Formal Models of Distributed Systems

Seif Haridi – Royal Institute of Technology
Peter Van Roy – Université catholique de Louvain
haridi(at)kth.se
peter.vanroy(at)uclouvain.be

Formal Modeling

Models

- What is a model?
  - Abstraction of relevant system properties

- Why construct or learn a model?
  - Real world is complex, model simplifies

Modeling

- What can modeling do for us?
  - Help *solve* problems
    - Making algorithms
  - Help *analyze* problems/solutions
    - Analysis, proofs, simulations

- Very important skill

Modeling

- Different types of models:
  - Discrete event models
    - Often described by state transition systems: system evolves, moving from one state to another at discrete time steps
  - Continuous models
    - Often described by differential equations involving variables which can take real (continuous) values
- This course: *models of distributed computing (discrete)*

Granularity of Models

- Biggest challenge of modeling
  - Choosing the *right level of abstraction*!
- Model must be powerful enough to construct
  - *Impossibility proofs*
    - A statement about all possible algorithms in a system
- Our model should therefore be:
  - *Complete*: explain all relevant properties
  - *Correct*: behave as the system does (without error)
  - *Concise*: explain a class of distributed systems compactly

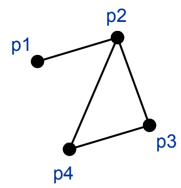
Model of Distributed Systems

Based on model from Attiya & Welch

2/20/12 7

Model of Distributed Computing

- 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
  - Can send and receive messages
  - Can do local computations
- Model node by state transition system (STS)
  - Like a finite state machine, except
    - Need not be finite
    - No input

State Transition System – Informal

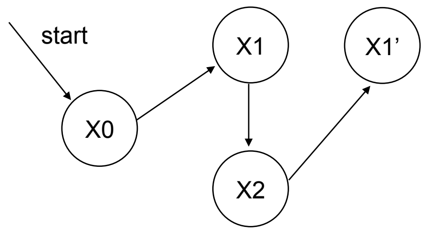
- A state transition system consists of
  - A set of states
  - Rule for which state to go to from each state (transition function/binary relation)
  - The set of starting states (initial states)

State Transition System – Example

- Example algorithm:

```
X:=0;
while (X<2) do
    X := X + 1;
endwhile
X:=1
```

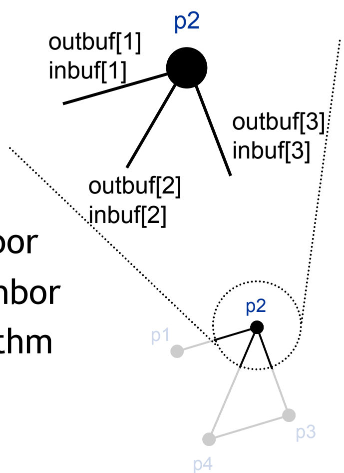
Using graphs:



- Formally:
  - States {X0, X1, X2, X1’}
  - Transition function {X0→X1, X1→X2, X2→X1’}
  - Initial states {X0}

