LINFO2345

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 (infinitely) complex, model simplifies reality (finite approximation)

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

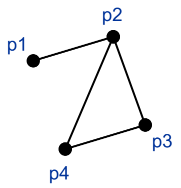
- Biggest challenge of modeling
  - Choosing the *right level of abstraction*!
- Model must be powerful enough to construct
  - *Impossibility proofs*
    - A statement about all possible algorithms in a system
  - But it should not be too complicated! This is a delicate balance.
- Our model should therefore be:
  - *Complete*: explain all relevant properties (nothing missing)
  - *Correct*: behave as the system does (no mistakes)
  - *Concise*: explain a class of distributed systems compactly

Model of Distributed Systems

Based on model from Attiya & Welch

Model of Distributed Computing

- What is a distributed system?
  - A set of nodes/processes
  - sending messages over a network
  - to solve a common goal (algorithm)



- How do we model this?
  - Model one node: internal node state
  - Model many nodes: communication network

Modeling one node

9/21/21 9

9

Modeling one 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

10

State Transition System – Informal

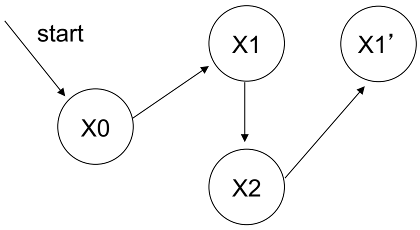
- A state transition system consists of
  - A set of states
  - Rule for which state to go to from each state (transition function/binary relation)
  - The set of starting states (initial states)

State Transition System – Example

- Example algorithm:

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

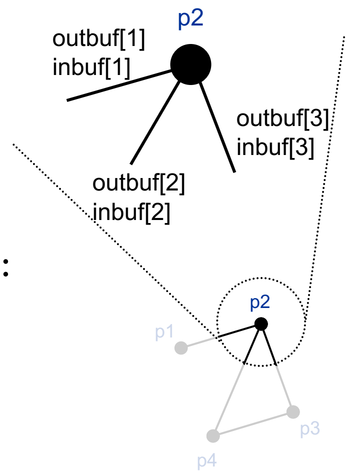
Using graphs:



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

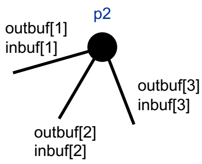
Modeling one node

- State machine of node i
  - Set of states Q_i
- Each state consists of
  - Local state of the node
  - Network state *relevant to node*:
    - 1 inbuffer set for each neighbor
    - 1 outbuffer set for each neighbor
- Initial states
  - inbuf[j] is empty for all j

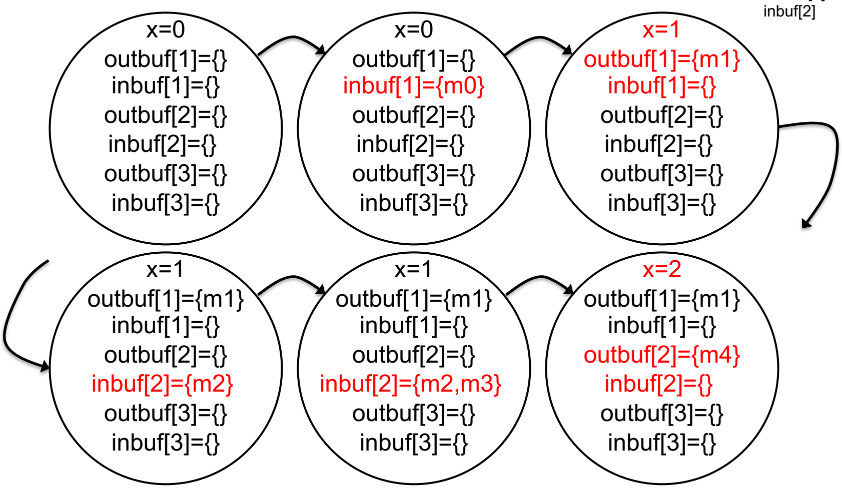


13

State of one node



- Example states

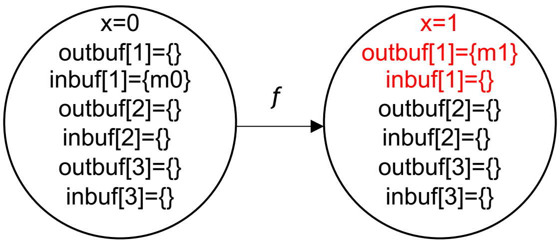


14

Transition functions

- All state *except outbufs* is called the accessible state
  - when in outbuf, can't read it any more ("in the network")
  - inbuf is read-only and outbuf is write-only ("network state")

- Transition function *f*
  - takes accessible state and returns new state
  - sends messages: adds at most 1 msg to each outbuf[i]
  - reads messages: all inbuf[i] of new state must be empty



15

Transition functions formally

- State of a node (with k channels) is triple <I,O,s>
  - <I,O> is the network state
    - 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

