Distributed Computing

José Orlando Pereira

HASLab / Departamento de Informática
Universidade do Minho

DOCTORAL PROGRAMME IN COMPUTER SCIENCE

2011/2012

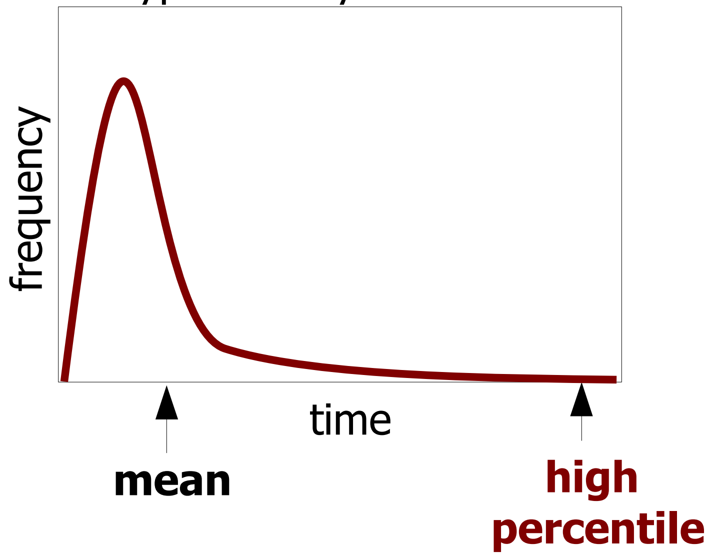
Asynchronous systems

- Assume no bounds on:
  - clock drift
  - processing time
  - message passing time

In practice

- Tight synchronous limits are dangerous:
  - Round time proportional to mean delay
  - Low coverage or expensive systems

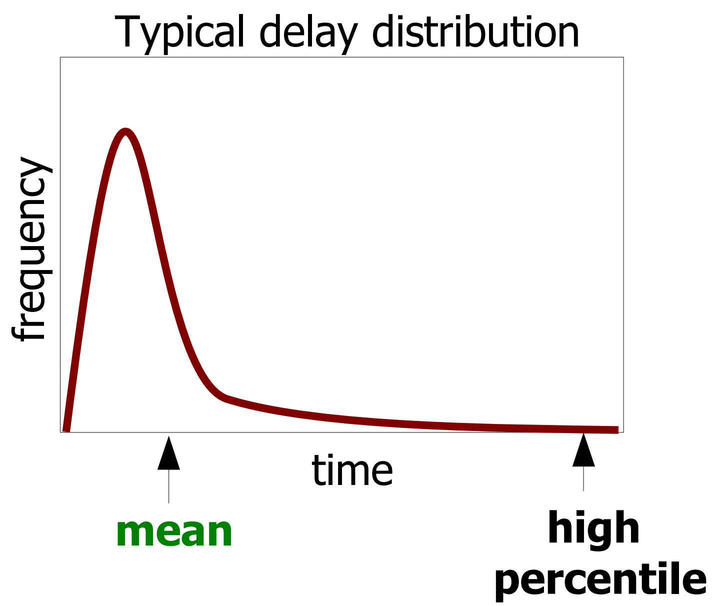
Typical delay distribution



- Large synchronous limits are not useful:
  - Round time proportional to high percentile delay
  - Taking advantage of synchrony causes a very large performance penalty

In practice

- Solutions for asynchronous systems might have better performance:
  - Round time proportional to mean delay
  - Even if more message exchanges are necessary



In theory

- Start with a synchronous reliable fully connected network
- Relax the system model:
  - Unbounded message loss
  - Large/unknown graph diameter
  - Dynamic graph
- Example: Leader election

Example: Leader election

Static known participants

Synchronous
Reliable static

Synchronous
Reliable dynamic

Synchronous
Reliable clique

Synchronous
Unreliable clique

Synchronous
Bounded unreliable
Clique

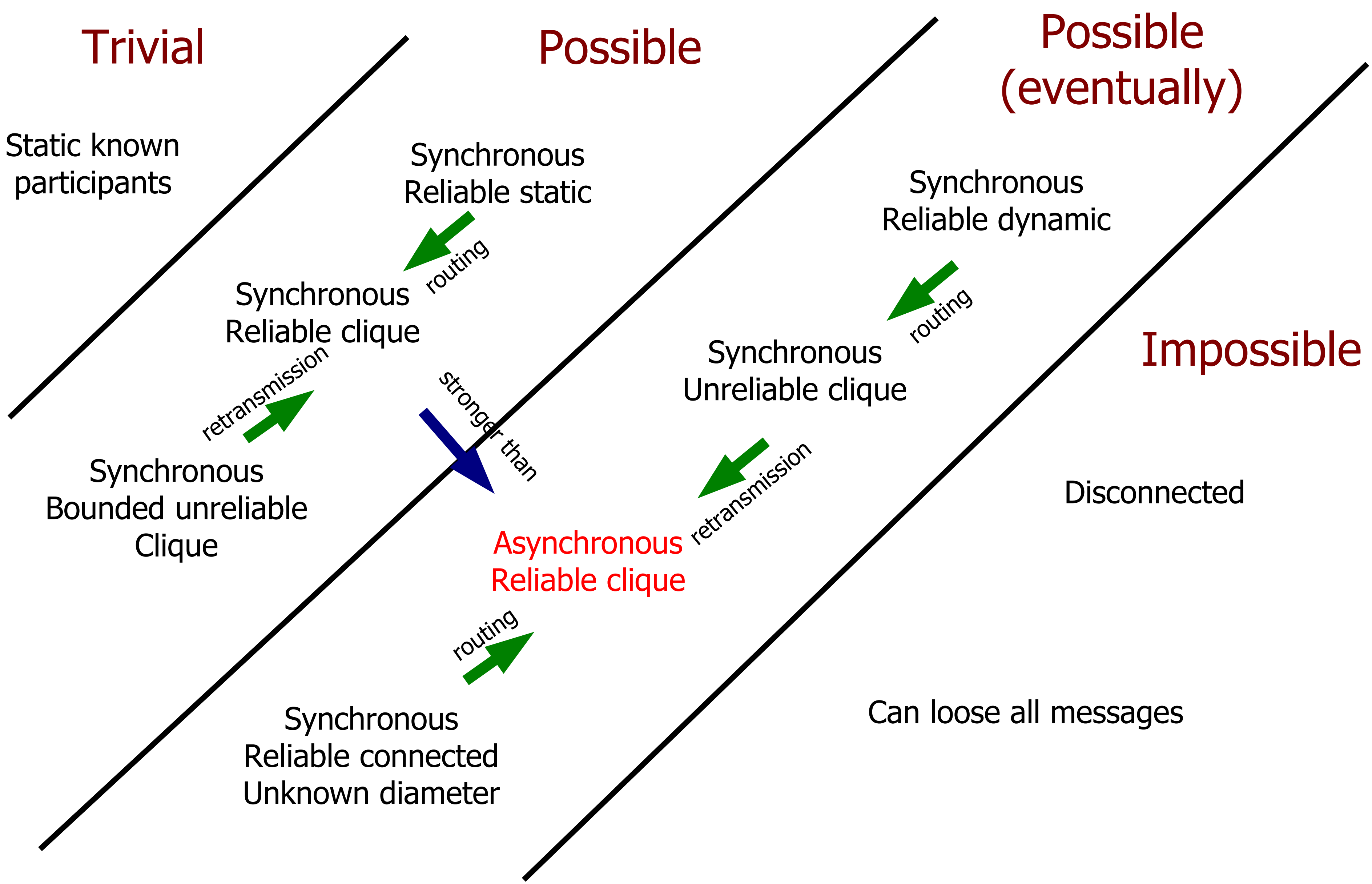
Disconnected

Asynchronous
Reliable clique

Synchronous
Reliable connected
Unknown diameter

Can loose all messages

Example: Leader election



In theory

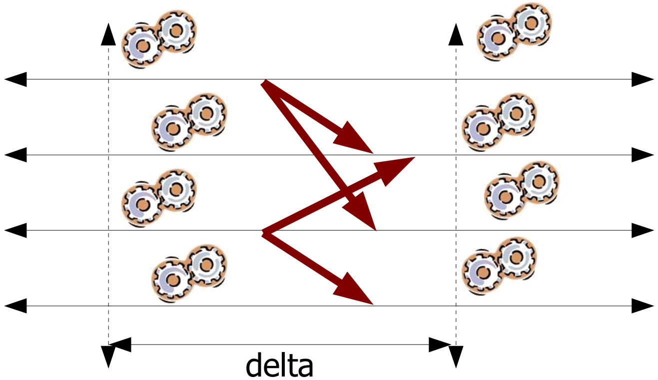
- Asynchrony subsumes:
  - Heterogeneity
  - Dynamics
  - Uncertainty
- Much simpler than handling them explicitly
- Often considered an Universal model:
  - Widely applicable solutions

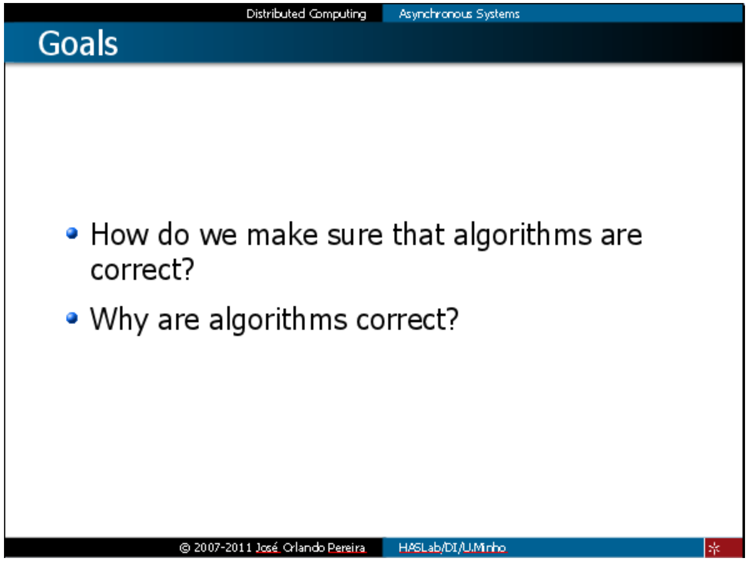
Goals

- How do we make sure that algorithms are correct?
- Why are algorithms correct?

Synchronous System

- With synchronous rounds:
  - Simple proofs by induction
  - Local state easily reflects global state




Distributed Computing | Asynchronous Systems

Goals

- How do we make sure that algorithms are correct?
- Why are algorithms correct?

© 2007-2011 José Orlando Pereira | HASLab/DI/U.Minho

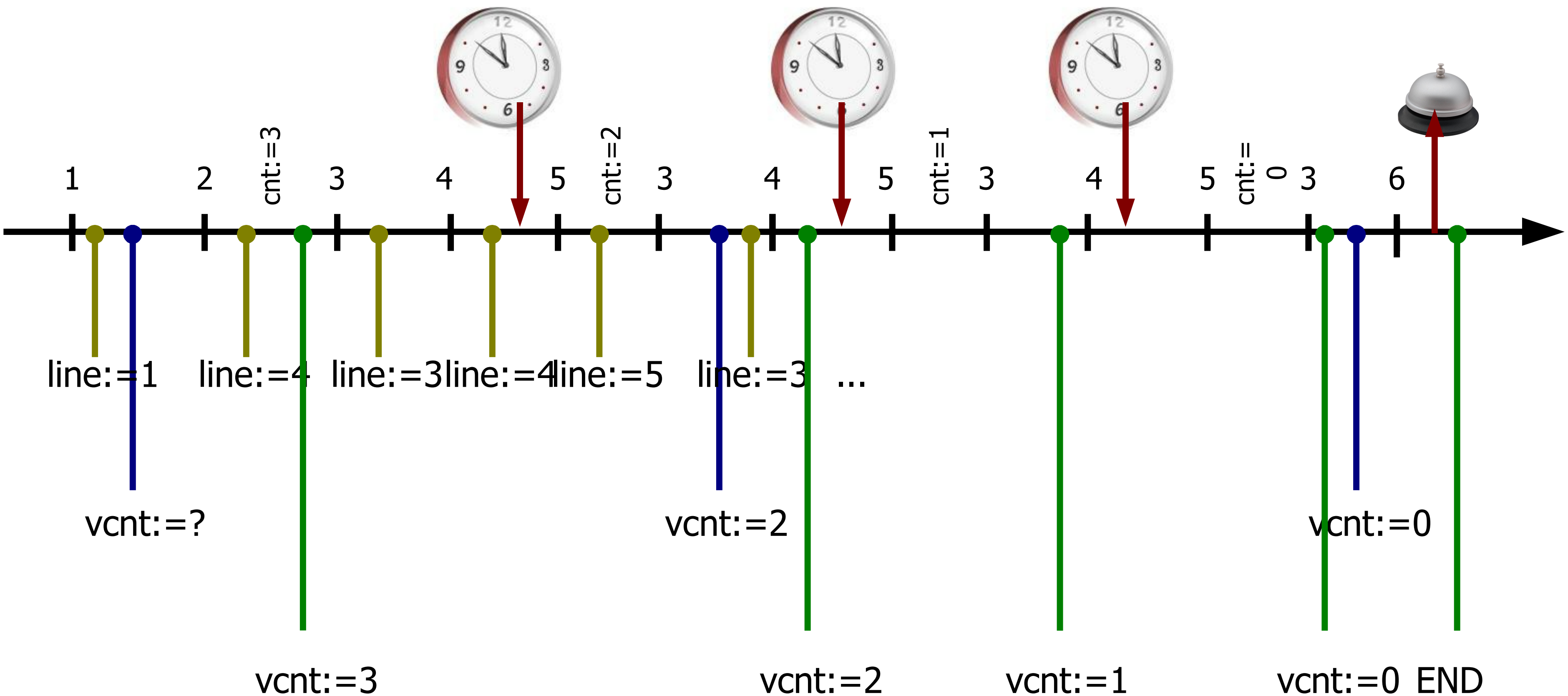
Sample computation

- An alarm clock program:

```
main:                   // line 1
    cnt:=3              // line 2
    while cnt>0:        // line 3
        sleep 1s        // line 4
        cnt := cnt-1    // line 5
    ring                // line 6
```