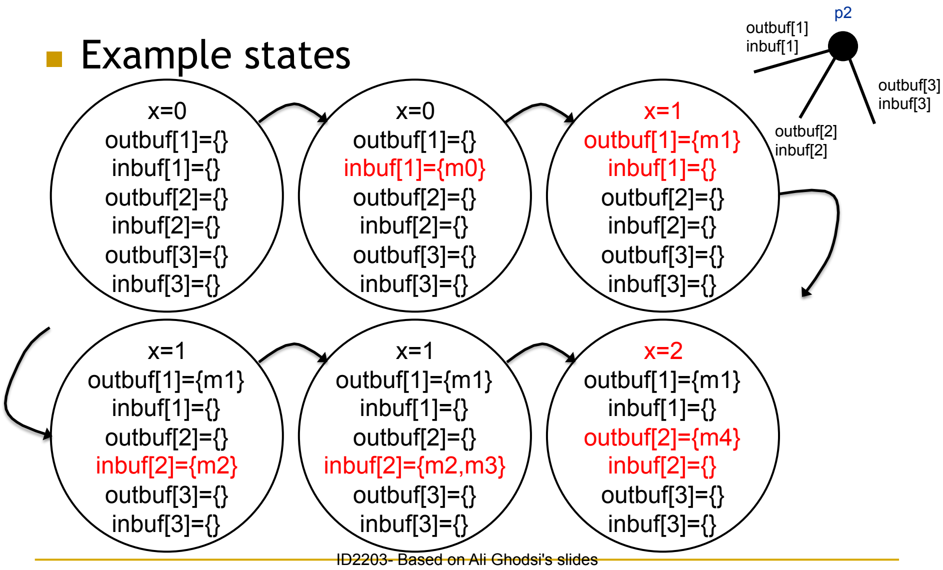
Modeling a Node

- State machine of node i
  - Set of states Q_i
- Each state consists of
  - 1 inbuffer set for each neighbor
  - 1 outbuffer set for each neighbor
  - Other data relevant to algorithm
- Initial states
  - inbuf[j] is empty for all j



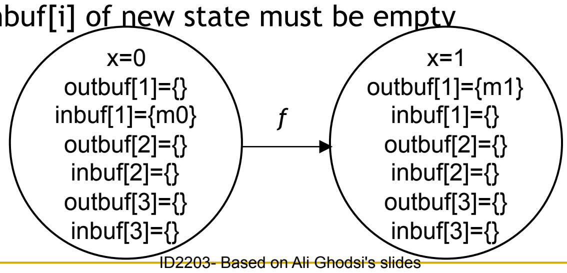
State of one node

- Example states



Transition functions

- All of the state except outbufs is called the accessible state of a node
  - when in outbuf, can't read it any more
- Transition function *f*
  - takes accessible state and gives full state, and
  - adds at most 1 new msg in each outbuf[i] of new state.
  - all inbuf[i] of new state must be empty



Transition functions formally

- Model in Attiya & Welch is a little bit broken
  - Output buffers are not read, yet they must be passed on
  - This is a tedious fix…
- State of a node (with k channels) is triple $<I,O,s>$
  - I is a vector of inbufs, $<I[1],…,I[k]>$
  - O is a vector of outbufs, $<O[1],…,O[k]>$
  - s is the local state
- We require that for any two
  - $f(<I_1,O_1,s_1>)=<I_2,O_2,s_2>$ and $f(<I_3,O_3,s_3>)=<I_4,O_4,s_4>$
    - $I_2=I_4=<\varnothing,…,\varnothing>$, i.e. all inbufs are empty, and
    - If $I_1=I_3$ and $s_1=s_3$ then
      - $s_2=s_4$, i.e. don't "observe" channel (1),
      - $O_1[i]\subseteq O_2[i]$ and $O_3[i]\subseteq O_4[i]$, i.e. only add messages to outbufs, and
      - $O_2[i]-O_1[i] = O_4[i]-O_3[i]$, i.e. don't "observe" channel (2)

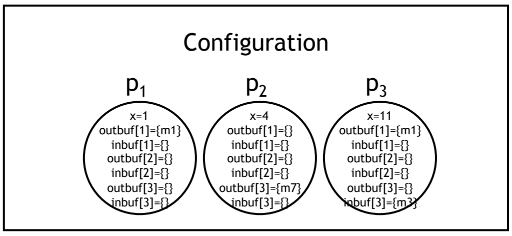
Single node perspective

- This is how computers in a distributed system work:
  - 1. Wait for message
  - 2. When received message, do some local computation, send some messages
  - Goto 1.
- Is this a correct model? [d]
  - Determinism?
  - I/O?
  - Atomicity?

Single Node to a Distributed System

A configuration is snapshot of state of all nodes

- $C = (q_0, q_1, \ldots, q_{n-1})$ where q_i is state of node p_i



- An initial configuration is a configuration where each q_i is an initial state

Single Node to a Distributed System

- 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)

- Computation event comp(i)
  - Apply transition function *f* on node i's state

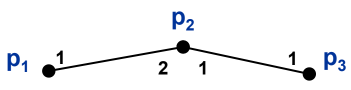
- Delivery event del(i,j,m)
  - Move message m from outbuf of p_i to inbuf of p_j

