Time, clocks and the ordering of events

Amir H. Payberah
amir@sics.se

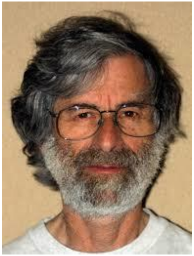
Amirkabir University of Technology
(Tehran Polytechnic)

What is a Distributed System?

Distributed System

A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable.

- Leslie Lamport



What is a Distributed System?

- A set of nodes, connected by a network, which appear to its users as a single coherent system.

Distributed Algorithms

Two Generals' Problem (1/3)

- Two generals need to coordinate an attack.

Two Generals' Problem (1/3)

- Two generals need to coordinate an attack.
  - Must agree on time to attack.

Two Generals' Problem (1/3)

- Two generals need to coordinate an attack.
  - Must agree on time to attack.
  - They will win only if they attack simultaneously.

Two Generals' Problem (1/3)

- Two generals need to coordinate an attack.
  - Must agree on time to attack.
  - They will win only if they attack simultaneously.
  - Communicate through messengers.

Two Generals' Problem (1/3)

- Two generals need to coordinate an attack.
  - Must agree on time to attack.
  - They will win only if they attack simultaneously.
  - Communicate through messengers.
  - Messengers may be killed on their way.

Two Generals' Problem (2/3)

- Lets try to solve it for general g1 and g2.

Two Generals' Problem (2/3)

- Lets try to solve it for general g1 and g2.

- g1 sends time of attack to g2.
  - Problem: how to ensure g2 received message?

Two Generals' Problem (2/3)

- Lets try to solve it for general g1 and g2.

- g1 sends time of attack to g2.
  - Problem: how to ensure g2 received message?
  - Solution: let g2 ack receipt of message.

Two Generals' Problem (2/3)

- Lets try to solve it for general g1 and g2.

- g1 sends time of attack to g2.
  - Problem: how to ensure g2 received message?
  - Solution: let g2 ack receipt of message.
  - Problem: how to ensure g1 received ack?

Two Generals' Problem (2/3)

- Lets try to solve it for general g1 and g2.

- g1 sends time of attack to g2.
  - Problem: how to ensure g2 received message?
  - Solution: let g2 ack receipt of message.
  - Problem: how to ensure g1 received ack?
  - Solution: let g1 ack the receipt of the ack.

Two Generals' Problem (2/3)

- Lets try to solve it for general g1 and g2.

- g1 sends time of attack to g2.
  - Problem: how to ensure g2 received message?
  - Solution: let g2 ack receipt of message.
  - Problem: how to ensure g1 received ack?
  - Solution: let g1 ack the receipt of the ack.
  - ...

Two Generals' Problem (2/3)

- Lets try to solve it for general g1 and g2.

- g1 sends time of attack to g2.
  - Problem: how to ensure g2 received message?
  - Solution: let g2 ack receipt of message.
  - Problem: how to ensure g1 received ack?
  - Solution: let g1 ack the receipt of the ack.
  - ...

- This problem is impossible to solve!

Two Generals' Problem (3/3)

- Applicability to distributed systems:

Two Generals' Problem (3/3)

- Applicability to distributed systems:
  - Two nodes need to agree on a value.

Two Generals' Problem (3/3)

- Applicability to distributed systems:
  - Two nodes need to agree on a value.
  - Communicate by messages using an unreliable channel.

Two Generals' Problem (3/3)

- Applicability to distributed systems:
  - Two nodes need to agree on a value.
  - Communicate by messages using an unreliable channel.

- Agreement is a core problem.

Distributed Algorithms

- Algorithms that are supposed to work in distributed networks or on multiprocessors.

Distributed Algorithms

- Algorithms that are supposed to work in distributed networks or on multiprocessors.

- Accomplish tasks like:
  - Data management
  - Resource management
  - Consensus
  - ...

Distributed Algorithms

- Algorithms that are supposed to work in distributed networks or on multiprocessors.

- Accomplish tasks like:
  - Data management
  - Resource management
  - Consensus
  - ...

- Must work in difficult settings:
  - Concurrency: uncertainty of timing, order of events and inputs.
  - Fault-tolerance: failure and recovery of machines/processors, of communication channels.