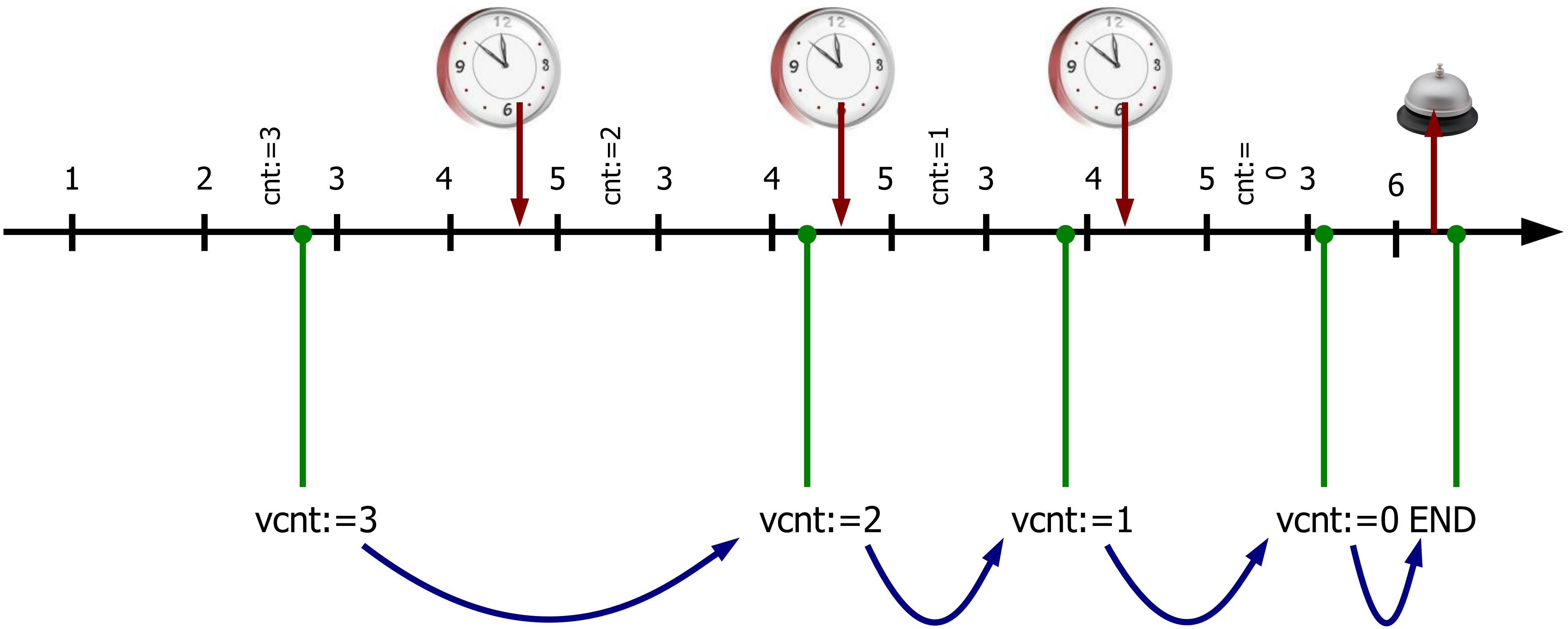
Observation

- Select model variables and periodically observe the system:



Abstraction

- Choose observation that allows reasoning on the desired properties:



Behaviors/Executions

- Consider all possible sequences of chosen atomic actions:

vcnt:=3 vcnt:=2 vcnt:=1 vcnt:=0 END

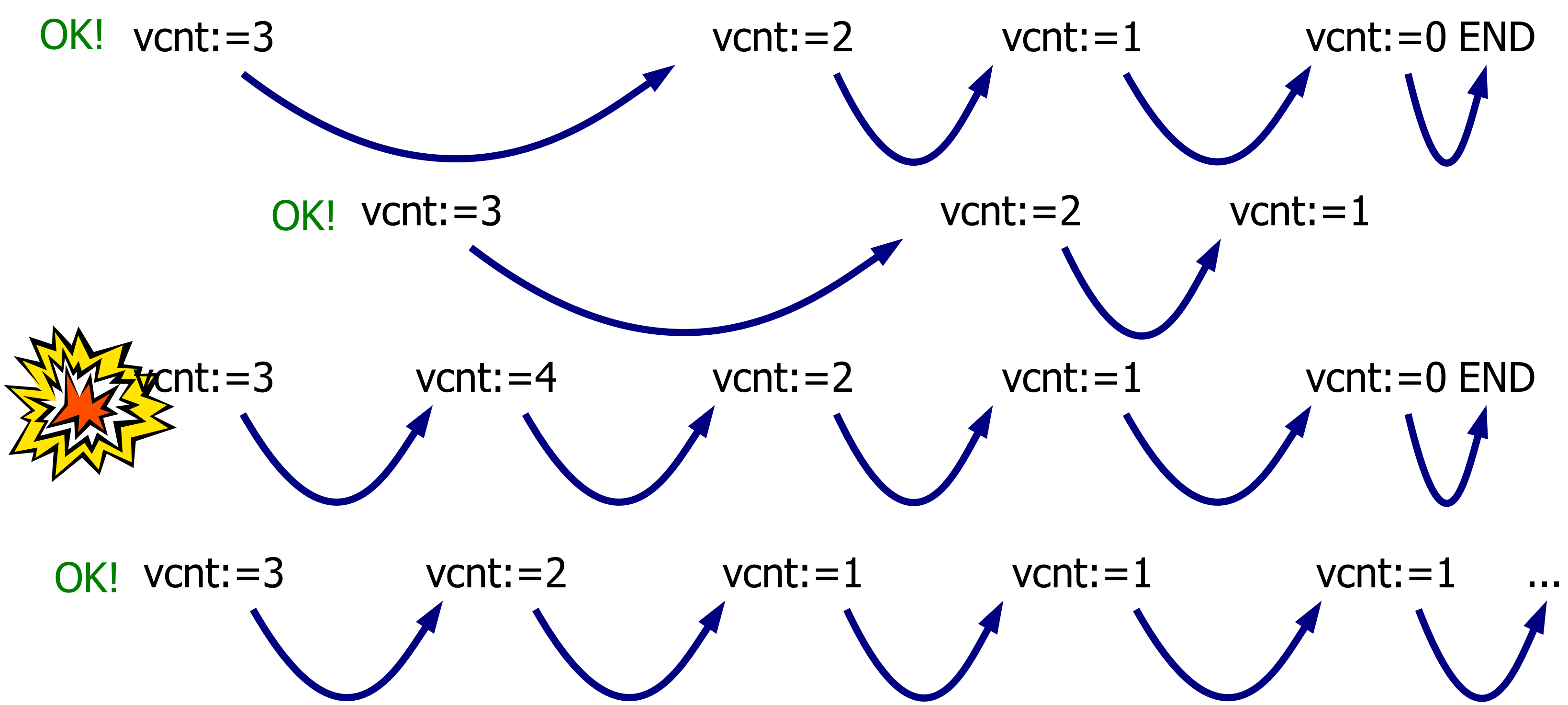
vcnt:=3 vcnt:=2 vcnt:=1

vcnt:=3 vcnt:=4 vcnt:=2 vcnt:=1 vcnt:=0 END

vcnt:=3 vcnt:=2 vcnt:=1 vcnt:=1 vcnt:=1 ...

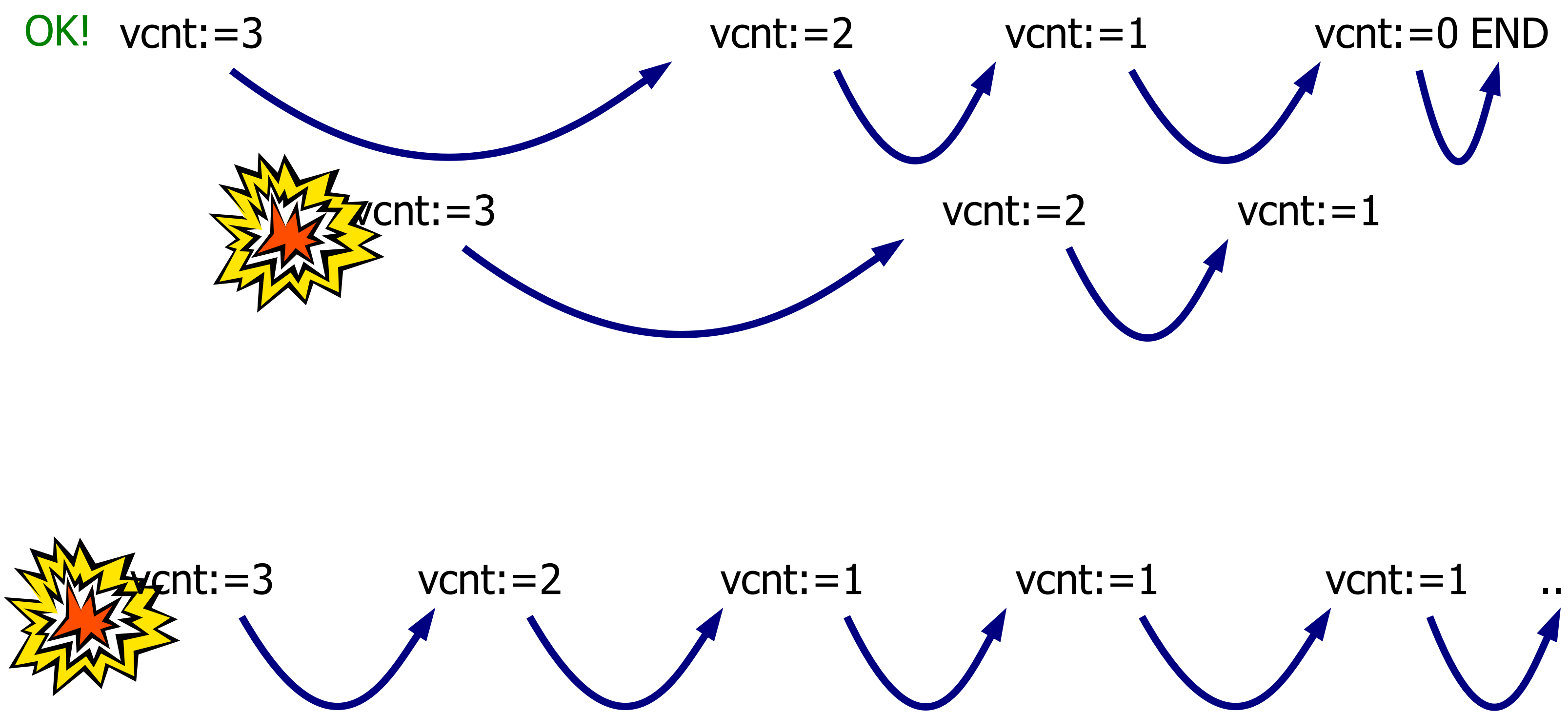
Safety properties

- Nothing bad ever happens:



Liveness properties

- Something good eventually(*) happens:



(*) eventually = inevitavelmente ≠ eventualmente

Specification

- Specification is a set of allowable behaviors:

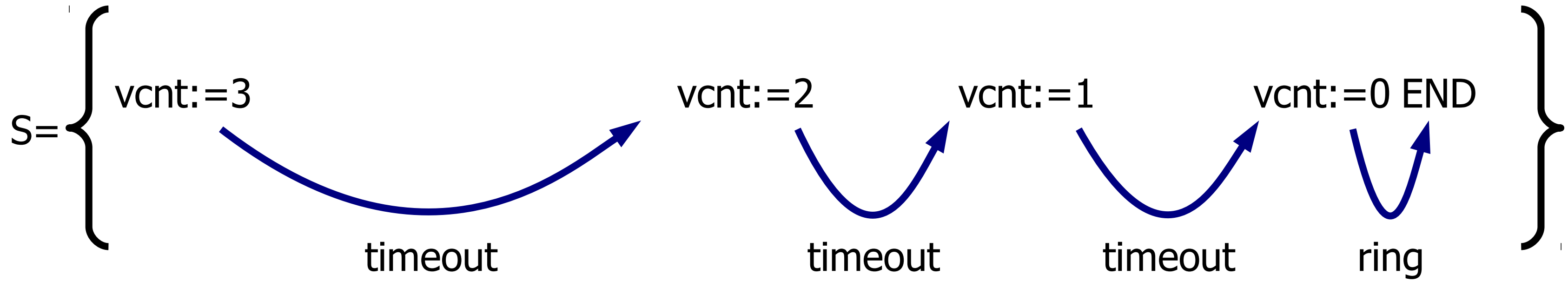
$$S=\left\{ \text{vcnt:=3} \xrightarrow{\text{timeout}} \text{vcnt:=2} \xrightarrow{\text{timeout}} \text{vcnt:=1} \xrightarrow{\text{timeout}} \text{vcnt:=0} \xrightarrow{\text{ring}} \text{END} \right\}$$

Goal 1: Is it correct?

- Is there a convenient representation for specification sets?
  - Compact
  - Practical
- How to prove safety and liveness properties?

Specifications and automata

- Specification is a set of allowable behaviors:



- An automaton provides a compact and practical representation

I/O Automata

- An I/O automaton A has five components:
  - sig(A), a triplet S of disjoint sets of actions:
    - in(S), the input actions
    - out(S), the output actions
    - int(S), the internal actions
  - states(A), a (possibly infinite) set of states
  - start(A), a non-empty subset of states(A)
  - trans(A), a subset of
    states(A) x acts(sig(A)) x states(A)
  - tasks(A), a partition of local(sig(A))

Transitions

- A transition is enabled in state s if there is some π,s' such that $(s,\pi,s') \in trans(A)$
- Input transitions are required to be enabled in all reachable states of A
- A state in which only input transitions are enabled is said to be quiescent

Signature and State

- Input:
  - none
- Internal:
  - Timeout
- Output:
  - Ring

- States:
  - vcnt, integer, initially 3
  - END, boolean, initially false

Transitions

- Timeout:
  - Pre-condition:
    - ¬END and vcnt>0
  - Effect:
    - vcnt := vcnt - 1

- Ring:
  - Pre-condition:
    - ¬END and vcnt = 0
  - Effect:
    - END := True

This is an equation,
not an attribution!

Effects

- Effect equation:
  - vcnt := vcnt - 1
- Read this as:
  - "vcnt-after = vcnt-before – 1 and the state otherwise unchanged"
- Could be written as:
  - vcnt-after + 1 = vcnt-before
  - vcnt-before - vcnt-after = 1
  - ...

Safe behaviors

- Enumerating safe behaviors:
  - Start with a behavior for each state s in start(A)
  - For each transition (s,a,s') in trans(A) enabled for some state s at the end of any known safe behavior:
    - Create a behavior with (a,s') appended
  - Repeat (possibly, for ever...)