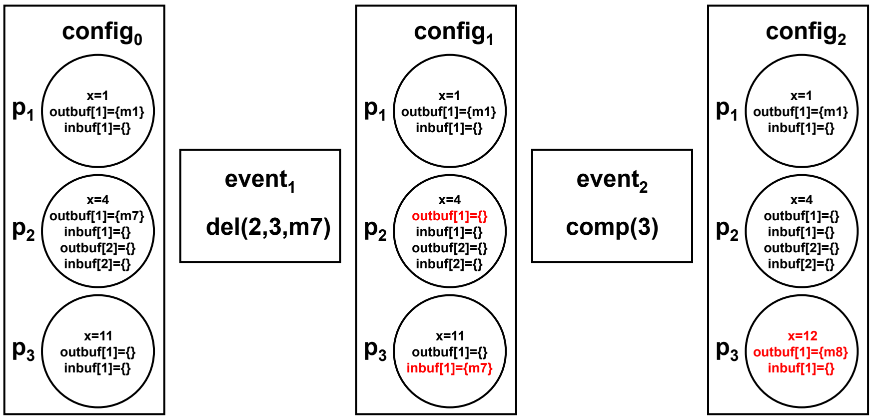
Execution

- An execution is an infinite sequence of
  - $config_0$, $event_1$, $config_1$, $event_2$, $config_2$…
  - $config_0$ is an initial configuration
- If $event_k$ is comp(i)
  - $config_{k-1}$ changes to $config_k$ by applying p_i's transition function on i's state in $config_{k-1}$
- If $event_k$ is del(i,j,m)
  - $config_{k-1}$ changes to $config_k$ by moving m from i's outbuf for link i-j to j's inbuf for link i-j

Single Node to a Distributed System

Example execution



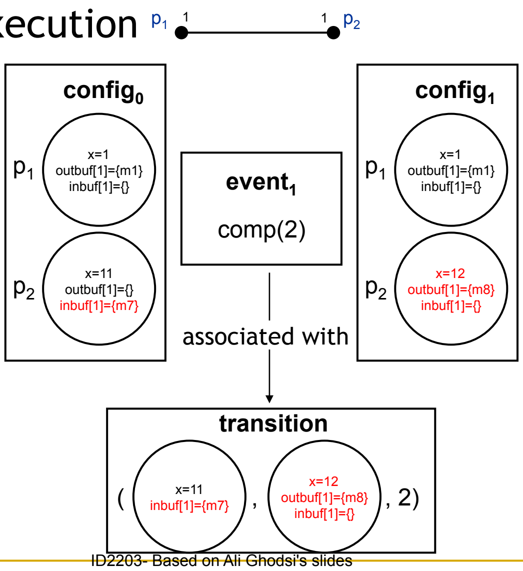


Some definitions for later use…

- Each comp(i) is associated with a transition
  - If *f* of process i maps $state_1$ to $state_2$: the triple $(state_1, state_2, i)$ is called a transition
- Transition (s_1,s_2,j) is applicable in configuration c if
  - The accessible state of node j is s_1 in c
- A del(i,j,m) is applicable in configuration c if
  - m is in outbuf for link i-j of node i in c

Single Node to a Distributed System

Example execution



Some definitions for later use… (2)

- If transition e=(s1,s2,i) is applicable to conf c
  - Then app(e,c) gives new configuration after the event comp(i)
- If e=del(i,j,m) is applicable to conf c
  - Then app(e,c) gives new configuration after the event del(i,j,m)

Schedules (Asynchronous Model)

- Our processes are deterministic
  - Given some message, update state, send some messages, and wait…
- Non-determinism comes from asynchrony
  - Messages take arbitrary time to be delivered
  - Processes execute at different speeds
- A schedule is the sequence of events
  - Message asynchrony determined by del(i,j,m)
  - Process speeds determined by comp(i)
  - All non-determinism embedded in schedule!

Schedules (2)

- Given the initial configuration
  - The schedule determines the whole execution
- Not all schedules allowed for an initial conf.
  - del(i,j,m) only allowed if m is in outbuf of i in previous configuration

Admissible executions (aka fairness)

- An execution is admissible if
  - each process has infinite number of comp(i), and
  - every message m sent is eventually del(i,j,m)
- Why infinity?
  - Executions are infinite (this permits messages to wait arbitrary long finite times before being delivered!)
  - When algorithm is finished, only make dummy transitions (same state)

Synchronous Systems

- Lockstep execution
  - 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 Formally

