a
Space-time Diagram

Space

\[ P_1 \quad e_{11} \quad e_{12} \quad e_{13} \quad e_{14} \]

Internal
Events

\[ P_2 \]

\[ e_{21} \quad e_{22} \quad e_{23} \quad e_{24} \]

Messages

Time
Logical Clocks

- **Conditions satisfied:**
  - Ci is clock in Process Pi.
  - If a -> b in process Pi, Ci(a) < Ci(b)
  - Let a: sending message m in Pi; b : receiving message m in Pj; then, Ci(a) < Cj(b).

- **Implementation Rules:**
  - R1: Ci = Ci + d (d > 0); clock is updated between two successive events.
  - R2: Cj = max(Cj, tm) + d; (d > 0); When Pj receives a message m with a time stamp tm (tm assigned by Pi, the sender; tm = Ci(a), a being the event of sending message m).

- A reasonable value for d is 1
Space-time Diagram

Space

Time

P1 e11 e12 e13 e14 e15 e16 e17
(1) (2) (3) (4) (5) (6) (7)

P2 e21 e22 e23 e24 e25
(1) (2) (3) (4) (7)
C(e11) < C(e32) but not causally related.
This inter-dependency not reflected in Lamport’s Clock.
Vector Clocks

- Keep track of transitive dependencies among processes for recovery purposes.
- $C_i[1..n]$: is a “vector” clock at process $P_i$ whose entries are the “assumed”/”best guess” clock values of different processes.
- $C_i[j]$ (j != i) is the best guess of $P_i$ for $P_j$’s clock.
- Vector clock rules:
  - $C_i[i] = C_i[i] + d$, (d > 0); for successive events in $P_i$
  - For all k, $C_j[k] = \max (C_j[k], tm[k])$, when a message $m$ with time stamp $tm$ is received by $P_j$ from $P_i$. 
  \[ : \text{For all} \]
Vector Clocks Comparison

1. Equal: $ta = tb$ iff $\forall i, ta[i] = tb[i]$
2. Not Equal: $ta \neq tb$ iff $ta[i] \neq tb[i]$, for at least one $i$
3. Less than or equal: $ta \leq tb$ iff $ta[i] \leq tb[i]$, for all $i$
4. Less than: $ta < tb$ iff $ta[i] \leq tb[i]$ and $ta[i] \neq tb[i]$, for all $i$
6. Not less than or equal ...
7. Not less than ..
Vector Clock ...

Space

P1  e11  e12  e13
(1,0,0) (2,0,0) (3,4,1)

P2
(0,1,0) (2,2,0) (2,3,1) (2,4,1)

P3
(0,0,1) (0,0,2)

Time

B. Prabhakaran
Causal Ordering of Messages

Space

P1 Send(M1)

P2 Send(M2)

P3

Time

Send(M1)

(1)

(2)
Message Ordering …

- Not really worry about maintaining clocks.
- Order the messages sent and received among all processes in a distributed system.
- (e.g.,) Send(M1) -> Send(M2), M1 should be received ahead of M2 by all processes.
- This is not guaranteed by the communication network since M1 may be from P1 to P2 and M2 may be from P3 to P4.
- Message ordering:
  - Deliver a message only if the preceding one has already been delivered.
  - Otherwise, buffer it up.
BSS Algorithm

- BSS: Birman-Schiper-Stephenson Protocol
- Broadcast based: a message sent is received by all other processes.
- Deliver a message to a process only if the message preceding it immediately, has been delivered to the process.
- Otherwise, buffer the message.
- Accomplished by using a vector accompanying the message.
BSS Algorithm ...

1. Process Pi increments the vector time VTpi[i], time stamps, and broadcasts the message m. VTpi[i] - 1 denotes the number of messages preceding m.

2. Pj != Pi receives m. m is delivered when:
   a. VTpj[i] == VTm[i] - 1
   b. VTpj[k] >= VTm[k] for all k in {1,2,...,n} - {i}, n is the total number of processes. Delayed message are queued in a sorted manner.
   c. Concurrent messages are ordered by time of receipt.

3. When m is delivered at Pj, VTpj updated according Rule 2 of vector clocks.
   2(a) : Pj has received all Pi’s messages preceding m.
   2(b): Pj has received all other messages received by Pi before sending m.
BSS Algorithm …

P1
(buffer) (0,0,1) (0,1,1)

P2 (0,0,1) (0,1,1)

P3 (0,0,1) (0,1,1)

deliver from buffer
SES Algorithm

- SES: Schiper-Eggli-Sandoz Algorithm. No need for broadcast messages.
- Each process maintains a vector $V_P$ of size $N - 1$, $N$ the number of processes in the system.
- $V_P$ is a vector of tuple $(P', t)$: $P'$ the destination process id and $t$, a vector timestamp.
- $Tm$: logical time of sending message $m$
- $Tpi$: present logical time at $pi$
- Initially, $V_P$ is empty.
SES Algorithm

- **Sending a Message:**
  - Send message M, time stamped tm, along with V_P1 to P2.
  - Insert (P2, tm) into V_P1. Overwrite the previous value of (P2,t), if any.
  - (P2,tm) is not sent. Any future message carrying (P2,tm) in V_P1 cannot be delivered to P2 until tm < tP2.