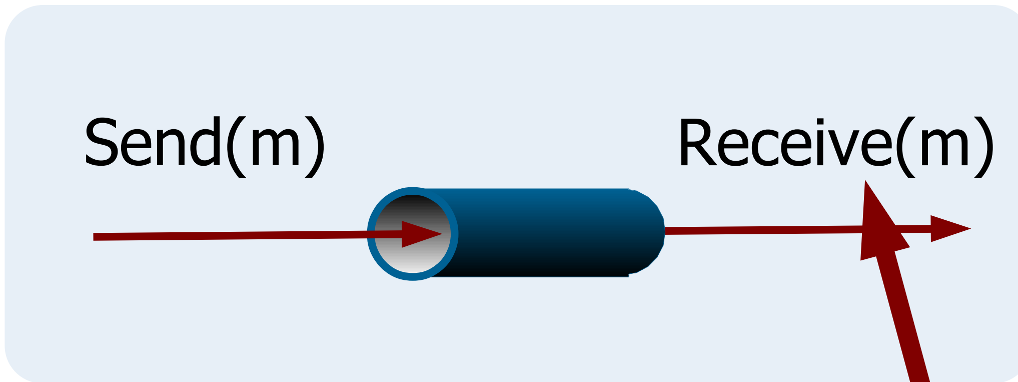
Safety properties

- Proof of safety properties:
  - Invariant proof by induction
- Strategies:
  - Strengthen the invariant
  - Include trace in state

Invariants

- Goal: Prove that always vcnt < 4 (safety!).
- Proof by induction:
  - Base step: True for all initial states?
    - 3<4: Yes!
  - Induction step: True for any next step?
    - Timeout transition:
      - vcnt-after = vcnt-before - 1
      - vcnt-before < 4
vcnt-after+1 < 4
vcnt-after < 3 < 4: Done
    - Ring transition:
      - always vcnt-after = vcnt-before = 0
      - 0<4: Done

Example: Reliable channel



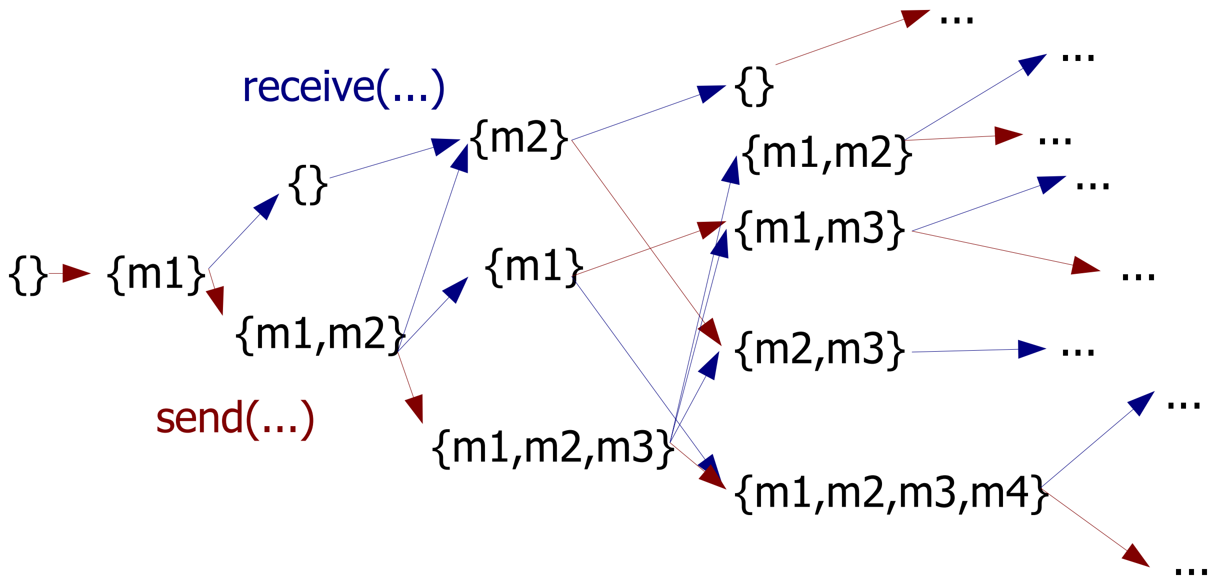
- Reliable channel:
  - Unordered
  - FIFO

Why *Receive(m)* and not *m := Receive()*?

Example: Reliable channel

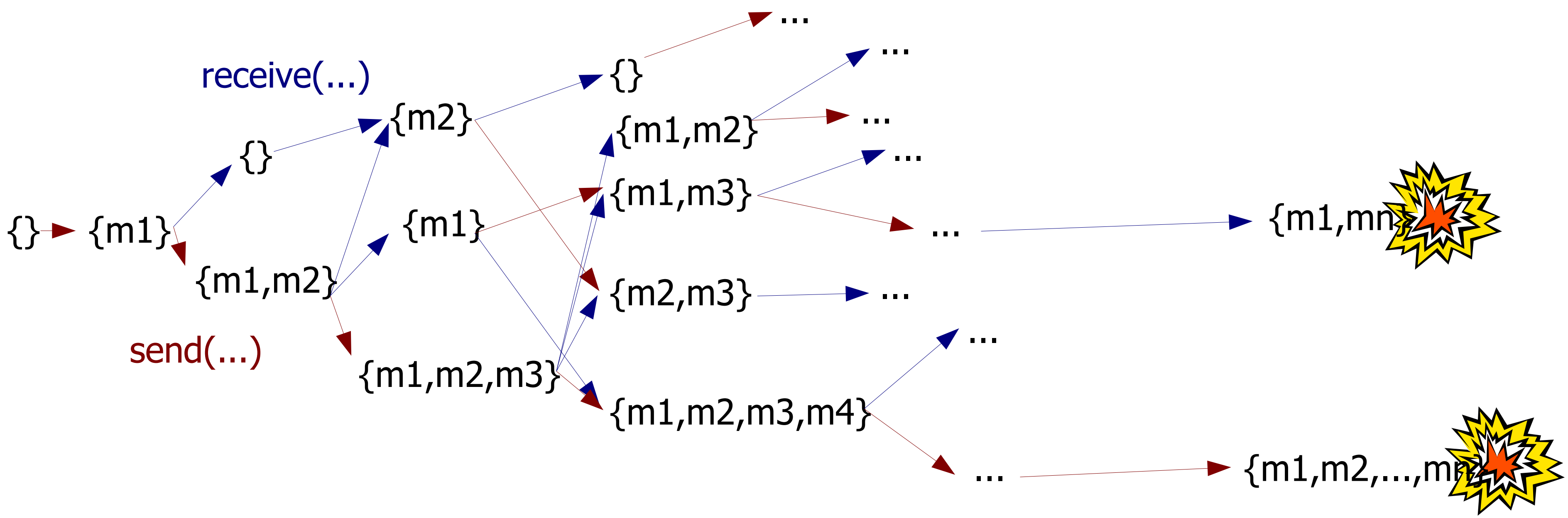
- State:
  - transit, bag of M, initially {}
- Send(m), $m \in M$:
  - Pre-condition:
    - True
  - Effect:
    - transit :=transit + {m}
- Receive(m), $m \in M$:
  - Pre-condition:
    - m in transit
  - Effect:
    - transit := transit - {m}

Behaviors of a channel



- Concurrency is modeled by alternative enabled transitions:
  - Sender and receiver
  - Within the channel (reordering)

Liveness and fairness

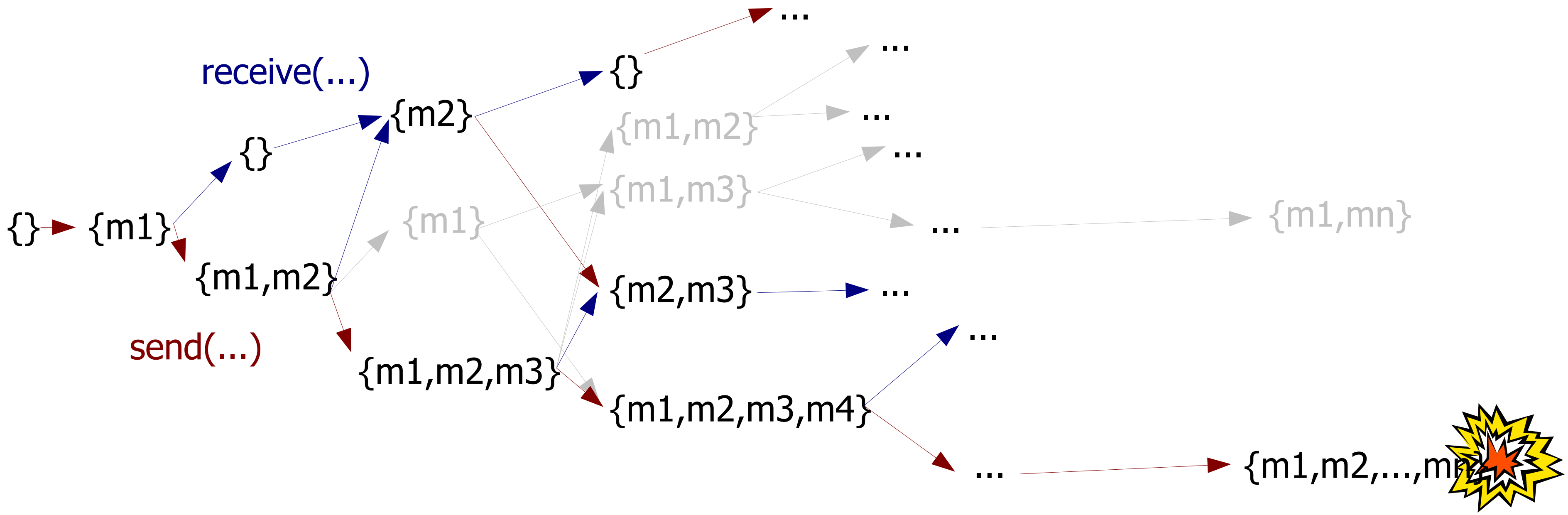


- Some behaviors do not satisfy liveness:
  - If m is sent, eventually m is received
- Some transitions don't get a fair chance to run:
  - receive(m1) and receive(m*)

Fairness

- Partition transitions in tasks:
  - Tasks:
    - For all m: {receive(m)}
- Assume that no task can be forever prevented from taking a step
- What about a FIFO reliable channel?

Liveness and fairness



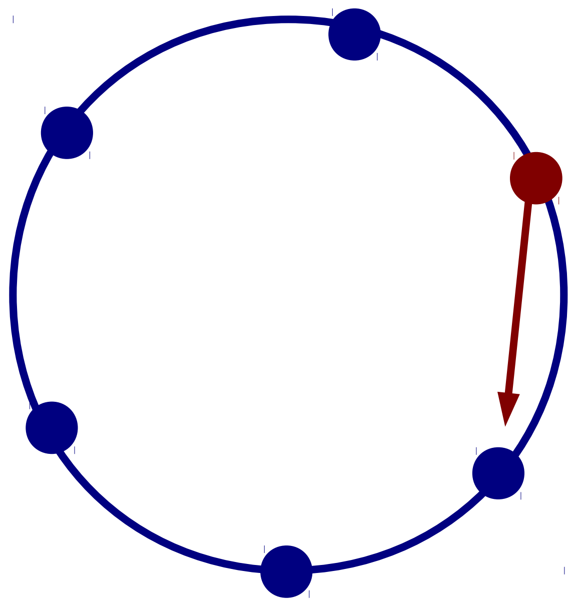
- FIFO order excludes a number of behaviors
  - Only executions with a finite number of receive(m) steps are unfair
- Fairness ensured by a single task:
  - {For all m: receive(m)}

Example: FIFO channel

- State:
  - transit, seq. of M, initially <>
- Send(m), $m \in M$:
  - Pre-condition:
    - True
  - Effect:
    - transit :=transit+<m>
- Receive(m), $m \in M$:
  - Pre-condition:
    - m=head(transit)
  - Effect:
    - transit := tail(transit)
- Tasks:
  - {For all m: receive(m)}

Example: Token ring

- Rotating token algorithm:



- Mutual exclusion?
- Deadlock freedom?

Example: Token ring

- State:
  - n is the number of nodes
  - token[0]=1
  - token[i]=0, for 0<i<n
- Move(i):
  - Pre-condition:
    - token[i]=1
  - Effect:
    - token[i]:=0
    - token[(i+1) mod n]:=1

Example: Token ring

- Mutual exclusion:
  - There is at most one token in the ring (i.e. sum of token[i]$\leq$1)
- Proof by induction:
  - Base step:
    - $\sum$token[i]=1 trivially true
  - Induction step:
    - $\sum$token-before[i]$\leq$1$\Rightarrow$$\sum$token-after[i]$\leq$1

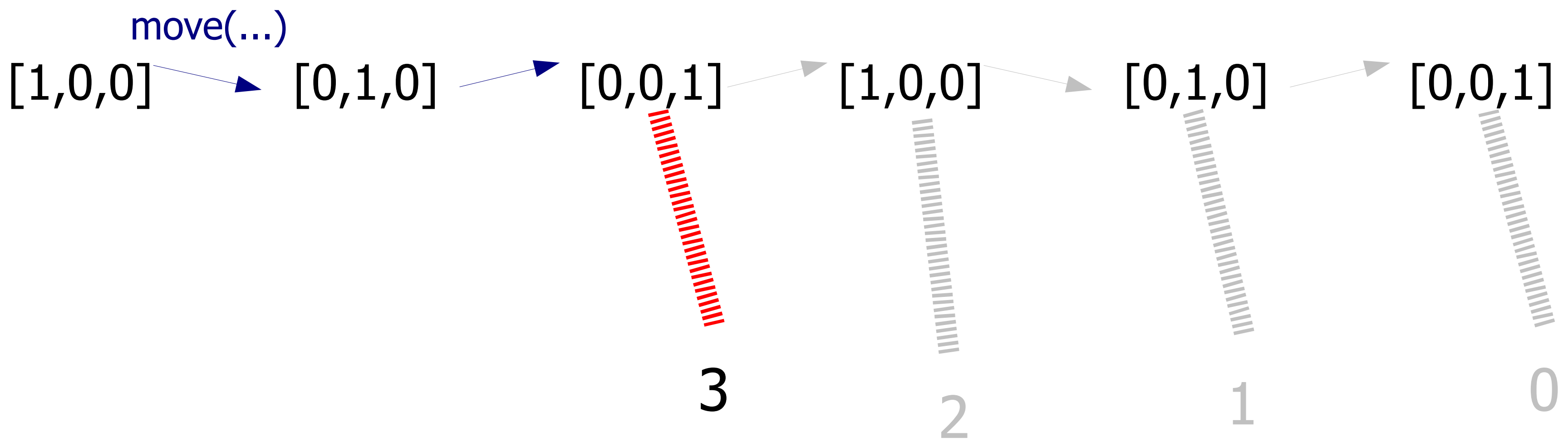
Example: Token ring

- No starvation:
  - Eventually i gets the token at least k times
- Proof with a progress function:
  - Function from state to a well-founded set
  - Helper actions decrease the value
  - Other actions do not increase the value
  - Helper actions are taken until goal is met (i.e. enabled and in separate tasks)

Invariant assertion

Progress function

- Define progress function f as:
  - Target is non-negative integers
  - Value is ((k-1) x n + i - 1) - length(trace)
- Example with n=3, k=2, and i=3:

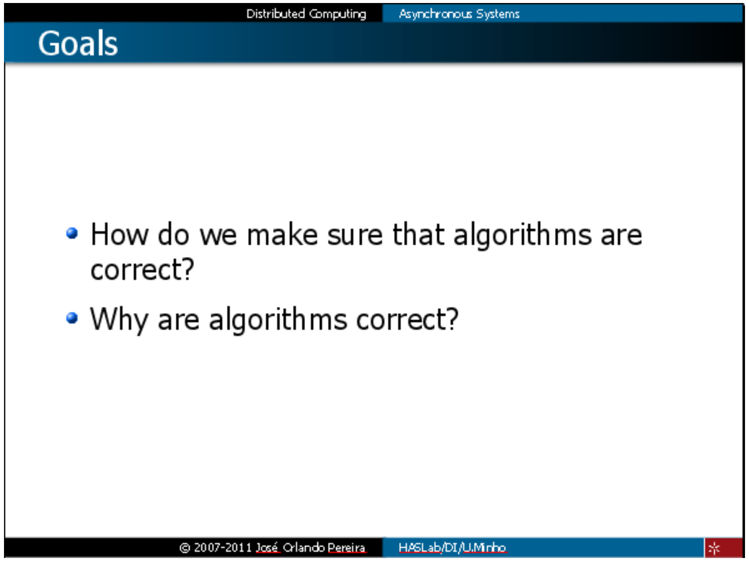


Summary

- I/O Automata definition
  - Safety specification
  - Fairness specification
- Proof strategies for:
  - Invariants
  - Trace properties
    - Safety
    - Liveness