Correctness of Distributed Algorithms

Correctness

- Always expressed in terms of
  - Safety and Liveness
  - B. Alpern and F.B. Schneider, Defining Liveness, Technical Report, 1985

Correctness

- Always expressed in terms of
  - Safety and Liveness
  - B. Alpern and F.B. Schneider, Defining Liveness, Technical Report, 1985

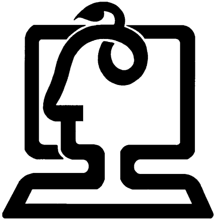
- Safety
  - Properties that state that nothing bad ever happens.

Correctness

- Always expressed in terms of
  - Safety and Liveness
  - B. Alpern and F.B. Schneider, Defining Liveness, Technical Report, 1985
- Safety
  - Properties that state that nothing bad ever happens.
- Liveness
  - Properties that state that something good eventually happens.

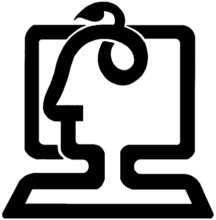
Correctness Example (1/3)

- Correctness of you in this course :)



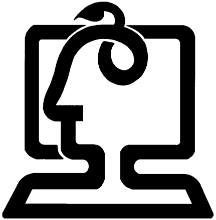
Correctness Example (1/3)

- Correctness of you in this course :)
- You should never fail the exam:



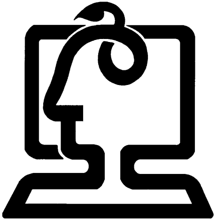
Correctness Example (1/3)

- Correctness of you in this course :)
- You should never fail the exam: Safety



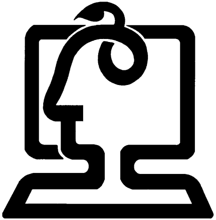
Correctness Example (1/3)

- Correctness of you in this course :)
- You should never fail the exam: Safety
- You should eventually take the exam:



Correctness Example (1/3)

- Correctness of you in this course :)
- You should never fail the exam: Safety
- You should eventually take the exam: Liveness



Correctness Example (2/3)

- Correctness of traffic lights at intersection.

Correctness Example (2/3)

- Correctness of traffic lights at intersection.
- Only one direction should have a green light:

Correctness Example (2/3)

- Correctness of traffic lights at intersection.
- Only one direction should have a green light: Safety

Correctness Example (2/3)

- Correctness of traffic lights at intersection.
- Only one direction should have a green light: Safety
- Every direction should eventually get a green light:

Correctness Example (2/3)

- Correctness of traffic lights at intersection.
- Only one direction should have a green light: Safety
- Every direction should eventually get a green light: Liveness

Correctness Example (3/3)

- Correctness of point-to-point message communication.

Correctness Example (3/3)

- Correctness of point-to-point message communication.
- A message sent is delivered at most once:

Correctness Example (3/3)

- Correctness of point-to-point message communication.
- A message sent is delivered at most once: Safety

Correctness Example (3/3)

- Correctness of point-to-point message communication.
- A message sent is delivered at most once: Safety
- A message sent is delivered at least once:

Correctness Example (3/3)

- Correctness of point-to-point message communication.
- A message sent is delivered at most once: Safety
- A message sent is delivered at least once: Liveness

More on Safety and Liveness

- Safety
  - Often involves the word never, at most, cannot, ...

More on Safety and Liveness

- Safety
  - Often involves the word never, at most, cannot, ...
- Liveness
  - Often involves the word eventually: some point in future
  - Liveness is often just termination

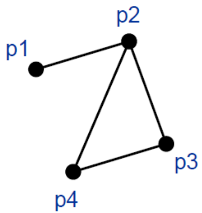
Modeling
Distributed Systems

What is a Model?

- Abstraction of relevant system properties.
- Real world is complex, model simplifies it.
- Help solving problems.
- Help analyze problems/solutions.

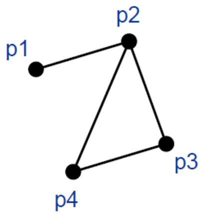
Modeling Distributed Systems

- What is a distributed system?
  - Bunch of nodes/processes
  - Sending messages over a network
  - To solve a common goal (algorithm)



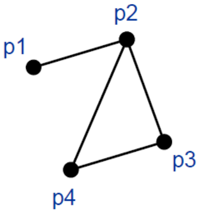
Modeling Distributed Systems

- What is a distributed system?
  - Bunch of nodes/processes
  - Sending messages over a network
  - To solve a common goal (algorithm)
- How do we model this?