- Correctness condition for network state:
  - $f(<I_1,O_1,s_1>)=<I_2,O_2,s_2>$ and $f(<I_3,O_3,s_3>)=<I_4,O_4,s_4>$
    - $I_2=I_4=<\varnothing,\ldots,\varnothing>$, i.e. all inbufs are empty, and
    - If $I_1=I_3$ and $s_1=s_3$ then
      - $s_2=s_4$, i.e. deterministic but don't read outbufs (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. outbufs keep their old messages (2)

- Definition from Attiya & Welch book
  - Note there is a small error in the book: output buffers are not read, yet they must be passed on

16

Single node perspective

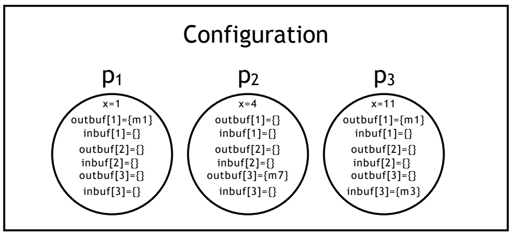
- Execution of one node:
  - 1. Wait for message
  - 2. When received message, do some local computation, send some messages
  - Goto 1.
- Is this a correct model? [d]
  - Is it deterministic? Where is the nondeterminism?
  - How is I/O done?
  - Is the execution of one node atomic?

Modeling many nodes

Modeling many nodes (full 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

Modeling many nodes (full system)

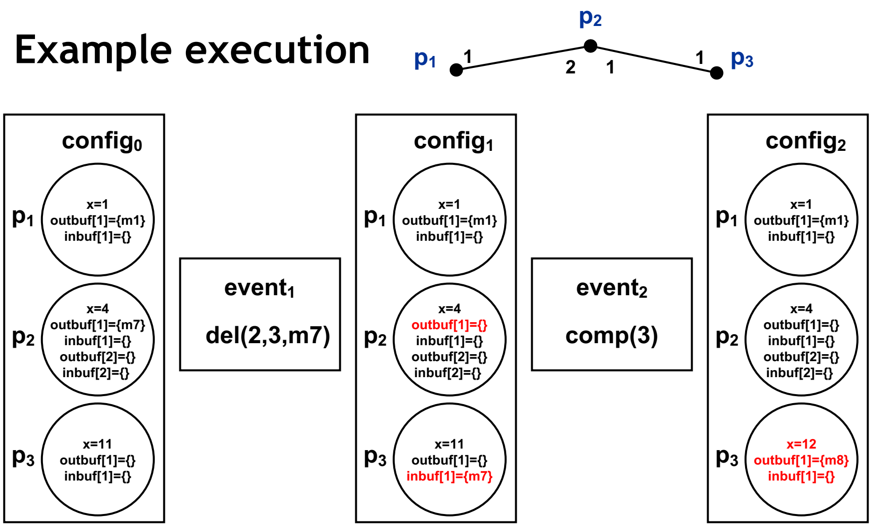
- The distributed system evolves by *events*
  - Computation event at node i: *comp(i)*
  - Delivery event of msg m from i to j: *del(i,j,m)*
- Computation event comp(i)
  - Apply transition function *f* on node i's state
- Delivery event del(i,j,m)
  - Move message m from outbuf of p_i to inbuf of p_j

Execution of distributed system

- An execution E is an infinite sequence of configurations c_k and events e_k
  - $E=(c_0, e_1, c_1, e_2, c_2, e_3, c_3, \ldots)$
  - where c_0 is the initial configuration
- If e_k is comp(i)
  - c_{k-1} changes to c_k by
    applying p_i's transition function on i's state in $config_{k-1}$
- If e_k is del(i,j,m)
  - c_{k-1} changes to c_k by
    moving m from i's outbuf for link i-j to j's inbuf for link i-j

Modeling many nodes

Example execution

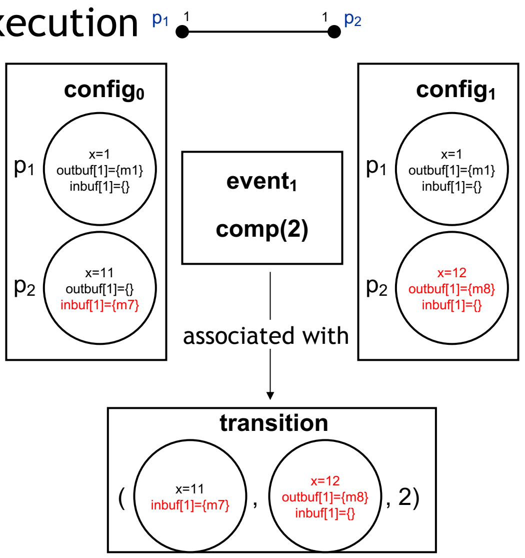


Some definitions for later use… (1)

- 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

From one node to 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 and Synchronism

Schedules (Asynchronous Model)