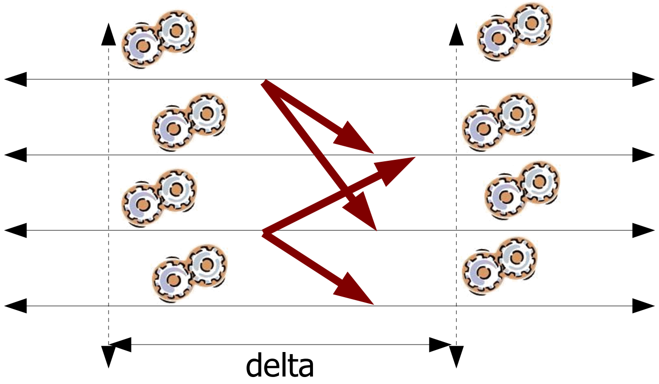
Conclusion

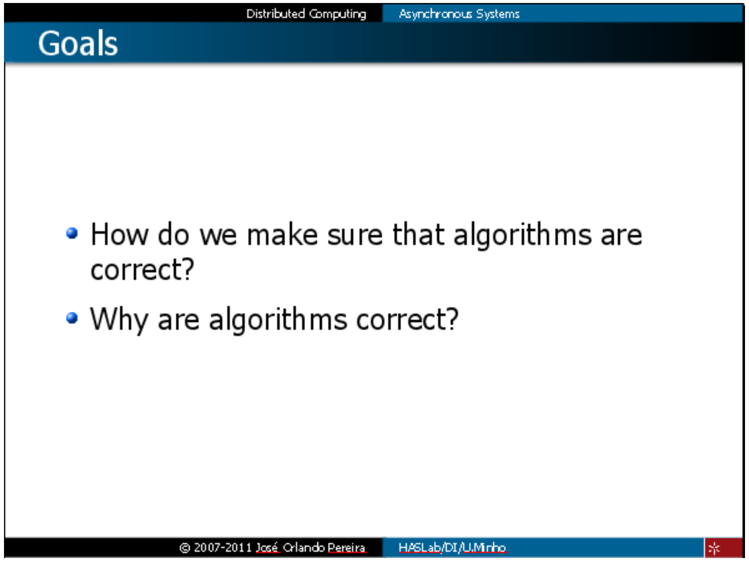
- First goal achieved:
  - I/O Automata
  - Safety and liveness proofs
- More:
  - Composition
  - Refinement



Goal 2: Why is it correct?

- With synchronous rounds, local state easily reflects global state
- What about in an asynchronous system?



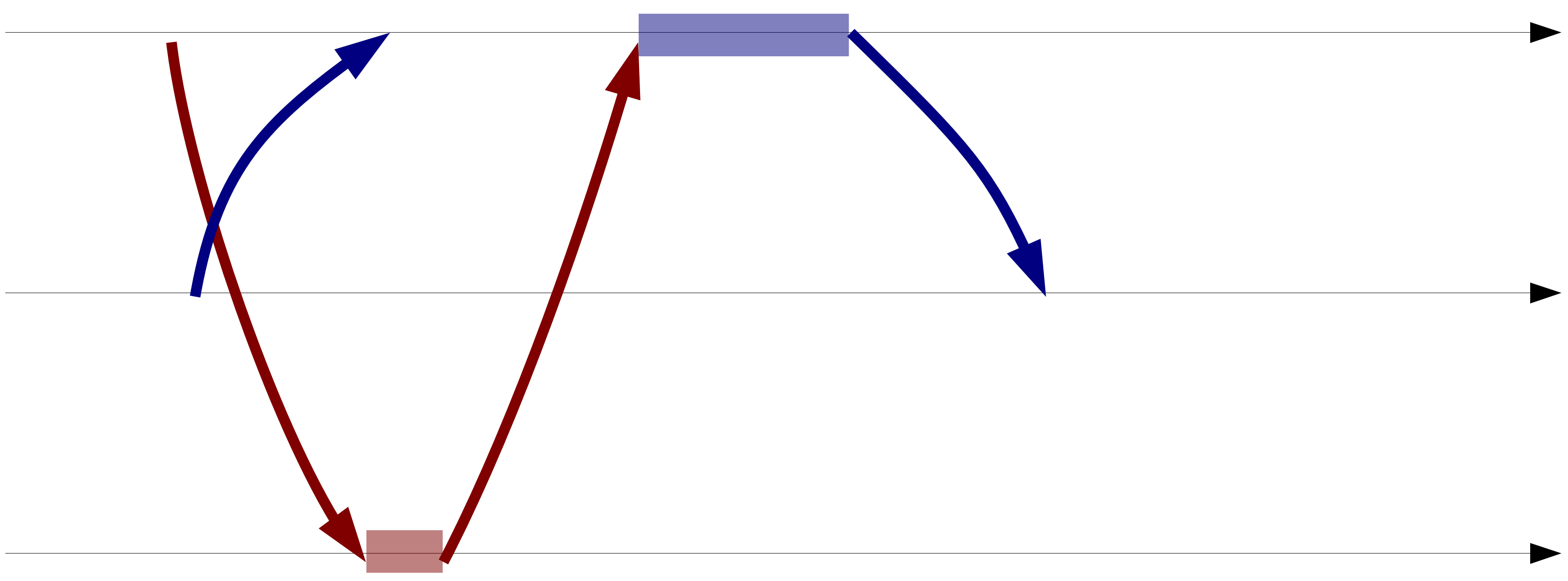


Example: Distributed deadlock

- Remote invocation
- All processes request and reply to invocations
- A mutex is held while invoking remotely or handling remote invocations
- Distributed deadlock possible when multiple processes invoke each other

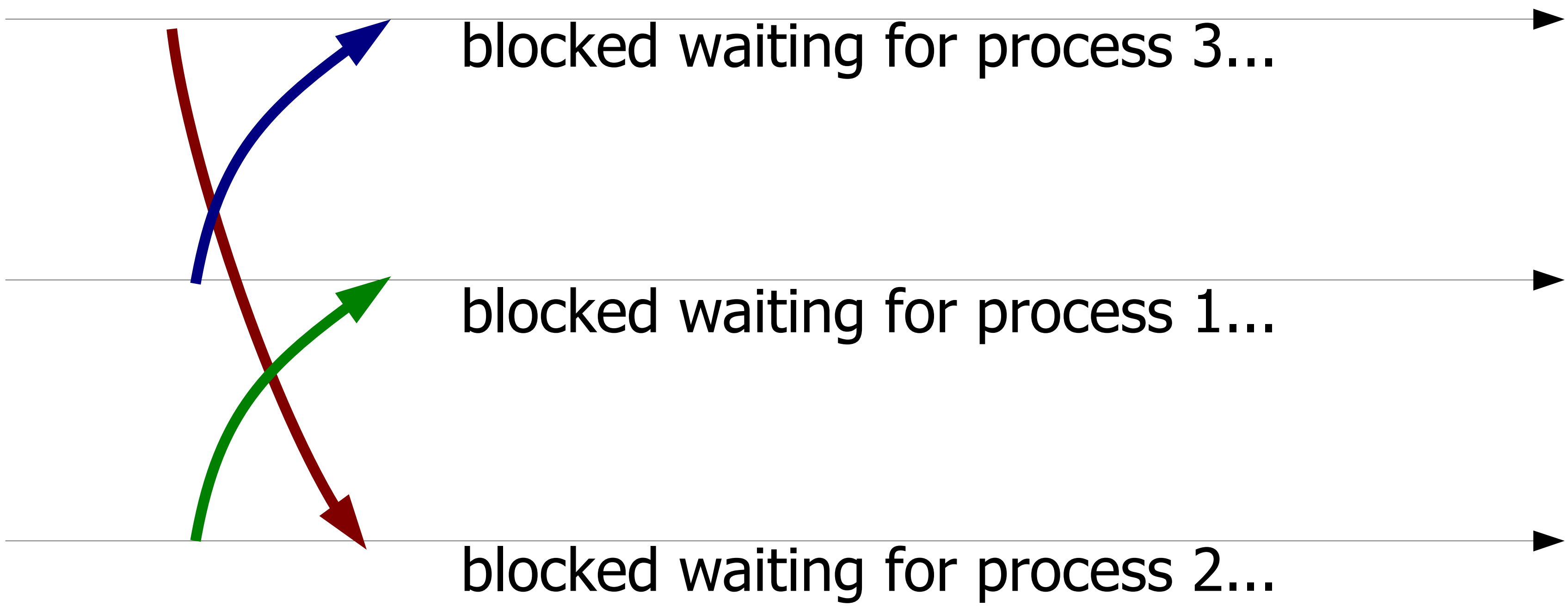
Example: Distributed deadlock

- Deadlock-free run:



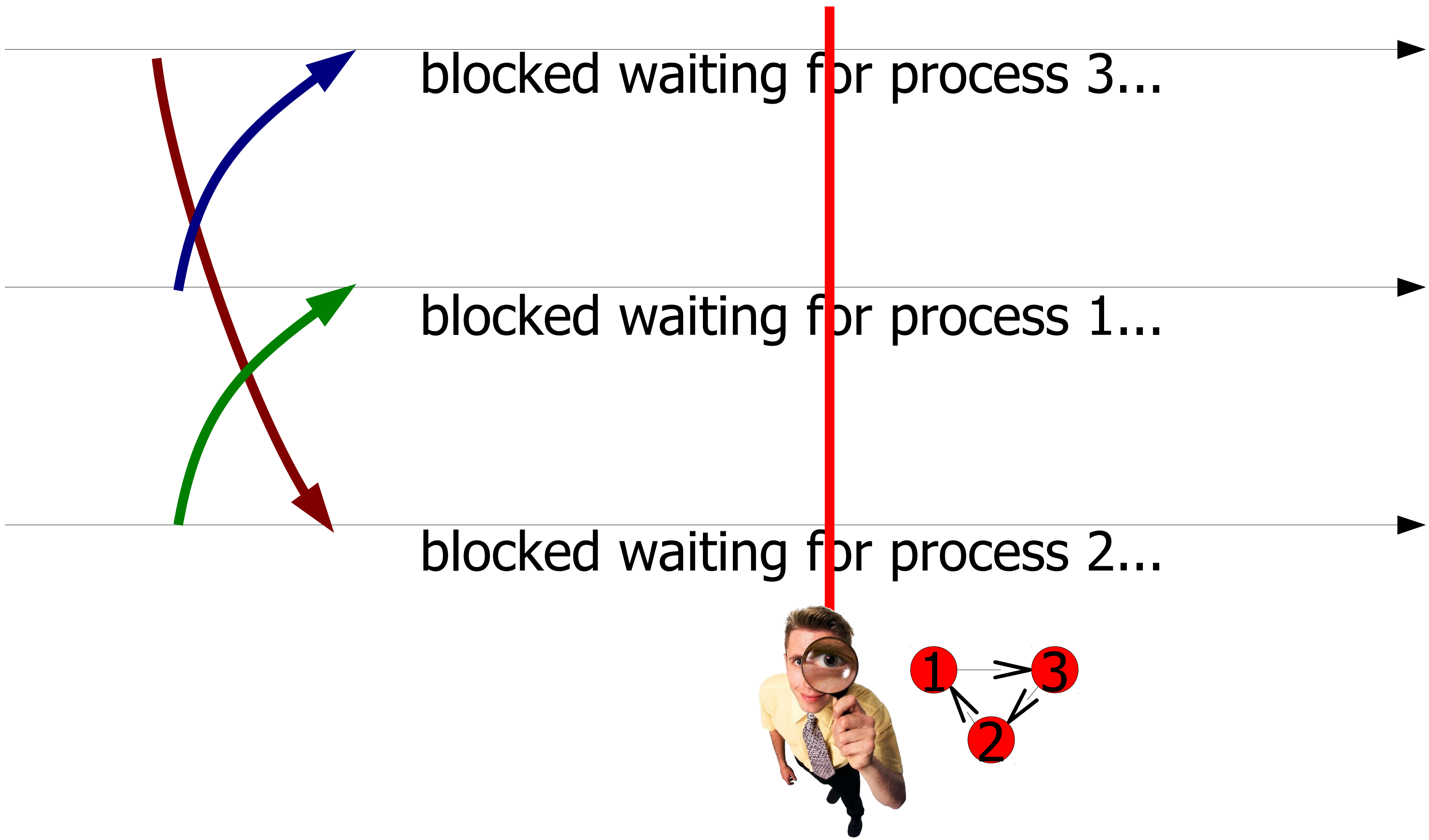
Example: Distributed deadlock

- Distributed deadlock:



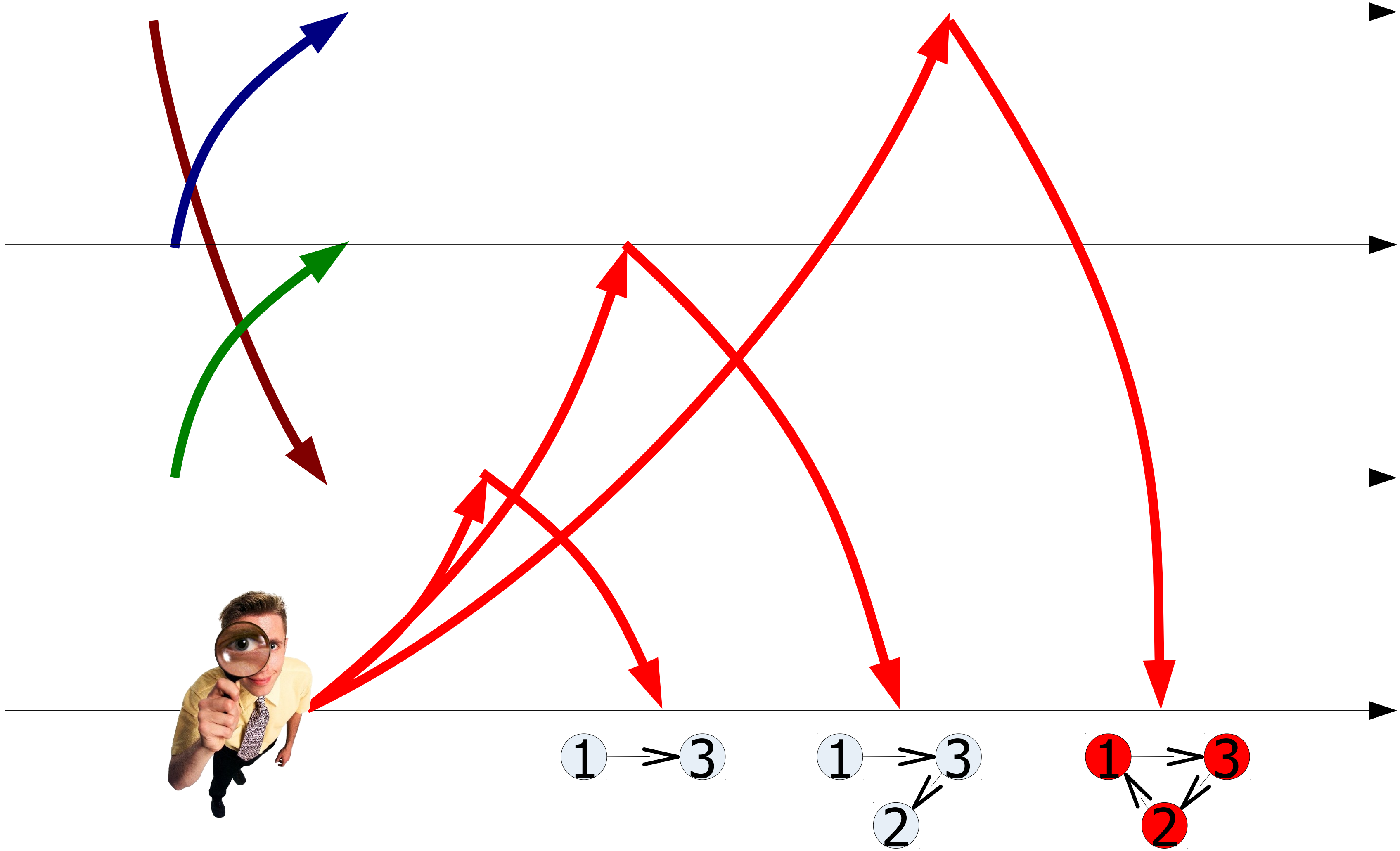
Example: Distributed deadlock

- Instant observation is impossible:



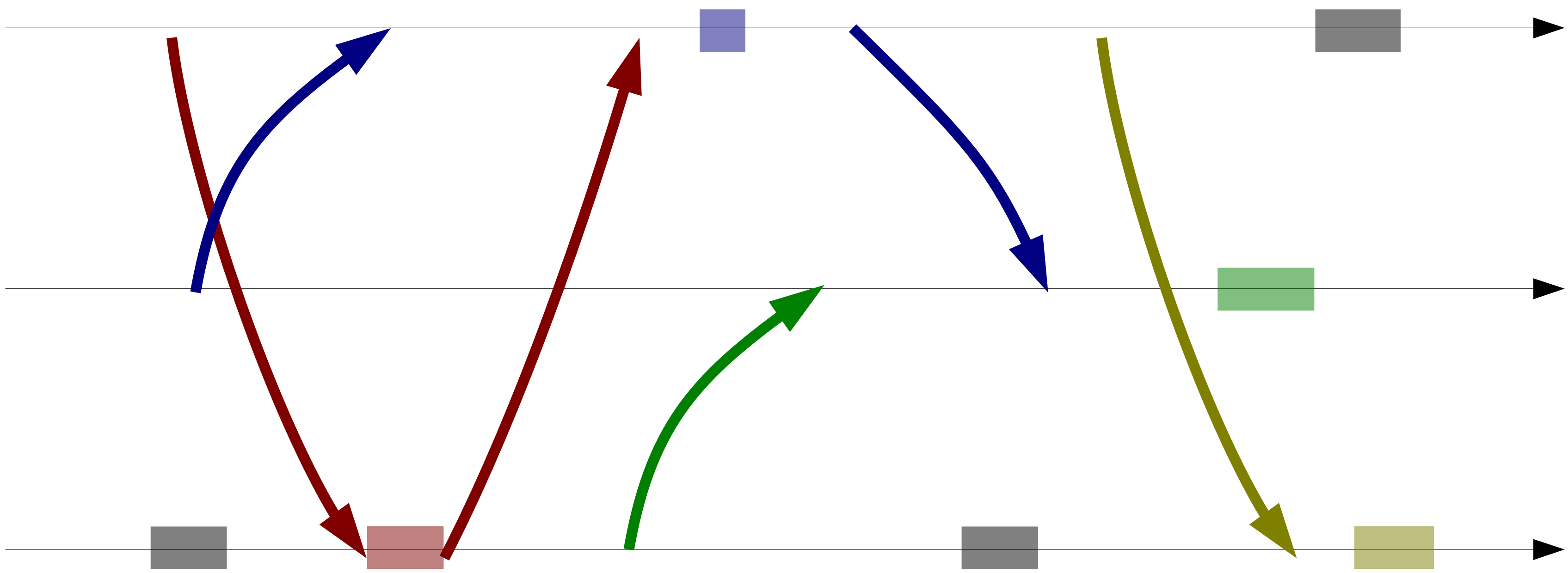
Example: Distributed deadlock

- Deadlock detection with a "wait for" graph:



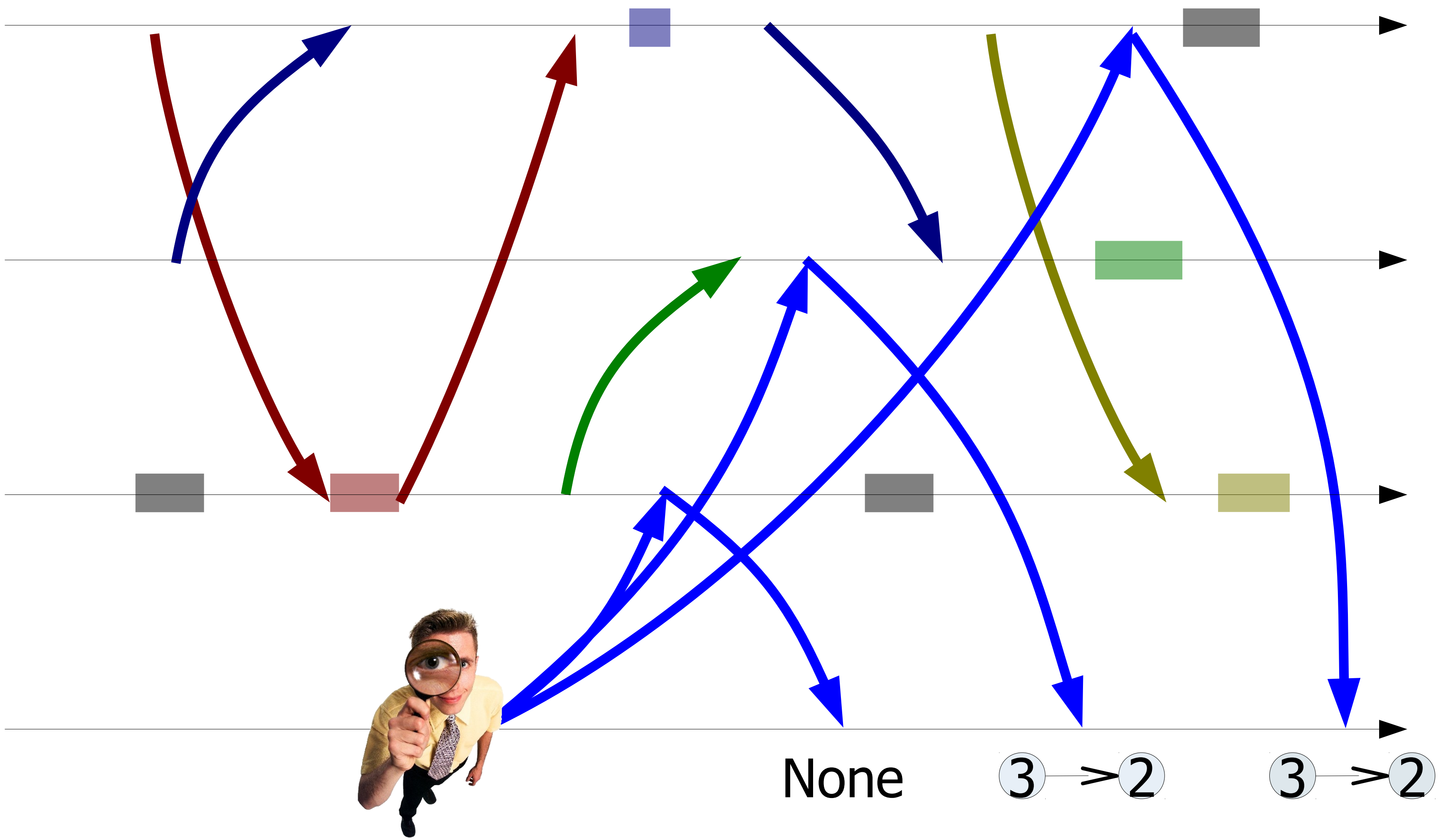
Example: Distributed deadlock

- A more complex deadlock-free run:



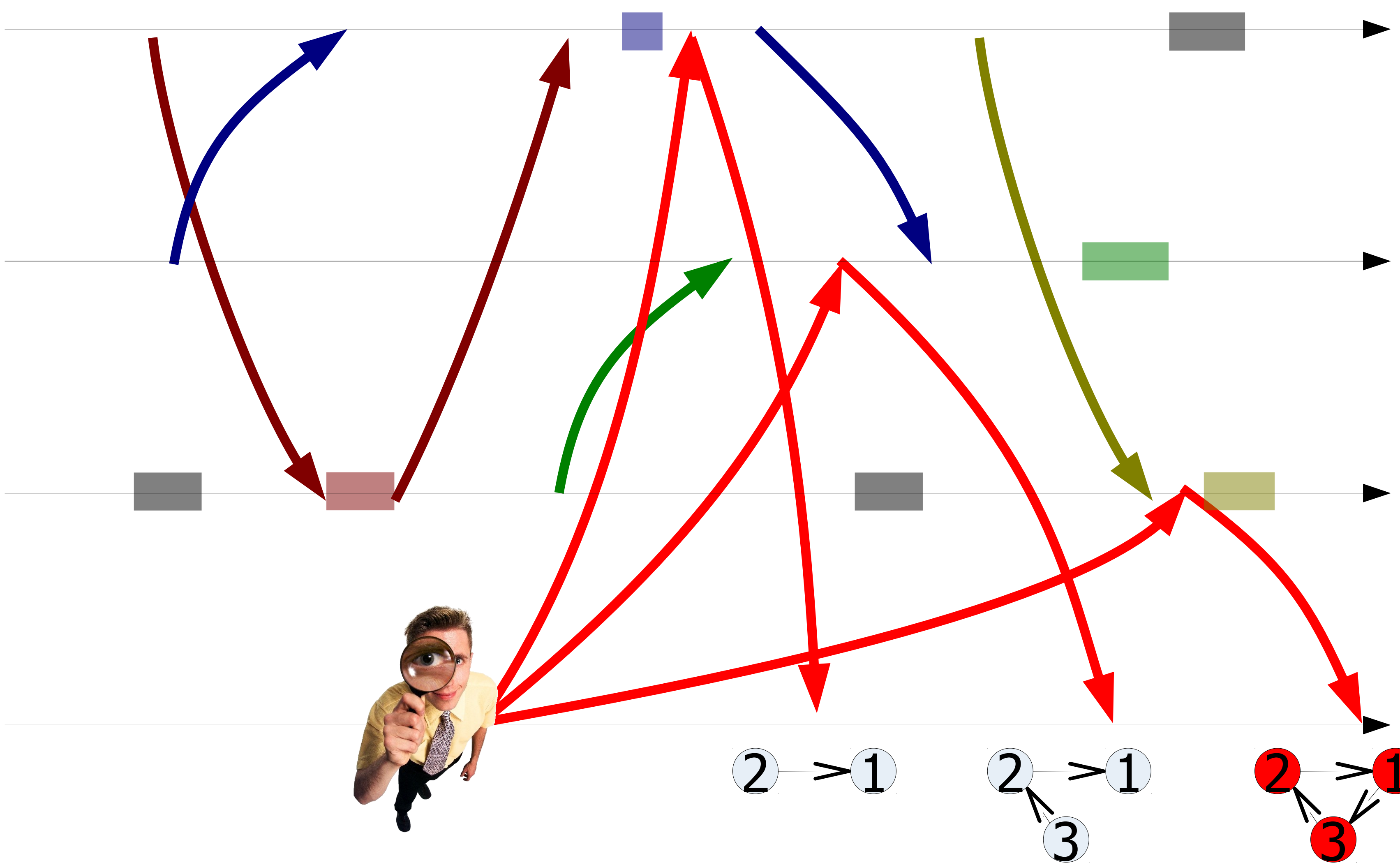
Example: Distributed deadlock

- A deadlock-free WFG:



Example: Distributed deadlock

- A WFG with a ghost deadlock:

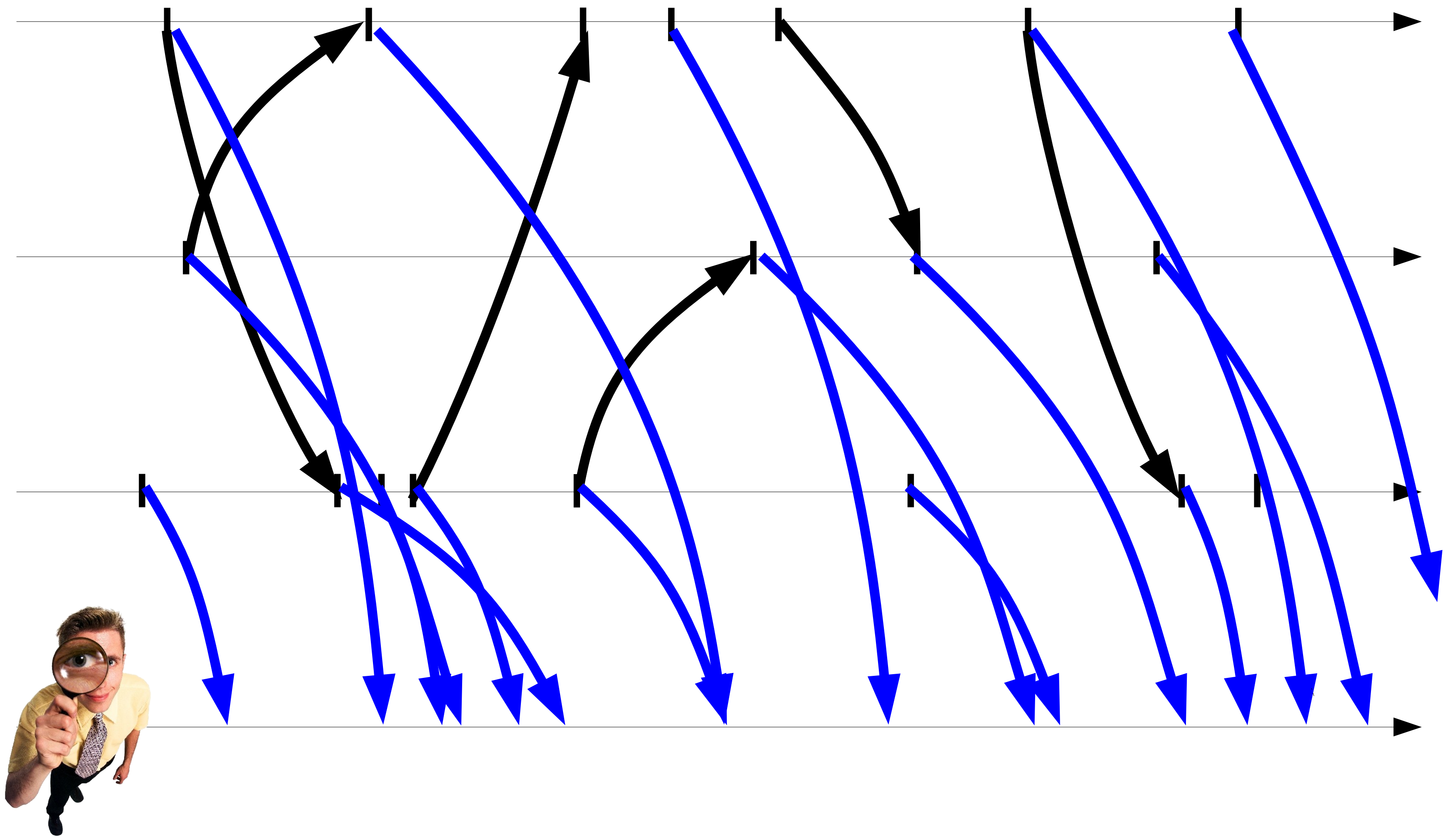


Global Property Evaluation

- All these problems are instances of the Global Property Evaluation (GPE) problem
- Can it be solved in an asynchronous system?
- Methods that can be used? Relative cost?