- Execution partitioned into disjoint rounds
- Round consists of
  - Deliver event for every message in all outbufs
  - One computation event on every process
- Every execution is admissible
  - Executions by definition infinite
    - Processes take infinite steps
    - Every message is delivered

Time, clocks & order of events

Order of Events

- The following theorem shows an important result:
  - The order in which two applicable computation events or two applicable delivery events are executed is irrelevant!
  - (make a picture!)
- **Theorem:**
  - Let a and b be two different comp events applicable in **c**, then
    - a is applicable to app(b, **c**)
    - b is applicable to app(a, **c**)
    - app(b, app(a, **c**)) = app(a ,app(b, **c**))

Ordering Proof

- a and b are associated with transitions
  - a and b cannot be on the same node, since two different events cannot be applicable at the same time
  - a=(s1,s2,i) and b=(s3,s4,j) for i≠j
  - Since a is applicable in c, state of node i is s1 in c
  - Since b is applicable in c, state of node j is s3 in c
  - Since transition b only changes state of node j in c
    - State of node i is still s1 in app(b, c)
    - Thus, a is applicable in app(b, c)
    - Similarly, b will be applicable in app(a,c)
    - app(a, app(b, c)) = app(b, app(a, c)) since a and b do not change the other one's state

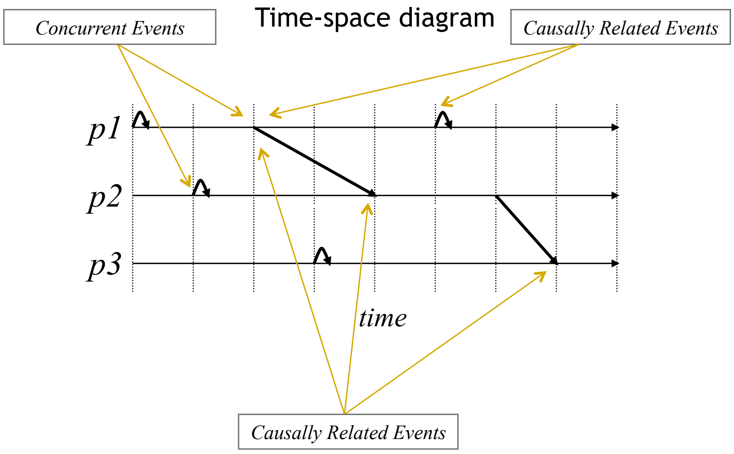
Order sometimes matters

- The theorem says nothing in two cases
  - If both are comp(i) on same node i
  - One delivers message m, other outputs or consumes m through a comp(i)
- In above cases both events cannot be applicable in C
- In above cases the two events are causally related

Causal Order

- The relation $<_H$ on the events of an execution (or schedule), called causal order, is defined as follows
  - If a occurs before b on the same process, then $a <_H b$
  - If a produces (comp) m and b delivers m, then $a <_H b$
  - If a delivers m and b consumes (comp) m, then $a <_H b$
  - $<_H$ is transitive.
    - I.e. If $a <_H b$ and $b <_H c$ then $a <_H c$
- Two events, a and b, are concurrent if not $a <_H b$ and not $b <_H a$
  a||b

Example of Causally Related events



Similarity of executions

- The view of p_i in E, denoted $E|p_i$, is
  - the subsequence of execution E restricted to events and state of p_i
- Two executions E and F are similar w.r.t p_i if
  - $E|p_i = F|p_i$
- Two executions E and F are similar if
  - E and F are similar w.r.t every node

Equivalence of Executions: *Computations*

- **Computation Theorem:**
  - Let E be an execution $(c_0,e_1,c_1,e_2,c_2,\ldots)$, and V the schedule of events $(e_1,e_2,e_3,\ldots)$
    - I.e. $app(e_i,c_{i-1})=c_i$
  - Let P be a permutation of V, preserving causal order
    - $P=(f_1, f_2, f_3\ldots)$ preserves the causal order of V when for every pair of events, $f_i <_H f_j$ implies $i<j$
  - Then E is similar to the execution starting in c_0 with schedule P