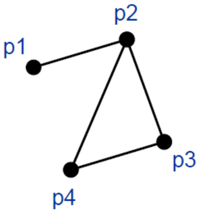
Modeling a Node

- A single node has a bunch of neighbors.



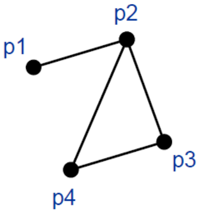
Modeling a Node

- A single node has a bunch of neighbors.
- Can send and receive messages.



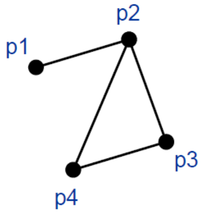
Modeling a Node

- A single node has a bunch of neighbors.
- Can send and receive messages.
- Can do local computations.



Modeling a Node

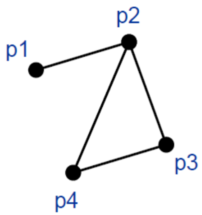
- A single node has a bunch of neighbors.
- Can send and receive messages.
- Can do local computations.



- Like a state machine: a node is in only one state at a time.

Modeling a Node

- A single node has a bunch of neighbors.
- Can send and receive messages.
- Can do local computations.



- Like a state machine: a node is in only one state at a time.
- The state of a node: input buffer, output buffer, and other data relevant to algorithm

Single Node Perspective

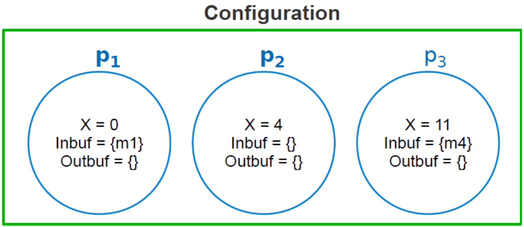
- This is how computers in a distributed system work:

1. Wait for message.
2. When received message, do some local computation, send some messages.
3. Goto 1

Single Node to a Distributed System (1/3)

- A configuration is a snapshot of state of all nodes.
  - $C = (s_0, s_1, \cdots, s_{n-1})$ where s_i is state of process p_i.

- An initial configuration is a configuration where each s_i is an initial state.



Single Node to a Distributed System (2/3)

- The system evolves through events:
  - Computation event at node i: $comp(i)$
  - Delivery event of msg m from i to j: $del(i, j, m)$

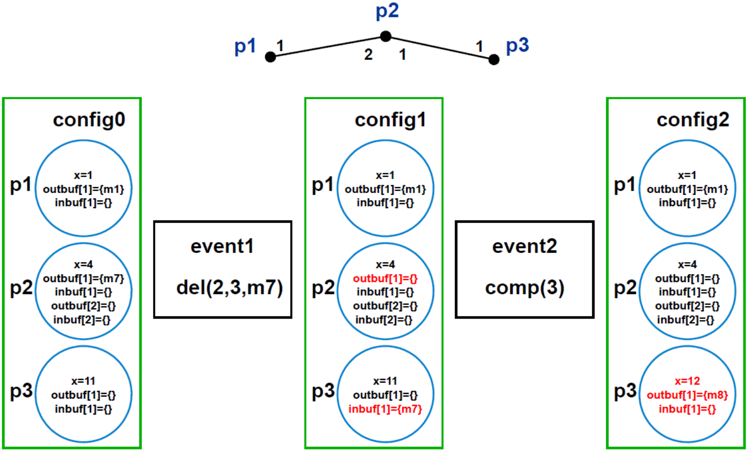
Single Node to a Distributed System (2/3)

- The system evolves through events:
  - Computation event at node i: $comp(i)$
  - Delivery event of msg m from i to j: $del(i, j, m)$

- An execution is an infinite sequence of
  - $config_0$, $event_1$, $config_1$, $event_2$, $config_2$, ...
  - $config_0$ is an initial configuration.
  - $event$ could be $comp$ or del.

Single Node to a Distributed System (3/3)

- Example execution



Synchronous Systems (1/2)

- An execution partitioned into non-overlapping rounds.