- **Delivering a message**
  - If V_M (in the message) does not contain any pair (P2, t), it can be delivered.
  - /* (P2, t) exists */ If t > Tp2, buffer the message. (Don’t deliver).
  - else (t < Tp2) deliver it
What does the condition $t > T_{p2}$ imply?
- $t$ is message vector time stamp.
- $t > T_{p2} \rightarrow$ For all $j$, $t[j] > T_{p2}[j]$
- This implies some events occurred without P2’s knowledge in other processes. So P2 decides to buffer the message.

When $t < T_{p2}$, message is delivered & $T_{p2}$ is updated with the help of $V_{P2}$ (after the merge operation).
SES Buffering Example

P1
-------------------------
Tp1: (1,1,0) (2,2,2)

P2
-------------------------
Tp2: (0,1,0) (0,2,0)

M1
V_P2: empty

P3
-------------------------
Tp3: (0,2,1) (0,2,2)

M2
V_P2: (P1, <0,1,0>)

M3
V_P3: (P1, <0,1,0>)
SES Buffering Example...

- M1 from P2 to P1: M1 + Tm (=<0,1,0>) + Empty V_P2
- M2 from P2 to P3: M2 + Tm (<0, 2, 0>) + (P1, <0,1,0>)
- M3 from P3 to P1: M3 + <0,2,2> + (P1, <0,1,0>)
- M3 gets buffered because:
  - Tp1 is <0,0,0>, t in (P1, t) is <0,1,0> & so Tp1 < t
- When M1 is received by P1:
  - Tp1 becomes <1,1,0>, by rules 1 and 2 of vector clock.
- After updating Tp1, P1 checks buffered M3.
  - Now, Tp1 > t [in (P1, <0,1,0>].
  - So M3 is delivered.
SES Algorithm ...

- On delivering the message:
  - Merge $V_M$ (in message) with $V_{P2}$ as follows.
    - If $(P,t)$ is not there in $V_{P2}$, merge.
    - If $(P,t)$ is present in $V_{P2}$, $t$ is updated with $\max(t \text{ in } V_m, t \text{ in } V_{P2})$.
  - Message cannot be delivered until $t$ in $V_M$ is greater than $t$ in $V_{P2}$
  - Update site P2’s local, logical clock.
  - Check buffered messages after local, logical clock update.
SES Algorithm …

P1

(1,2,1) (2,2,1)

P2

(0,1,1) M2

(0,2,1) V_P2 is empty

P3

M1

(0,0,1)

V_P3 is empty

(0,2,2)
Handling Multicasts

- Each node can maintain $n \times n$ matrix $M$, $n$ being the number of processes.
- Node $i$ multicasts to $j$ and $k$: increments $M[i,j]$ and $M[i,k]$. $M$ sent along with the message.
- When node $j$ receives message $m$ from $i$, it can be delivered if and only if:
  - $M[j,i] = M[m,i] - 1$
  - $M[j,k] \geq M[m,k]$ for all $k \neq i$.
- Else buffer the message
- On message delivery: $M[j,x,y] = \max(M[j,x,y], M[m,x,y])$
Handling Multicasts: Example

P1: 000 000 000 000
     000 000 101 000
     000 110 110

P2: 000 000 000 110
     000 101 110

P3: 000 000 000 000
     000 101 110

M1: 000 000 000 000
     000 110 110

M2: 000 000 000 000
     000 101 110
Global State

Global State 1

$500
A
C1: Empty

$200
B
C2: Empty

Global State 2

$450
A
C1: Tx $50

$200
B
C2: Empty

Global State 3

$450
A
C1: Empty

$250
B
C2: Empty
(e.g.,) Global state of A is recorded in (1) and not in (2).
- State of B, C1, and C2 are recorded in (2)
- Extra amount of $50 will appear in global state
- Reason: A’s state recorded before sending message and C1’s state after sending message.

Inconsistent global state if $n < n'$, where
- $n$ is number of messages sent by A along channel before A’s state was recorded
- $n'$ is number of messages sent by A along the channel before channel’s state was recorded.

Consistent global state: $n = n'$
Recording Global State...

- Similarly, for consistency \( m = m' \)
  - \( m' \): no. of messages received along channel before B’s state recording
  - \( m \): no. of messages received along channel by B before channel’s state was recorded.