Passive monitor process

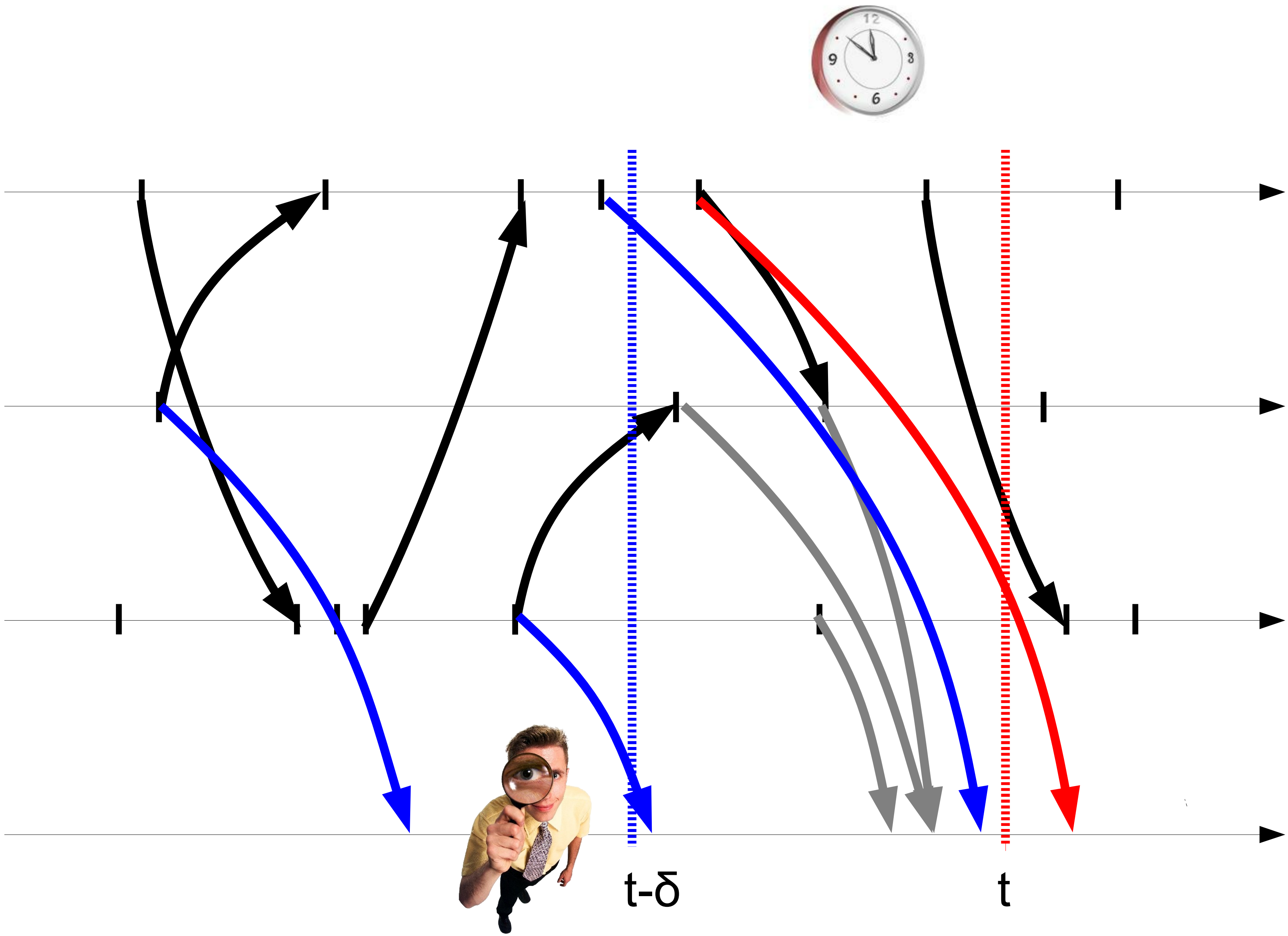
- Report all events to monitor:



First try: Synchronous system

- Global clock, δ upper bound on message delay
- Tag events with real time
- Consider events only up to t-δ
  - With synchronous rounds, this means using messages from the previous round!

First try: Synchronous system



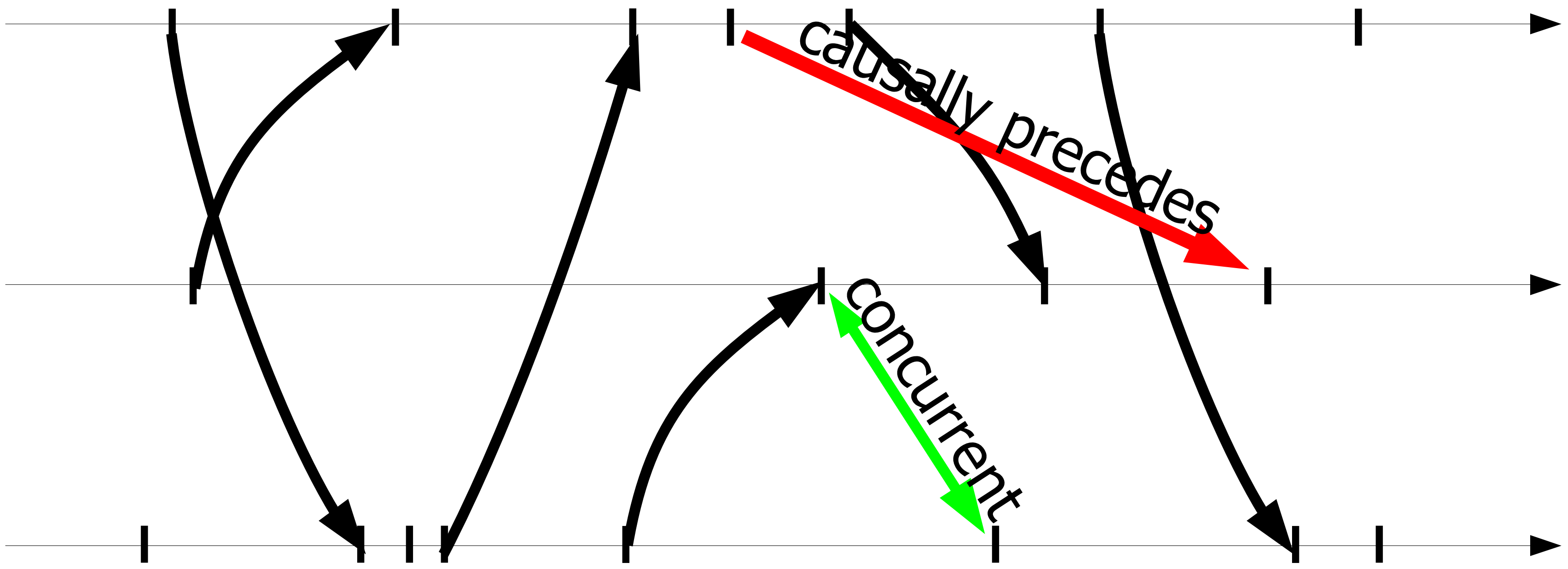
Clock properties

- What properties of a real-time clock make this approach correct?
- RC(i) the time at which i happened

Definition: Causality

- Events i and j are causally related ($i \rightarrow j$) iff:
  - i precedes j in some process p
  - for some m, i=send(m) and j=receive(m)
  - for some k, $i \rightarrow k$ and $k \rightarrow j$ (transitivity)
- Events i and j are concurrent ($i \| j$) iff neither $i \rightarrow j$ or $j \rightarrow i$

Causality



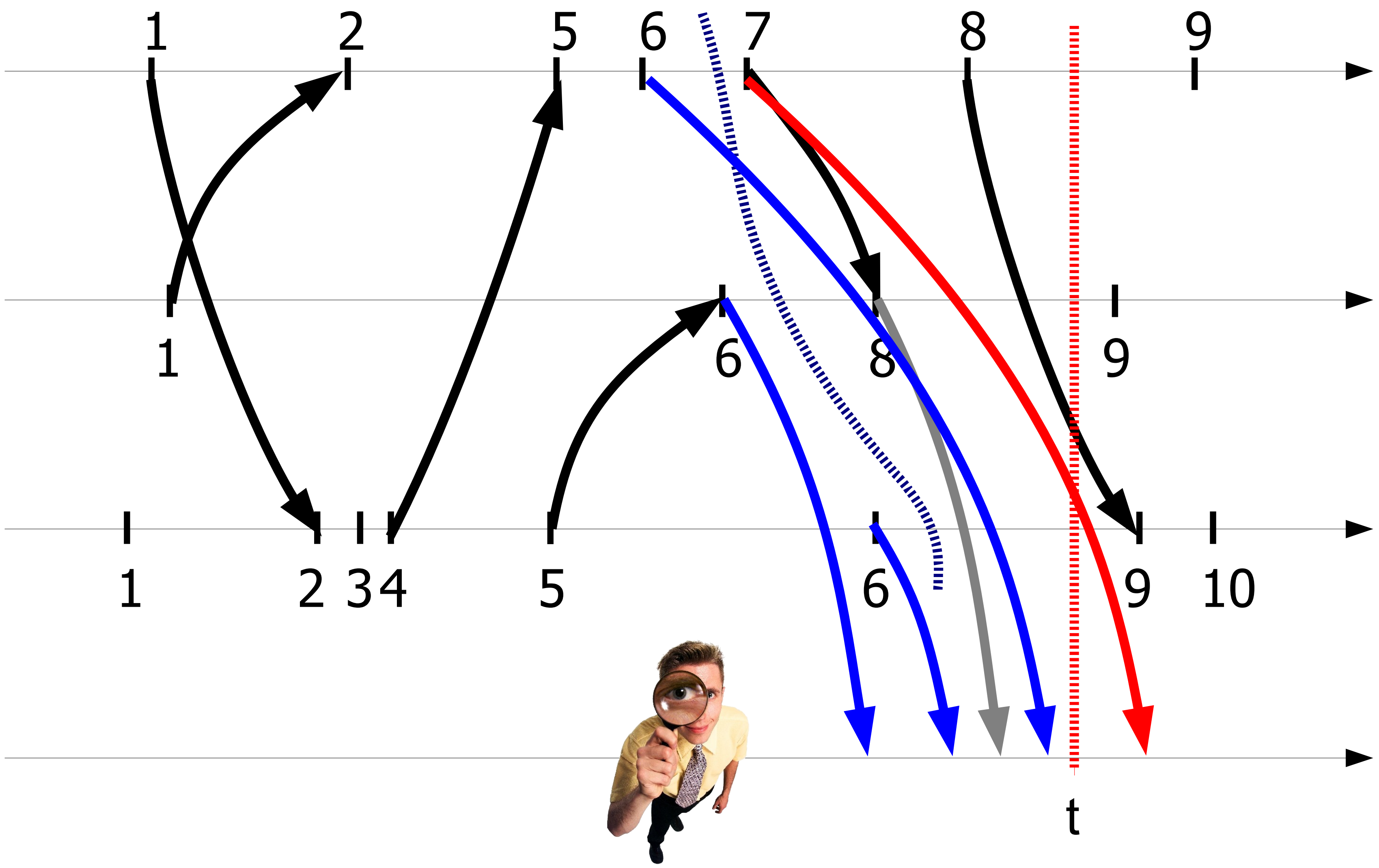
Clock properties

- If $i \rightarrow j$ then $RC(i) < RC(j)$
- For some event j:
  - When we are sure that there is no unknown i such that $RC(i) < RC(j)$
  - Then there is no i such that $i \rightarrow j$
- Can we build a logical clock with the same property?

Second try: Logical clock

- Tag events as follows:
  - Local events: increment counter
  - Send events: increment and then tag with counter
  - Receive events: update local counter to maximum and then increment
- Use FIFO channels
- Consider events only up to the minimum of maximum tags

Second try: Logical clock



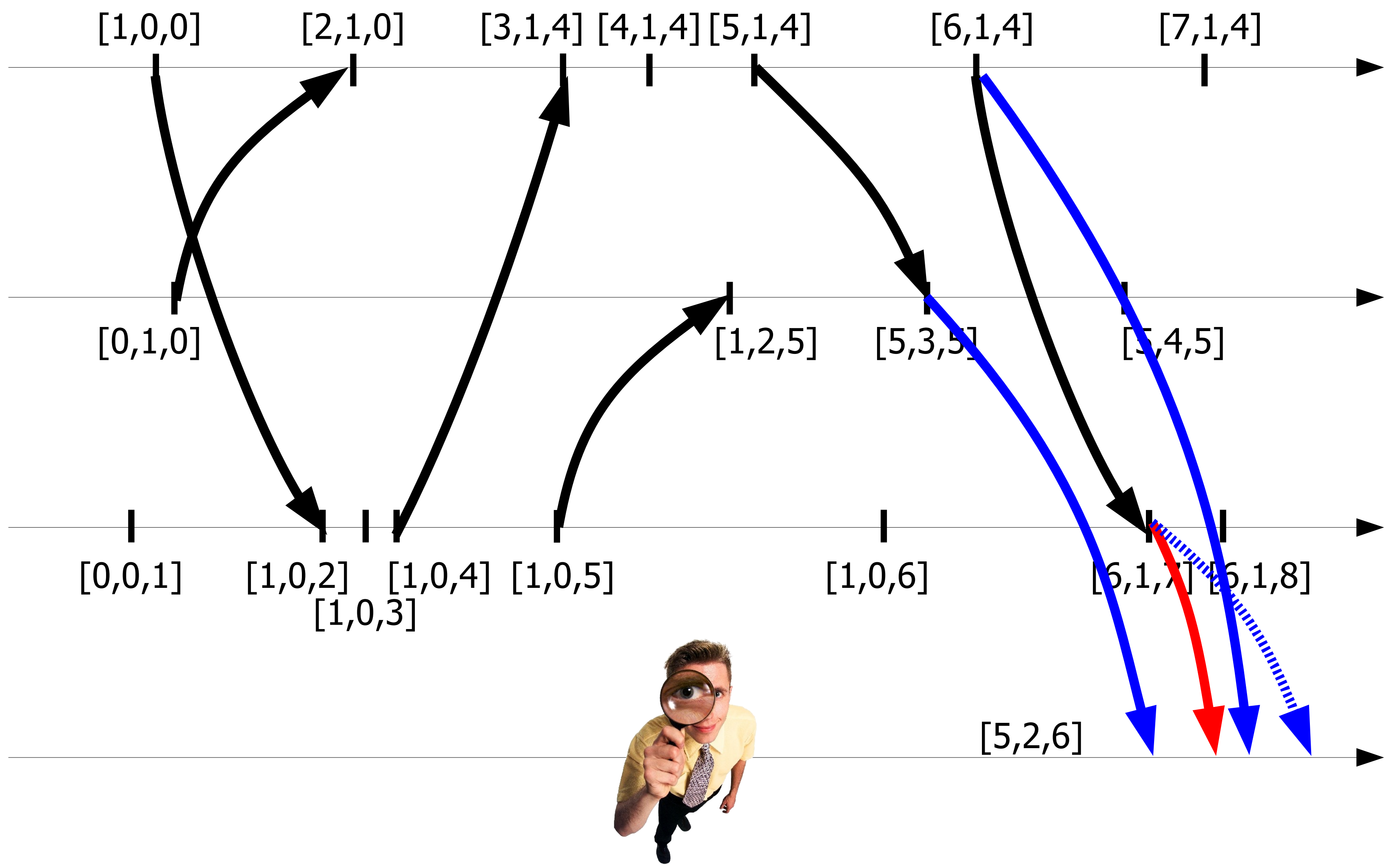
Scalar clocks

- Synchronous system (RC):
  - Delay δ to consistency
- Asynchronous system (LC):
  - Possible unbounded delay to consistency
  - Blocks if some process stops sending messages