Synchronous Systems (1/2)

- An execution partitioned into non-overlapping rounds.

- Informally, in each round:
  - Every process can send a message to each neighbor.
  - All messages are delivered.
  - Every process computes based on message received.

Synchronous Systems (1/2)

- An execution partitioned into non-overlapping rounds.
- Informally, in each round:
  - Every process can send a message to each neighbor.
  - All messages are delivered.
  - Every process computes based on message received.
- Formally, rounds consist of:
  - Deliver event for every message in all outbufs.
  - One computation event on every process.

Synchronous Systems (2/2)

- Time variance is bounded.
- Execution: bounded execution speed and time.
- Communication: bounded transmission delay.
- Clocks: bounded clock drift (and differences in clocks).

Asynchronous Systems

- Time variance is not bounded.
- Execution: different steps can have varying duration.
- Communication: transmission delays vary widely.
- Clocks: arbitrary clock drift.

Time, Clock, and Order of Events

Time

- Global time
  - Astronomical time (based on earth's rotation)
  - International Atomic Time (IAT)
  - Coordinated Universal Time (UTC)

- Local time
  - Not synchronized to a global source

Computer Clocks

- Computer clocks
  - Crystal oscillates at known frequency
  - Oscillations cause timer interrupts
  - Timer interrupts update clock

- Clock skew
  - Crystals in different computers run at slightly different rates
  - Clocks get out of sync
  - Skew: instantaneous difference
  - Drift: rate of change of skew

Synchronizing Computer Clocks

- Internal synchronization
  - Clocks synchronize locally
  - Only synchronized with each other
  - Berkeley algorithm

- Time server
  - Server that has the correct time
  - Server that calculates the correct time
  - Network Time Protocol (NTP)

Logical Clock

- Event ordering is more important than physical time.

Logical Clock

- Event ordering is more important than physical time.
- Events (e.g., state changes) in a single process are ordered.

Logical Clock

- Event ordering is more important than physical time.
- Events (e.g., state changes) in a single process are ordered.
- Processes need to agree on ordering of causally related events (e.g., message send and receive).

Causal Order

- $\rightarrow$: causal order relation.
  - Also called Leslie Lamport's happened before relation.

Causal Order

- $\rightarrow$: causal order relation.
  - Also called Leslie Lamport's happened before relation.

- The relation $\rightarrow$ on the events of an execution is defined as follows:
  - If a occurs before b on the same process, then $a \rightarrow b$.
  - If a produces (comp) m and b delivers m, then $a \rightarrow b$.
  - If a delivers m and b consumes (comp) m, then $a \rightarrow b$.

Causal Order

- $\rightarrow$: causal order relation.
  - Also called Leslie Lamport's happened before relation.

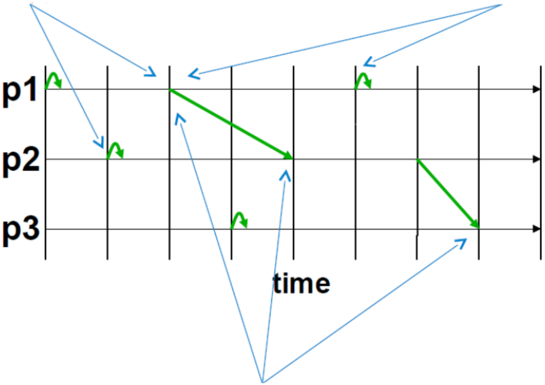
- The relation $\rightarrow$ on the events of an execution is defined as follows:
  - If a occurs before b on the same process, then $a \rightarrow b$.
  - If a produces (comp) m and b delivers m, then $a \rightarrow b$.
  - If a delivers m and b consumes (comp) m, then $a \rightarrow b$.

- Transitivity: if $a \rightarrow b$ and $b \rightarrow c$, then $a \rightarrow c$.