- Nodes are deterministic
  - Given some message, update state, send some messages, and wait…
- Nondeterminism comes from asynchrony
  - Messages take arbitrary time to be delivered
  - Processes execute at different speeds
- A schedule is a sequence of events (e_1, e_2, e_3, …)
  - Where each event e_k is del(i,j,m) or comp(i)
  - Message asynchrony is determined by del(i,j,m)
  - Process speeds is determined by comp(i)
  - All nondeterminism is embedded in the 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 (a.k.a. fairness)

- An execution is admissible if
  - each process has infinite number of comp(i)
    - This is exactly thread fairness in a concurrent system!
  - every message m sent is eventually del(i,j,m)
    - We have to add a fairness condition for the network
- Why infinity?
  - Executions are infinite because it lets messages 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

Events & Causality

Concurrent events

- Consider these two executions:
  - (c_0 , comp(1), c_1 , comp(2), c_2)
  - (c_0 , comp(2), c'_1 , comp(1), c_2)

    Same configuration!
  - Two different comp events can be switched! [Why?]
- Consider these two executions:
  - (c_0 , del(1,2,m), c_1 , del(2,3,m'), c_2)
  - (c_0 , del(2,3,m), c'_1 , del(1,2,m'), c_2)

    Same configuration!
  - Two different del events can be switched! [Why?]
- Theorem: Order of two applicable comp or del events is irrelevant. We say that the events are concurrent.

Causally related events

- Consider this execution:
  - (c_0 , del(1,2,m) , c_1 , comp(2) , c_2)

    del(1,2,m): Move m from outbuf 1 to inbuf 2

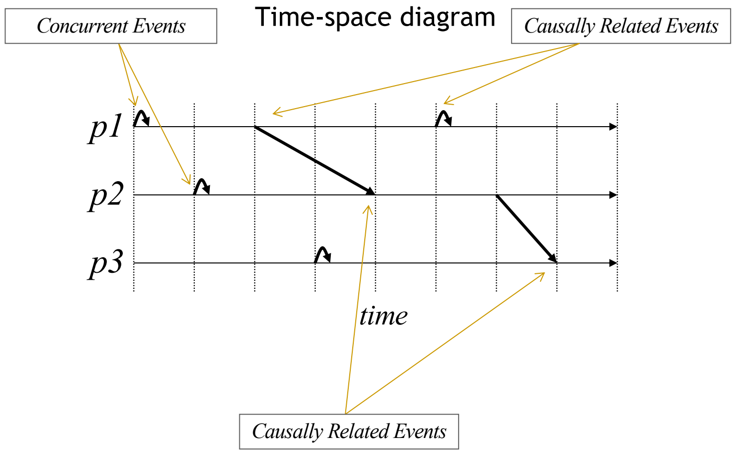
    comp(2): (s_1,s_2,2), m in inbuf 2
  - These two events cannot be switched! [Why not?]
- The events del(1,2,m) and comp(2) are causally related
- We can define causality for all events…

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 $b <_H a$
  We write this as follows: $a \| b$

35

Example of Causally Related events



36

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 $V=(e_1,e_2,e_3,\ldots)$
    - 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

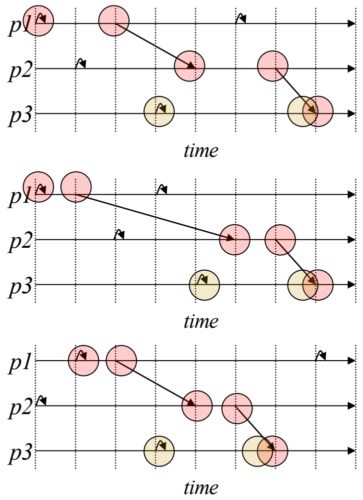
Equivalence of executions

- 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

41

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

42

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 and Vector Clock

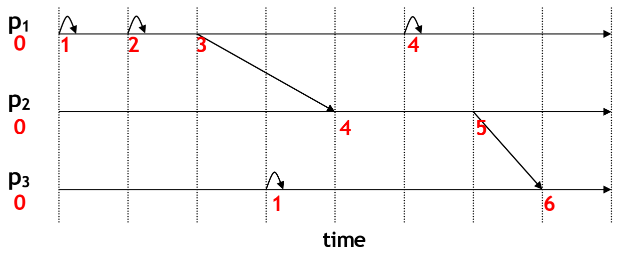
Lamport Clock

- 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



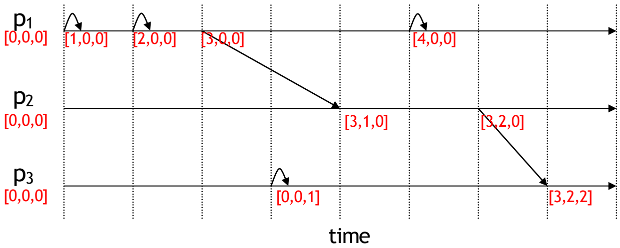
Vector Clock

- 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

51

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

52

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]

Summary

Summary of Causality and Clocks

- The total order of executions of events is not always important
  - Two different executions can yield the same result (if *same causality*)
- Causal order matters:
  - Order of two events on the same process
  - Order of two events, a send and the 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

55

Complexity

56

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 are 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 value 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 time unit?
  - Why not assume every message 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!