Third try: Vector clock

- Tag events with a vector as follows:
  - Local event at i: increment counter i
  - Send event at i: increment counter i and tag with vector
  - Receive event at i: update each counter to maximum and increment counter i

Third try: Vector clock



Causal delivery

- The monitor delivers events as follows:
  - With local vector l[...]
  - For some r[...] from i
  - Wait until:
    - $l[i]+1=r[i]$
    - For all $j \neq i$: $r[i] \leq l[i]$
- The monitor is always in a consistent cut
- Blocking can be avoided by forwarding past messages

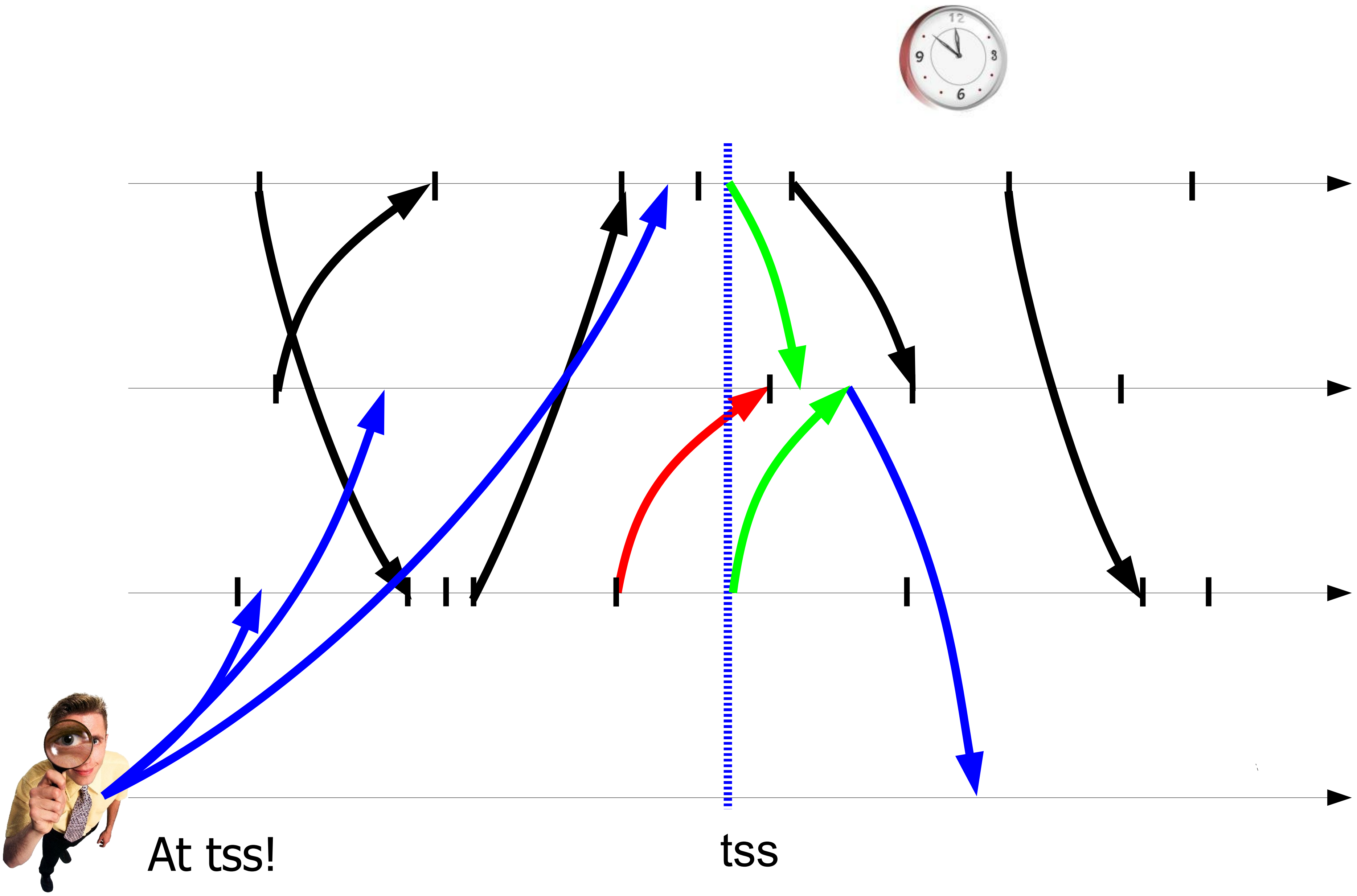
No reporting to monitor process

- Reporting all events to a monitor causes a large overhead
- Can a query be issued at some point in time?

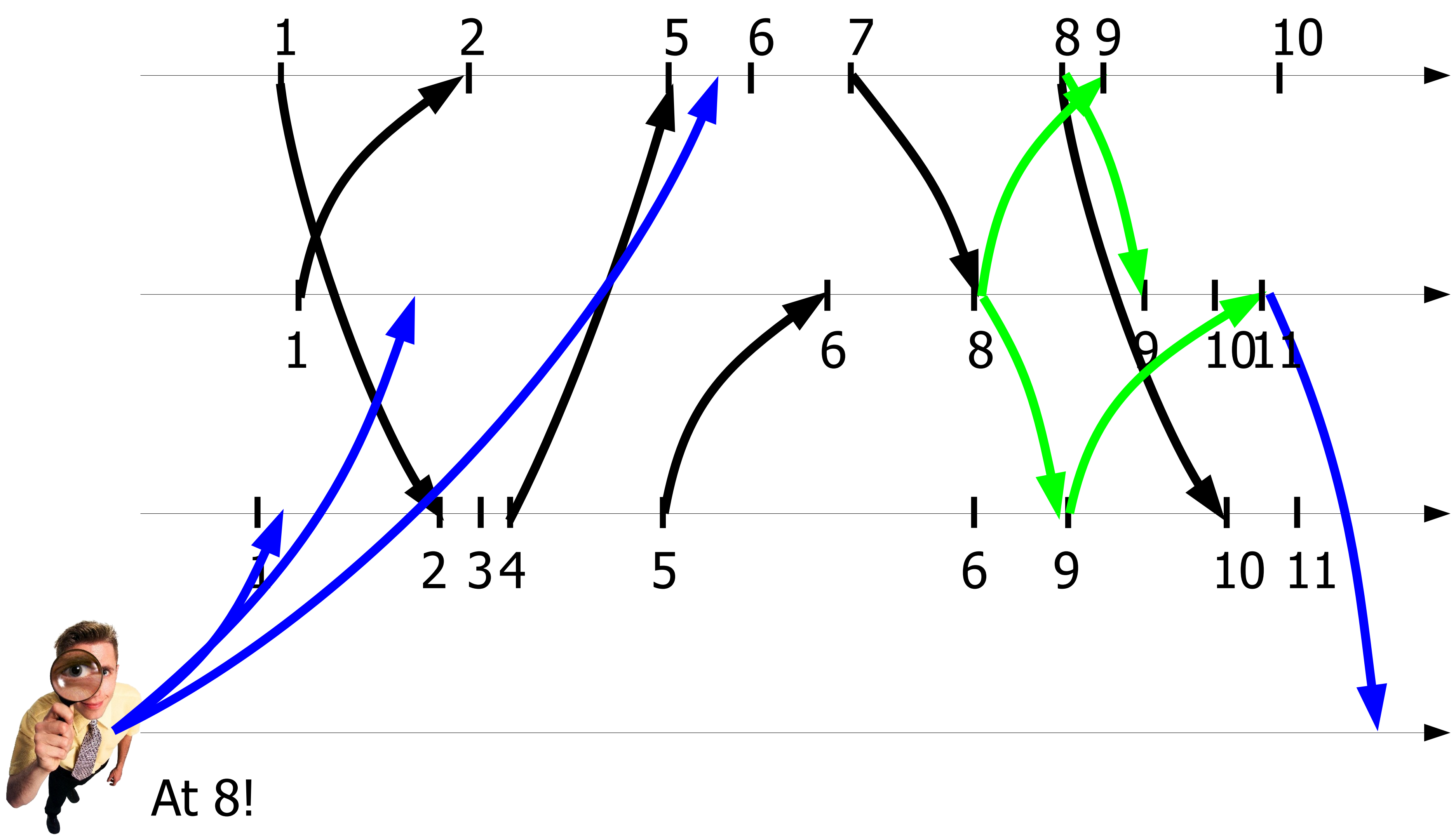
Fourth try: No reporting, synchronous

- Monitor broadcasts tss in the future
- At tss, each process:
  - Records state
  - Sends messages to all others
  - Starts recording messages until receiving a message with RC > tss
- After stopping, sends all data to monitor

Fourth try: No reporting, synchronous



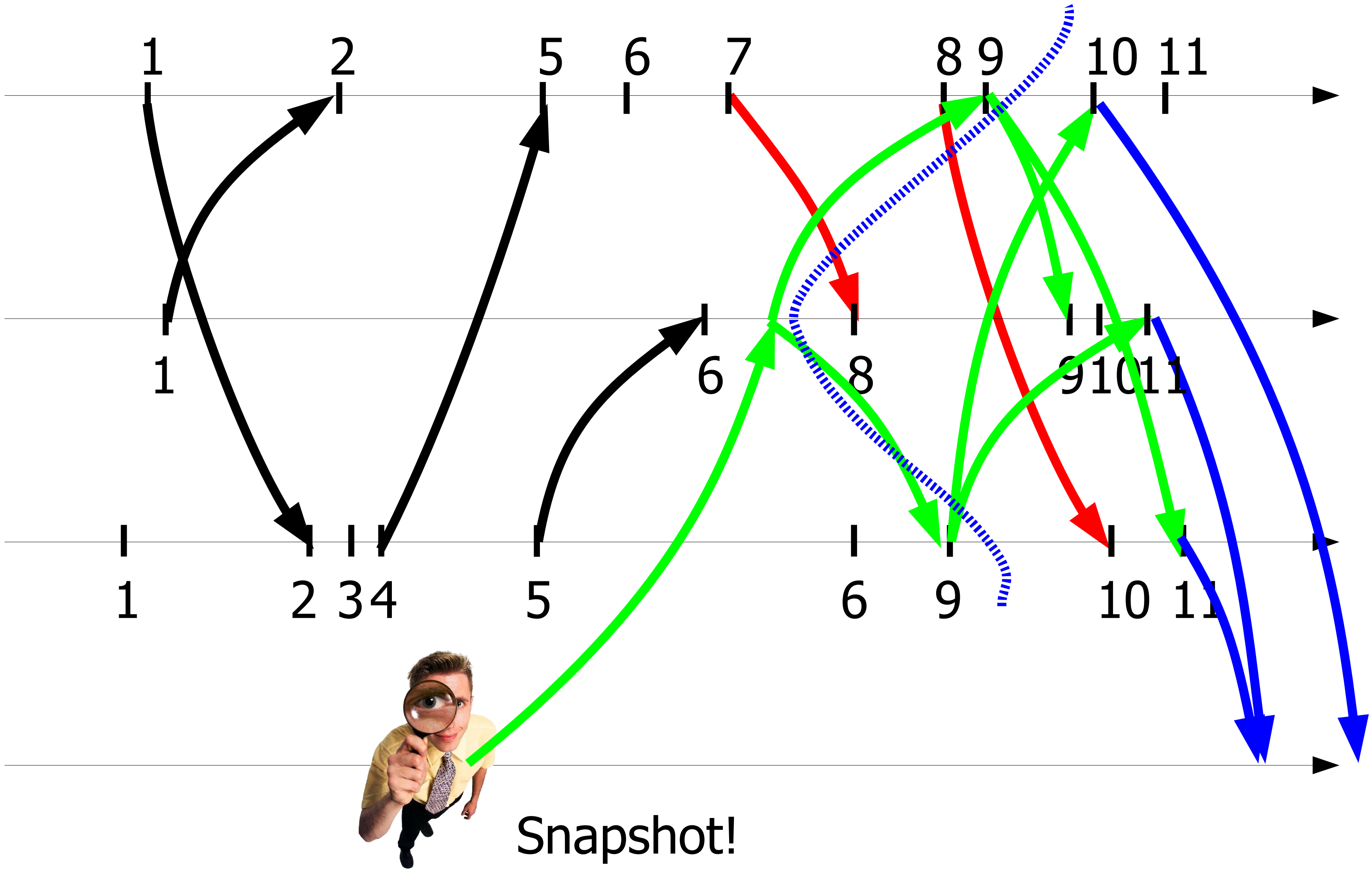
Fifth try: No reporting, logical clock



Chandy and Lamport

- Send a “Snapshot” message to some process
- Upon receiving for the first time:
  - Records state
  - Relays “Snapshot” to all others
  - Starts recording on each channel until receiving “Snapshot”
- Send all data to monitor

Chandy and Lamport



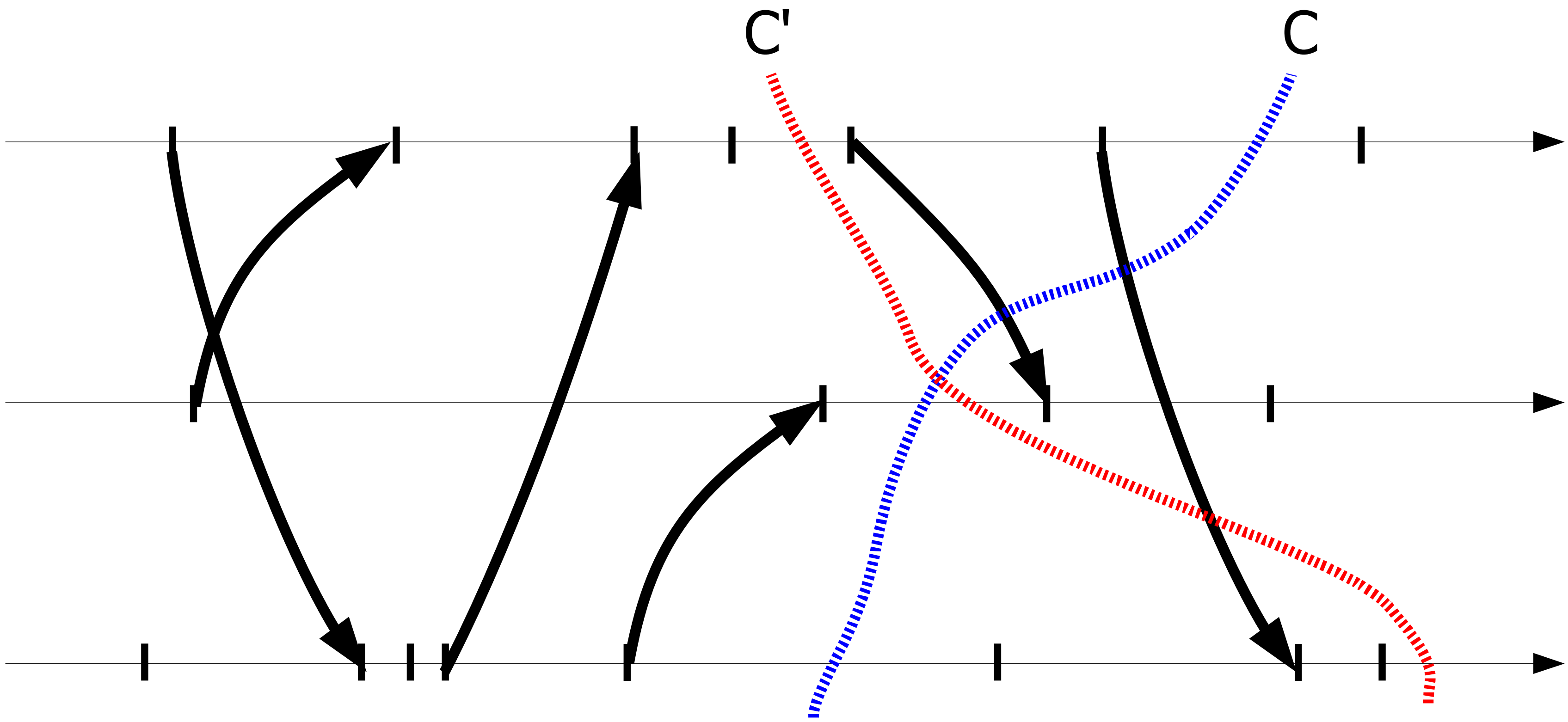
Global Property Evaluation

- GPE requires no gaps in observed history, regarding causality
- What properties can be evaluated?