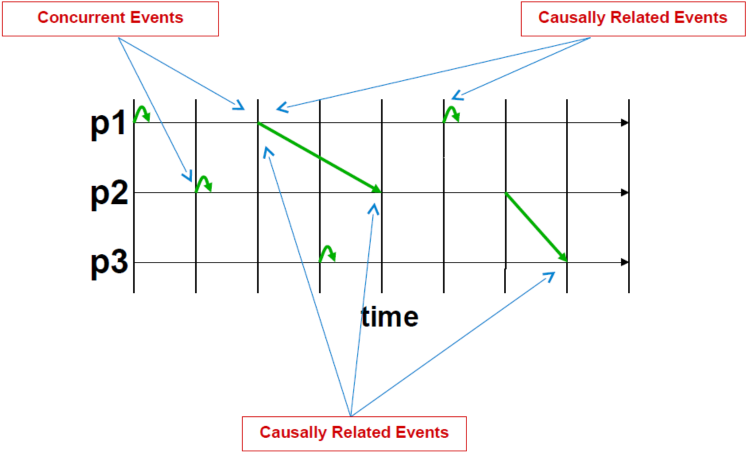
Causal Order

- $\rightarrow$: causal order relation.
  - Also called Leslie Lamport's happened before relation.
- The relation $\rightarrow$ on the events of an execution is defined as follows:
  - If a occurs before b on the same process, then $a \rightarrow b$.
  - If a produces (comp) m and b delivers m, then $a \rightarrow b$.
  - If a delivers m and b consumes (comp) m, then $a \rightarrow b$.
- Transitivity: if $a \rightarrow b$ and $b \rightarrow c$, then $a \rightarrow c$.
- concurrent events ($a||b$): if $a \not\rightarrow b$ and $b \not\rightarrow a$.

Causal Order Example



Causal Order Example



Equivalence of Executions

- If two executions F and E have the same collection of events, and their causal order is preserved, F and E are said to be similar executions.

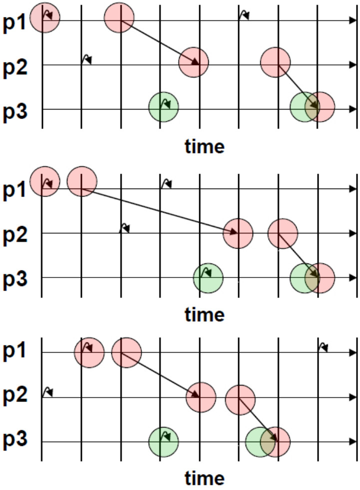
Equivalence of Executions

- If two executions F and E have the same collection of events, and their causal order is preserved, F and E are said to be similar executions.

- F and E could have different permutation of events as long as causality is preserved.

Example of Similar Executions

- Same color $\sim$ Causally related



- So causality is all that matters ...

- So causality is all that matters ...
- ... how to locally tell if two events are causally related?

Lamport Clocks

- Each process p_i has a local logical clock t_i.
  - Initially $t_i = 0$

Lamport Clocks

- Each process p_i has a local logical clock t_i.
  - Initially $t_i = 0$

- Before timestamping a local event p_i executes $t_i := t_i + 1$.

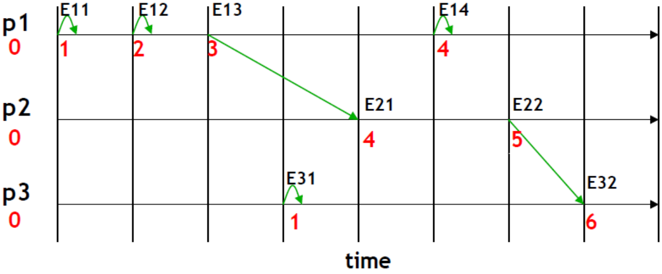
Lamport Clocks

- Each process p_i has a local logical clock t_i.
  - Initially $t_i = 0$

- Before timestamping a local event p_i executes $t_i := t_i + 1$.

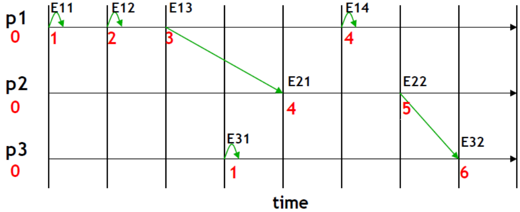
- Whenever a message m is sent from p_i to p_j:
  - p_i executes $t_i := t_i + 1$ and sends t_i with m.
  - p_j receives t_i with m and executes $t_j := max(t_j, t_i) + 1$.

Example of Lamport Logical Clock



Lamport Clocks Properties

- $a \rightarrow b$ implies $t(a) < t(b)$, where $t(a)$ is Lamport clock of event a.