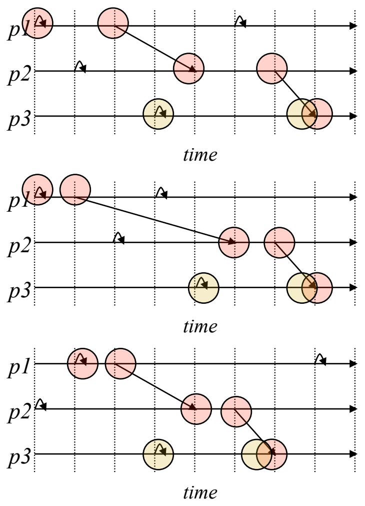
Equivalence of executions

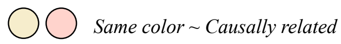
- If two executions F and E have the same set of events, and their causal order is preserved, then F and E are said to be similar executions, written $F{\sim}E$
  - F and E could have different permutation of events as long as causality is preserved!

Computations

- Similar executions form equivalence classes where every execution in class is similar to the other executions in the class.
- I.e. the following always holds for executions:
  - ~ is reflexive
    - I.e. a~ a for any execution
  - ~ is symmetric
    - I.e. If a~b then b~a for any executions a and b
  - ~ is transitive
    - If a~b and b~c, then a~c, for any executions a, b, c
- Equivalence classes of executions are called computations

Example of similar executions





- All three executions are part of the same computation, as causality is preserved

Two important results (1)

- Computation theorem implies two important results
- Result 1: There is no algorithm that can observe the order of the sequence of events (that can "see" the time-space diagram) for all executions
- Proof:
  - Assume such an algorithm exists. Assume node p knows the order in the final repeated configuration.
  - Take two distinct similar executions of algorithm that preserve causality
  - Computation theorem says their final repeated configurations are the same, therefore the algorithm cannot have observed the actual order of events as they differ

Two important results (2)

- Result 2: The computation theorem does not hold if the model is extended such that each process can read a local hardware clock

- Proof:
  - Assume a distributed algorithm in which each process reads the local clock each time a local event occurs
  - The final (repeated) configuration of different causality preserving executions will have different clock values, which would contradict the computation theorem

Observing Causality

- So causality is all that matters…

- …how to locally tell if two events are causally related?

Lamport Clock

- Each process has a local logical clock, kept in variable t, initially t=0
  - Node p piggybacks (t, p) on every sent message

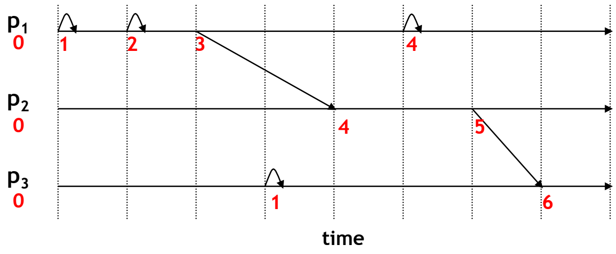
- On each event update t:
  - t := max(t, t_q)+1   (delivery)
    - When p receives message with timestamp (t_q, q)
  - t := t + 1 for every transition (comp)

Lamport Clock (2)

- Comparing two timestamps (t_p,p) and (t_q,q)
  - (t_p,p)<(t_q,q) iff (t_p<t_q or (t_p=t_q and p<q))
  - i.e. break ties using node identifiers
  - e.g. (5,p_5)<(7,p_2), (4,p_2)<(4,p_3)

- Lamport logical clocks guarantee that:
  - If a $<_H$ b, then t(a) < t(b),
    - where t(a) is Lamport clock of event a

Example of Lamport logical clock



Vector Clock

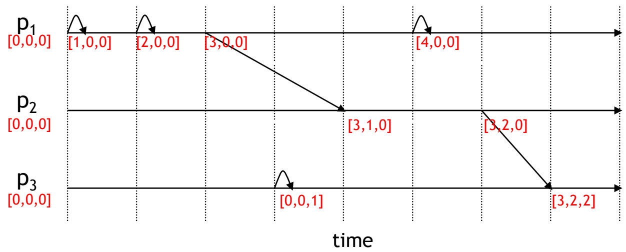
- Each process p has local vector v_p of size n
  - $v_p[i]=0$ for all i
  - Piggyback v_p on every sent message
- For each transition update local v_p by
  - $v_p[p] := v_p[p] + 1$
  - $\forall i$: $v_p[i] := \max(v_p[i], v_q[i])$
    - where v_q is clock in message received from node q