- Also, \( n' \geq m \), as in no system no. of messages sent along the channel be less than that received

- Hence, \( n \geq m \)

- Consistent global state should satisfy the above equation.
- Consistent global state:
  - Channel state: sequence of messages sent before recording sender’s state, excluding the messages received before receiver’s state was recorded.
  - Only transit messages are recorded in the channel state.
Recording Global State

- **Send(Mij)**: message M sent from Si to Sj
- **rec(Mij)**: message M received by Sj, from Si
- **time(x)**: Time of event x
- **LSi**: local state at Si
- **send(Mij)** is in LSi iff (if and only if) time(send(Mij)) < time(LSi)
- **rec(Mij)** is in LSj iff time(rec(Mij)) < time(LSj)
- **transit(LSi, LSj)**: set of messages sent/recorded at LSi and NOT received/recorded at LSj
Recording Global State …

- inconsistent(\(LS_i, LS_j\)): set of messages NOT sent/recorded at \(LS_i\) and received/recorded at \(LS_j\)
- Global State, GS: \{LS_1, LS_2, \ldots, LS_n\}
- Consistent Global State, GS = \{LS_1, \ldots, LS_n\} AND for all \(i\) in \(n\), inconsistent(LSi,LSj) is null.
- Transitless global state, GS = \{LS_1, \ldots, LS_n\} AND for all \(i\) in \(n\), transit(LSi,LSj) is null.
Recording Global State..

M1: transit
M2: inconsistent
Recording Global State...

- Strongly consistent global state: consistent and transitless, i.e., all send and the corresponding receive events are recorded in all LSi.
Chandy-Lamport Algorithm

- Distributed algorithm to capture a consistent global state. Communication channels assumed to be FIFO.
- Uses a *marker* to initiate the algorithm. Marker sort of dummy message, with no effect on the functions of processes.
- Sending Marker by P:
  - P records its state.
  - For each outgoing channel C, P sends a marker on C before P sends further messages along C.
- Receiving Marker by Q:
  - If Q has NOT recorded its state: (a). Record the state as an empty sequence. (b) SEND marker (use above rule).
  - Else (Q has recorded state before): Record the state of C as sequence of messages received along C, after Q’s state was recorded and before Q received the marker.
- FIFO channel condition + markers help in satisfying consistency condition.
Chandy-Lamport Algorithm

- Initiation of marker can be done by any process, with its own unique marker: <process id, sequence number>.
- Several processes can initiate state recording by sending markers. Concurrent sending of markers allowed.
- One possible way to collect global state: all processes send the recorded state information to the initiator of marker. Initiator process can sum up the global state.
Chandy-Lamport Algorithm ...

- Example:

Channel state example: M1 sent to Px at t1, M2 sent to Py at t2, ...
Cuts

- Cuts: graphical representation of a global state.
- Cut \( C = \{c_1, c_2, .., c_n\} \); \( c_i \): cut event at \( S_i \).
- Consistent Cut: If every message received by a \( S_i \) before a cut event, was sent before the cut event at Sender.
- One can prove: A cut is a consistent cut iff no two cut events are causally related, i.e., \(! (c_i \rightarrow c_j)\) and \(! (c_j \rightarrow c_i)\).
Time of a Cut

- C = \{c_1, c_2, .., c_n\} with vector time stamp VT_{ci}. Vector time of the cut, VT_c = \text{sup}(VT_{c_1}, VT_{c_2}, .., VT_{c_n}).

- \text{sup} is a component-wise maximum, i.e., VT_{ci} = \text{max}(VT_{c_1}[i], VT_{c_2}[i], .., VT_{c_n}[i]).

- Now, a cut is consistent iff VT_c = (VT_{c_1}[1], VT_{c_2}[2], .., VT_{c_n}[n]).
Termination Detection

- Termination: completion of the sequence of algorithm. (e.g.,) leader election, deadlock detection, deadlock resolution.

- Use a *controlling agent* or a *monitor process*.

- Initially, all processes are idle. Weight of controlling agent is 1 (0 for others).

- Start of computation: message from controller to a process. Weight: split into half (0.5 each).

- Repeat this: any time a process sends a computation message to another process, split the weights between the two processes (e.g., 0.25 each for the third time).

- End of computation: process sends its weight to the controller. Add this weight to that of controller’s. (Sending process’s weight becomes 0).

- *Rule:* Sum of W always 1.

- *Termination:* When weight of controller becomes 1 again.
Huang’s Algorithm

- $B(DW)$: computation message, $DW$ is the weight.
- $C(DW)$: control/end of computation message;
- Rule 1: Before sending $B$, compute $W_1$, $W_2$ (such that $W_1 + W_2$ is $W$ of the process). Send $B(W_2)$ to $P_i$, $W = W_1$.
- Rule 2: Receiving $B(DW)$ $\Rightarrow$ $W = W + DW$, process becomes active.
- Rule 3: Active to Idle $\Rightarrow$ send $C(DW)$, $W = 0$.
- Rule 4: Receiving $C(DW)$ by controlling agent $\Rightarrow$ $W = W + DW$, If $W == 1$, computation has terminated.