- $t(a) < t(b)$ does not necessarily imply $a \rightarrow b$
- $t(E31) < t(E13)$, but $E31 \not\rightarrow E13$

Shortcoming of Lamport Clocks

- Main shortcoming of Lamport's clocks:
  - $t(a) < t(b)$ does not necessarily imply $a \rightarrow b$
  - We cannot deduce causal dependencies from time stamps.

- Why?
  - Clocks advance independently or via messages.
  - There is no history as to where advances come from.

Vector Clocks

- At each process, maintain a clock for every other process.
  - Each clock V_i is a vector of size N.
  - $V_i[j]$ contains process p_is knowledge about process p_j's clock.
  - Initially, $V_i[j] := 0$ for $i, j \in \{1, \cdots, N\}$.

Vector Clocks

- At each process, maintain a clock for every other process.
  - Each clock V_i is a vector of size N.
  - $V_i[j]$ contains process p_is knowledge about process p_j's clock.
  - Initially, $V_i[j] := 0$ for $i, j \in \{1, \cdots, N\}$.

- Before pi timestamps an event: $V_i[i] := V_i[i] + 1$.

Vector Clocks

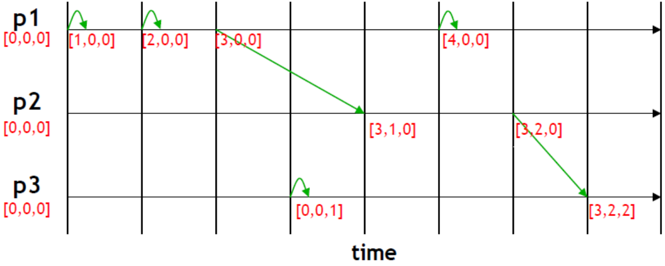
- At each process, maintain a clock for every other process.
  - Each clock V_i is a vector of size N.
  - $V_i[j]$ contains process p_is knowledge about process p_j's clock.
  - Initially, $V_i[j] := 0$ for $i, j \in \{1, \cdots, N\}$.

- Before pi timestamps an event: $V_i[i] := V_i[i] + 1$.

- Whenever a message m is sent from p_i to p_j:
  - p_i executes $V_i[i] := V_i[i] + 1$ and sends V_i with m.
  - p_j receives V_i with m and merges the vector clocks V_i and V_j:

$$V_j[k] = \begin{cases} max(V_j[k], V_i[k]) + 1 & \text{if } j = k \\ max(V_j[k], V_i[k]) & \text{otherwise} \end{cases}$$

Example of Vector Clock

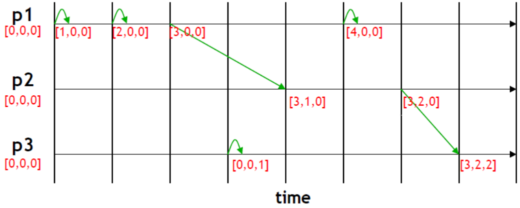


Comparing Vector Clocks

- $V = V'$ iff $V[i] = V'[i]$ for $i \in \{1, \cdots, N\}$
- $V \leq V'$ iff $V[i] \leq V'[i]$ for $i \in \{1, \cdots, N\}$
- $V||V'$ iff $V \leq V' \wedge V' \leq V$

Vector Clocks Properties

- $a \to b$ implies $V(a) < V(b)$
- $V(a) < V(b)$ implies $a \to b$



Logical Clock vs. Vector Clock

- Logical clock
  - If $a \rightarrow b$ then $t(a) < t(b)$
- Vector clock
  - If $a \rightarrow b$ then $V(a) < V(b)$
  - If $V(a) < V(b)$ then $a \rightarrow b$
  - Sending extra information: vector with size N, for N processes.

Global State

Global State

- Determining global properties
- Distributed checkpoint
  - What is a correct state of the system to save?
- Distributed garbage collection
  - Do any references exist to a given object?
- ...

Local History

- N processes $p_i, i \in \{1, \cdots, N\}$

Local History

- N processes $p_i, i \in \{1, \cdots, N\}$
- The local history of process p_i is a sequence of events $h_i = \langle e_i^0, e_i^1, \cdots \rangle$

Local History

- N processes $p_i, i \in \{1, \cdots, N\}$
- The local history of process p_i is a sequence of events $h_i = \langle e_i^0, e_i^1, \cdots \rangle$
- May be finite or infinite.