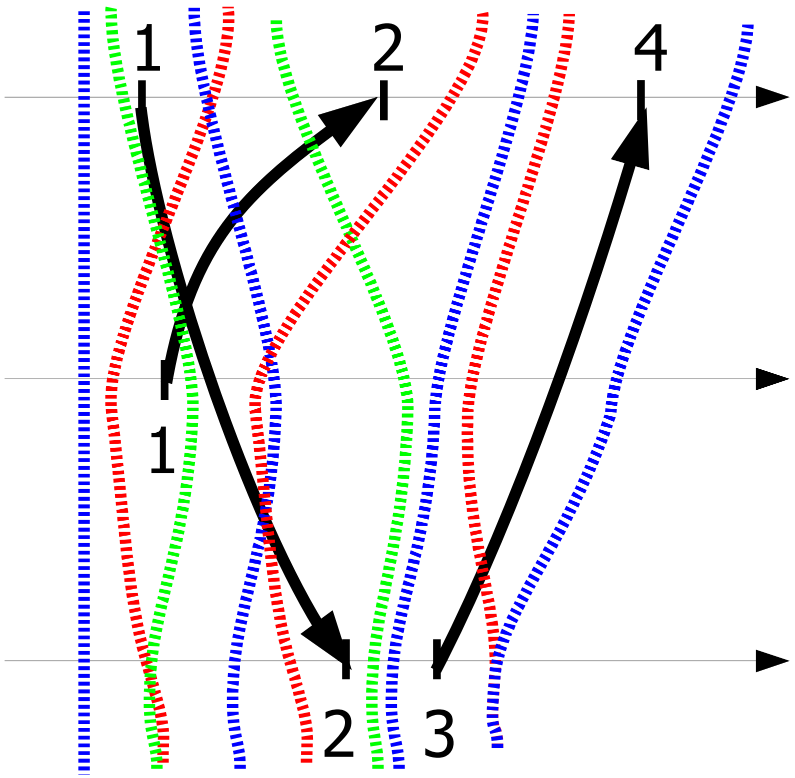
Cuts and consistency

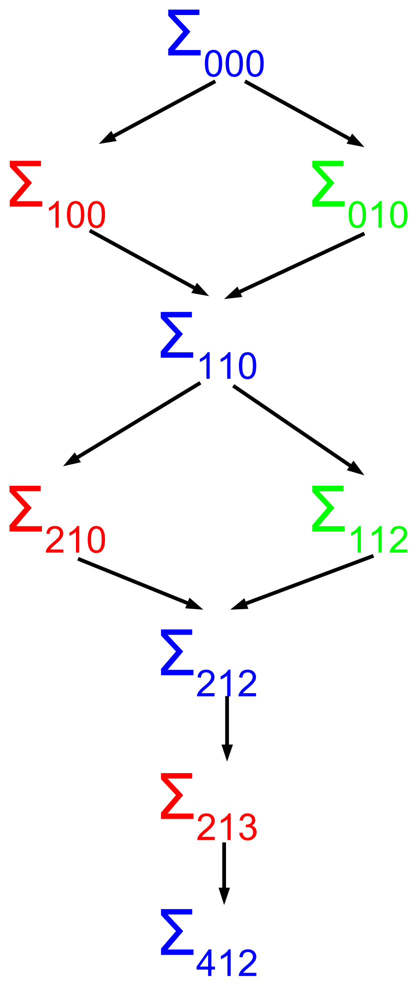
- A cut is the union of prefixes of process history
- A consistent cut includes all causal predecessors of all events in the cut
- Intuitive methods:
  - If a cut is an instant, there are no messages from the future
  - In the diagram, no arrows enter the cut
  - All events in the frontier are concurrent

Consistent cuts



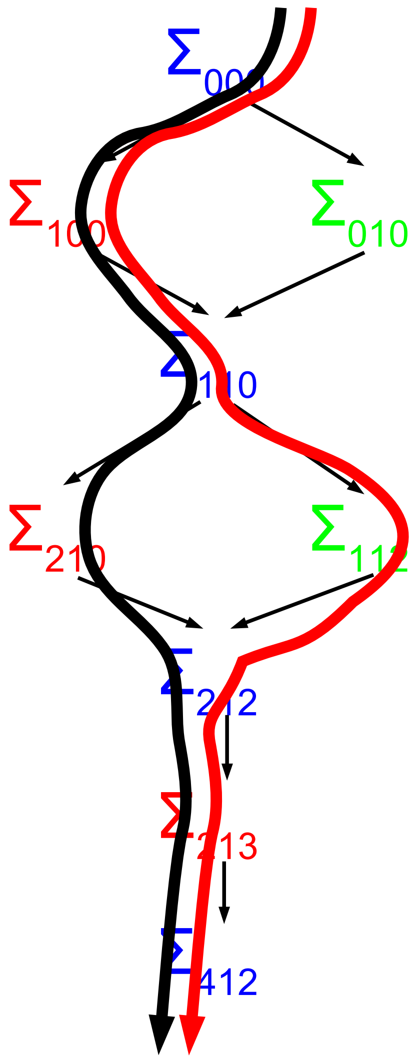
Consistent global states





Consistent global states

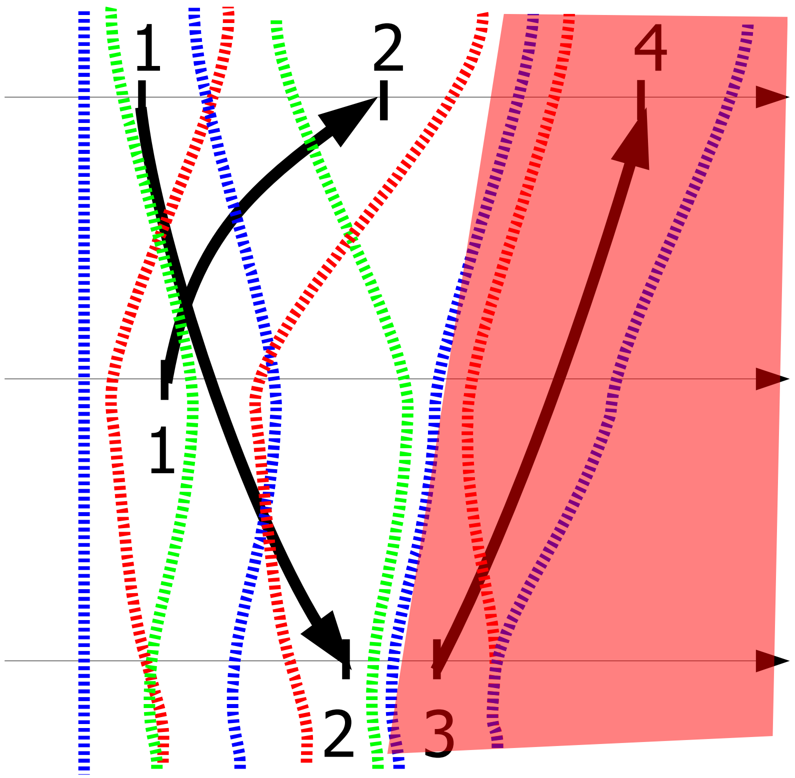
- Includes the true sequence of states in the system
- An observer within the system cannot deny any of the possible paths

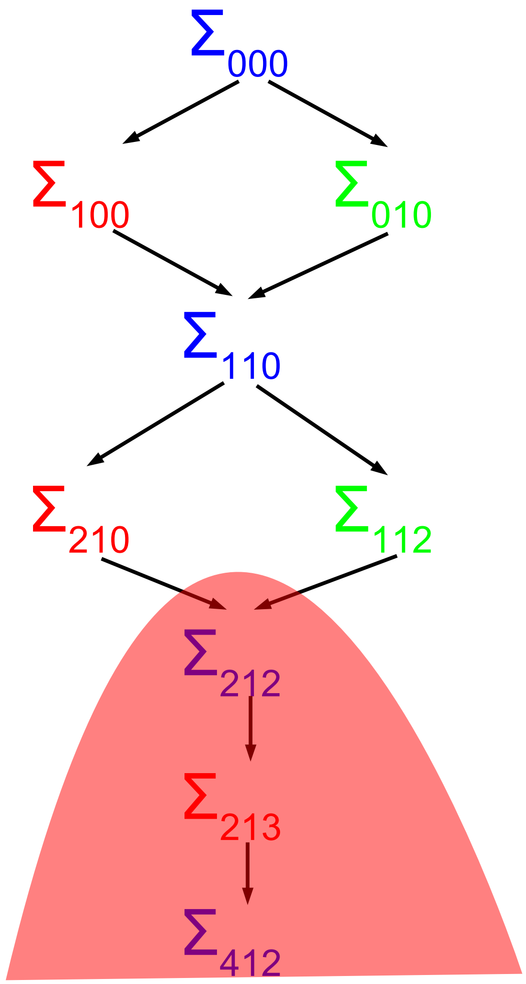


Stable predicates

- Once true, always true
- Examples:
  - Deadlock detection
  - Termination
  - Loss of token
  - Garbage collection
- Can be evaluated periodically on snapshots

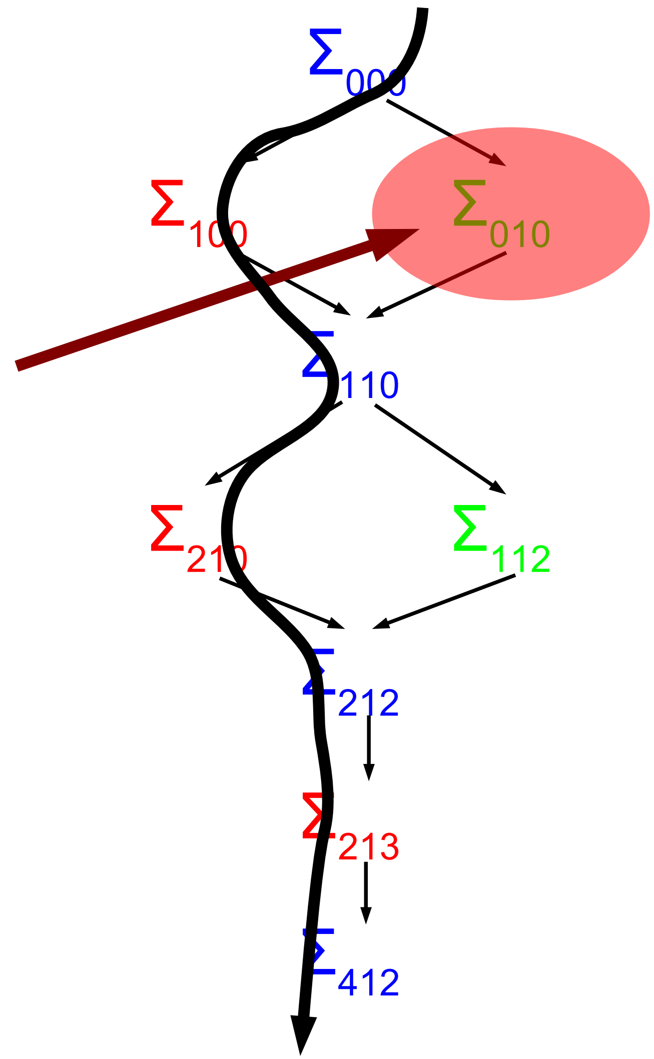
Stable predicates





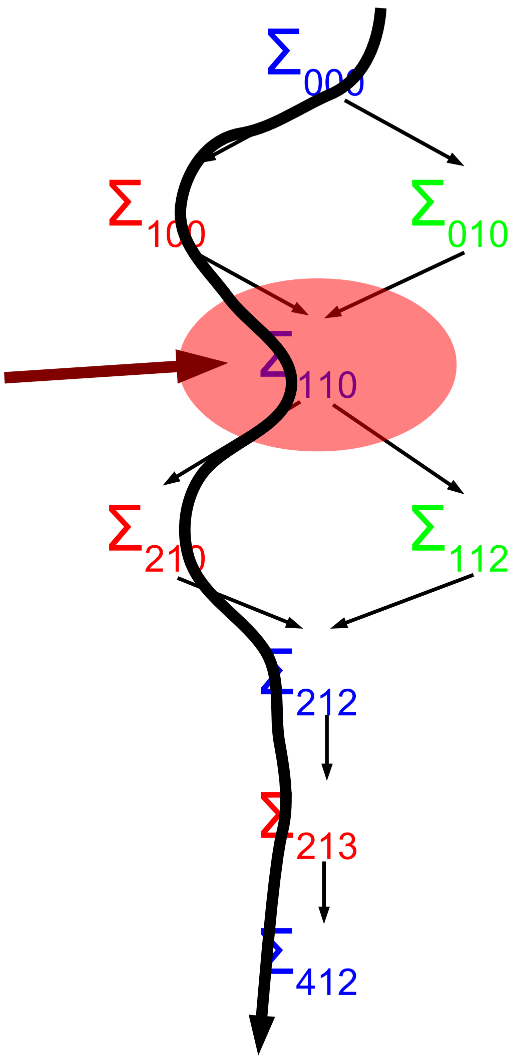
Non-stable predicates

- True in a subset of observable states
- Some are possibly true: an observer in the system cannot deny having been true
- The predicate does not hold on some paths



Non-stable predicates

- True in a subset of observable states
- Some are <u>definitely true</u>: an observer in the system is sure of having been true
- The predicate holds on all possible paths



Non-stable predicates

- Examples:
  - Total size of queues in the system
  - Number of messages in transit
  - Amount of memory used
- Can be detected by full monitoring of all (relevant) events

Conclusion

- Second goal achieved:
  - Causality
  - Global predicate evaluation