Comparing Vector Clocks

- $v_p \leq v_q$ iff
  - $v_p[i] \leq v_q[i]$ for all i
- $v_p < v_q$ iff
  - $v_p \leq v_q$ and for some i, $v_p[i] < v_q[i]$
- v_p and v_q are concurrent ($v_p \,||\, v_q$) iff
  - not $v_p<v_q$, and not $v_q<v_p$

- Vector clocks guarantee
  - If v(a) < v(b) then a $<_H$ b, and
  - If a $<_H$ b, then v(a) < v(b)
    - where v(a) is the vector clock of event a

Example of Vector Clocks



Great! But cannot be done with smaller vectors than size n, for n nodes

Partial and Total Orders

- Is it a partial order or a total order? [d]
  - the relation $<_H$ on events in executions
    - Partial: $<_H$ doesn't order concurrent events
  - the relation < on Lamport logical clocks
    - Total: any two distinct clock values are ordered
  - the relation < on vector timestamps
    - Partial: timestamp of concurrent events not ordered

Lamport Clock vs. Vector Clock

- Lamport clock
  - If $a <_H b$ then $t(a) < t(b)$ (1)
- Vector clock
  - If $a <_H b$ then $t(a) < t(b)$ (1)
  - If $t(a) < t(b)$ then $a <_H b$ (2)
- Which of (1) and (2) is more useful? [d]
- What extra information do vector clocks give? [d]

Complexity

Complexity of Algorithms

- We care about
  - Number of messages used before terminating
  - Time it takes to terminate
- Termination
  - A subset of the states Q_i are terminated states
- Algorithm has terminated when
  - All states in a configuration are terminated
  - No messages in {in,out}bufs

Message Complexity

- Maximum number of messages until termination for all admissible executions
  - This is worst-case message complexity…

(Admissible ≈ fairness ≈ all messages sent eventually delivered + executions infinitely long)

Time Complexity

- Basic idea of time complexity
  - Message delay is at most 1 time unit
  - Computation events take 0 time units
- Formally, timed execution is an execution s.t.
  - Time is associated with each comp(i) event
  - First event happens at time 0
  - Time can never decrease & strictly increases locally
  - Max time between comp(i) sending m and comp(j) consuming m is 1 time unit
  - Time complexity is maximum time until termination for all admissible timed executions

Time Complexity (2)

- Why at most 1?
  - Why not assume every msg takes exactly 1 time unit?

- Would not model reality
  - Some algorithms would have misleading time complexity

At most is less or more than equal?

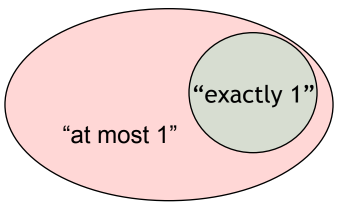
- Compare "at most" vs. "exactly" 1 time unit
  - How do they compare? [d]

Time Complexity: broadcasting

- **Init:**
```
parent = null
n = total number of nodes
```
- **Source:**
```
send <a> to all neighbors
wait to receive n-1 <b>
```
- **Others:**
```
when receive <a> from p:
if parent==null:
  parent := p
  forward <a> to all neighbors except <parent>
  send <b> to parent
when receive <b>:
  send <b> to parent
```
- What is the time complexity if every message takes
  - At most 1 time unit?
  - Exactly 1 time unit?

“At most” can only raise complexity

- “at most” can increase time complexity
  - Every timed execution with “exactly” 1 time unit is possible in the “at most” model
  - The “at most” model has other executions too
  - Time complexity considers the maximum time
    - Time complexity of “at most” can only increase over “exactly”
- In broadcast example:
  - <a> takes 0 or 1 time unit
  - <b> takes 1 time unit
  - Long <a> paths can be fast
  - But <b> path will be very slow!



Summary

- The total order of executions of events is not always important
  - Two different executions could yield the same “result”
- Causal order matters:
  - Order of two events on the same process
  - Order of two events, where one is a send and the other one a corresponding receive
  - Order of two events, that are transitively related according to above
- Executions which contain permutations of each others events such that causality is preserved are called *similar* executions
- Similar executions form equivalence classes called *computations*
  - Every execution in a computation is similar to every other execution in the computation
- Vector timestamps can be used to determine causality
  - Cannot be done with smaller vectors than size n, for n processes