Local History

- N processes $p_i, i \in \{1, \cdots, N\}$

- The local history of process p_i is a sequence of events $h_i = \langle e_i^0, e_i^1, \cdots \rangle$

- May be finite or infinite.

- Each event e_i^j is either a local event or a communication event.

Local and Global State

- s_i^k denotes the local state of process p_i after execution of event e_i^k.

Local and Global State

- s_i^k denotes the local state of process p_i after execution of event e_i^k.
- The local state s_i^k records all events included in the history h_i^k.

Local and Global State

- s_i^k denotes the local state of process p_i after execution of event e_i^k.
- The local state s_i^k records all events included in the history h_i^k.
- The global state, S, of a distributed computation is an N-tuple of local states $(s_1, s_2, \cdots, s_N)$, one for each process.

Cuts

- $h_i^{c_i}$ is history of p_i up to and including event $e_i^{c_i}$, called partial history.

Cuts

- $h_i^{c_i}$ is history of p_i up to and including event $e_i^{c_i}$, called partial history.

- A cut, C, of a distributed computation is the union of N partial histories, one for each process: $C = \bigcup_{i=1}^{N} h_i^{c_i}$

Cuts

- $h_i^{c_i}$ is history of p_i up to and including event $e_i^{c_i}$, called partial history.

- A cut, C, of a distributed computation is the union of N partial histories, one for each process: $C = \bigcup_{i=1}^{N} h_i^{c_i}$

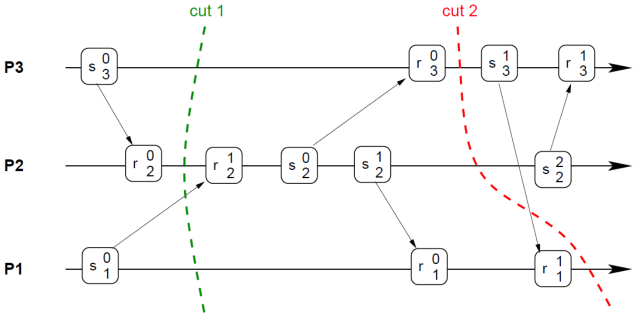
- Each cut C has a corresponding global state: $S = (s_1, s_2, \cdots, s_N)$.

Consistent Cuts (1/2)

- A cut C is consistent, if for all events e and e'
  $(e' \in C) \wedge (e \rightarrow e') \Rightarrow e \in C$

Consistent Cuts (1/2)

- A cut C is consistent, if for all events e and e'
  $(e' \in C) \wedge (e \rightarrow e') \Rightarrow e \in C$



Consistent Cuts (2/2)

- It can be written as two following conditions:

Consistent Cuts (2/2)

- It can be written as two following conditions:
  - Condition 1: any message that is sent by a process before recording its snapshot, must be recorded in the cut.

Consistent Cuts (2/2)

- It can be written as two following conditions:
  - Condition 1: any message that is sent by a process before recording its snapshot, must be recorded in the cut.
  - Condition 2: any message that is sent by a process after recording its snapshot, must not be recorded in the cut.

Consistent Cuts (2/2)

- It can be written as two following conditions:
  - Condition 1: any message that is sent by a process before recording its snapshot, must be recorded in the cut.
  - Condition 2: any message that is sent by a process after recording its snapshot, must not be recorded in the cut.

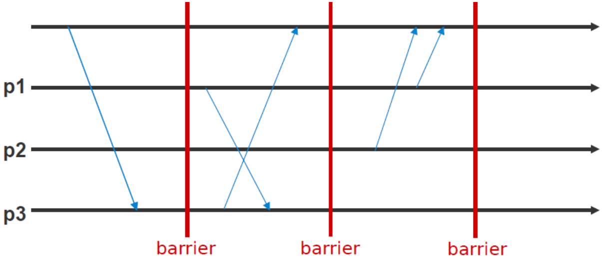
- A global state is consistent if it corresponds to a consistent cut.

Possible Solutions

- Coordinated blocking
- Coordinated non-blocking

Coordinated Blocking

- At barrier, all processes take their checkpoints.
- Bulk-Synchronous Parallel (BSP)



Coordinated Non-Blocking

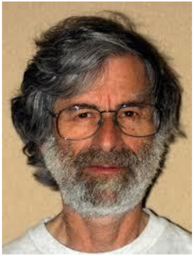
- Processes must be coordinated, but do we really need to block?

Coordinated Non-Blocking

- Processes must be coordinated, but do we really need to block?
- Chandy and Lamport's snapshot

Chandy and Lamport's Snapshot

- Determines a consistent global state.
- Takes care of messages that are in transit.

Model of a Distributed System

- Process: one process initiates taking of global snapshot
- Channels: directed, FIFO, reliable
- Process graph: fixed topology, strongly connected component

Chandy and Lamport's Snapshot Algorithm (1/3)

- The algorithm can be initiated by any process by executing the Marker Sending Rule, by which it records its local state and sends a marker on each outgoing channel.

Chandy and Lamport's Snapshot Algorithm (1/3)

- The algorithm can be initiated by any process by executing the Marker Sending Rule, by which it records its local state and sends a marker on each outgoing channel.

- A process executes the Marker Receiving Rule on receiving a marker. If the process has not yet recorded its local state, it records the state of the channel on which the marker is received as empty and executes the Marker Sending Rule to record its local state.

Chandy and Lamport's Snapshot Algorithm (1/3)

- The algorithm can be initiated by any process by executing the Marker Sending Rule, by which it records its local state and sends a marker on each outgoing channel.

- A process executes the Marker Receiving Rule on receiving a marker. If the process has not yet recorded its local state, it records the state of the channel on which the marker is received as empty and executes the Marker Sending Rule to record its local state.

- The algorithm terminates after each process has received a marker on all of its incoming channels.

Chandy and Lamport's Snapshot Algorithm (1/3)

- The algorithm can be initiated by any process by executing the Marker Sending Rule, by which it records its local state and sends a marker on each outgoing channel.
- A process executes the Marker Receiving Rule on receiving a marker. If the process has not yet recorded its local state, it records the state of the channel on which the marker is received as empty and executes the Marker Sending Rule to record its local state.
- The algorithm terminates after each process has received a marker on all of its incoming channels.
- All the local snapshots get disseminated to all other processes and all the processes can determine the global state.

Chandy and Lamport's Snapshot Algorithm (2/3)

- Marker Sending Rule for process p_i:
  - Process p_i records its state.
  - For each outgoing channel c on which a marker has not been sent, p_i sends a marker along c before p_i sends further messages along c.

Chandy and Lamport's Snapshot Algorithm (3/3)

- Marker Receiving Rule for process p_j on receiving a marker along channel c:
  - If p_j has not recorded its state, it records the state of channel c as the empty set, and follows the Marker Sending Rule.
  - Otherwise, it records the state of c as the set of messages received along c after p_j's state was recorded and before p_j received the marker along c.

Correctness of the Algorithm (1/2)

- Condition 1: any message that is sent by a process before recording its snapshot, must be recorded in the cut.

- When a process p_j receives message m_{ij} from p_i that precedes the marker on channel c_{ij}, it acts as follows: if process p_j has not taken its snapshot yet, then it includes m_{ij} in its recorded snapshot. Otherwise, it records m_{ij} in the state of the channel c_{ij}. Thus, condition Condition 1 is satisfied.

Correctness of the Algorithm (2/2)

- Condition 2: any message that is sent by a process after recording its snapshot, must not be recorded in the cut.

- Due to FIFO property of channels, it follows that no message sent after the marker on that channel is recorded in the channel state. Thus, condition Condition 2 is satisfied.

Summary

Summary

- Correctness of distributed algorithms: safety + liveness
- Casual order
- Logical clock and Vector clock
- Global state: Chandy and Lamport's algorithm

References:

- L. Lamport, Time, clocks, and the ordering of events in a distributed system. ACM Communications, 1978
- K.M. Chandy and L. Lamport, Distributed snapshots: determining global states of distributed systems. ACM Transactions on Computer Systems, 1985.
- B. Alpern and F.B. Schneider, Defining Liveness. Technical Report, 1984.

Questions?

Acknowledgements

Some slides were derived from Seif Haridi slides (KTH University).