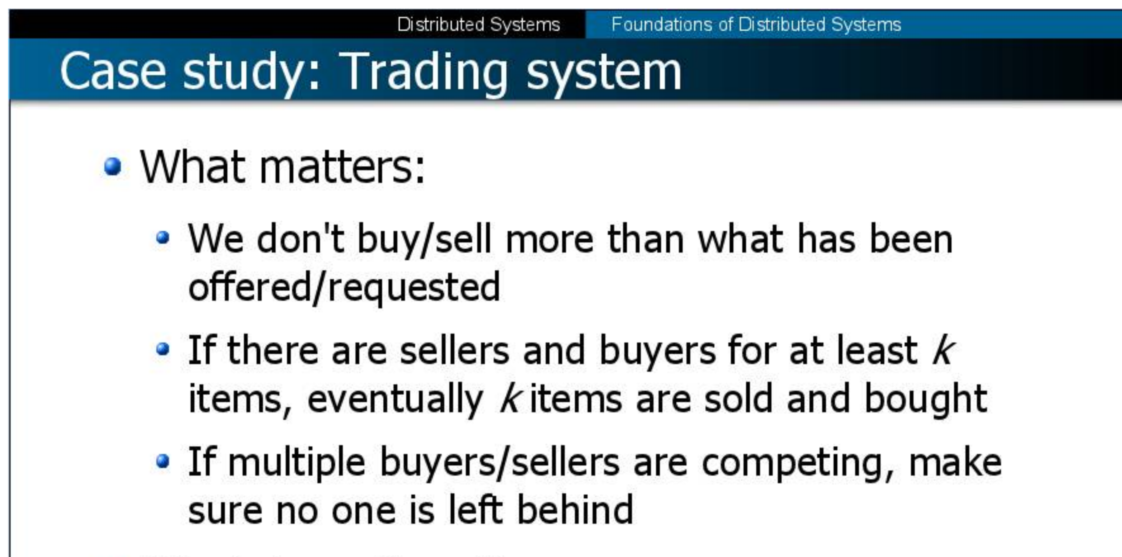
Goal

- Within the system, make sure the distributed program behaves correctly
- In our case study:
  - Discover if the system is stopped (e.g. all are selling)
  - Discover if the amount for sale exceeds all available (i.e. "shorting")

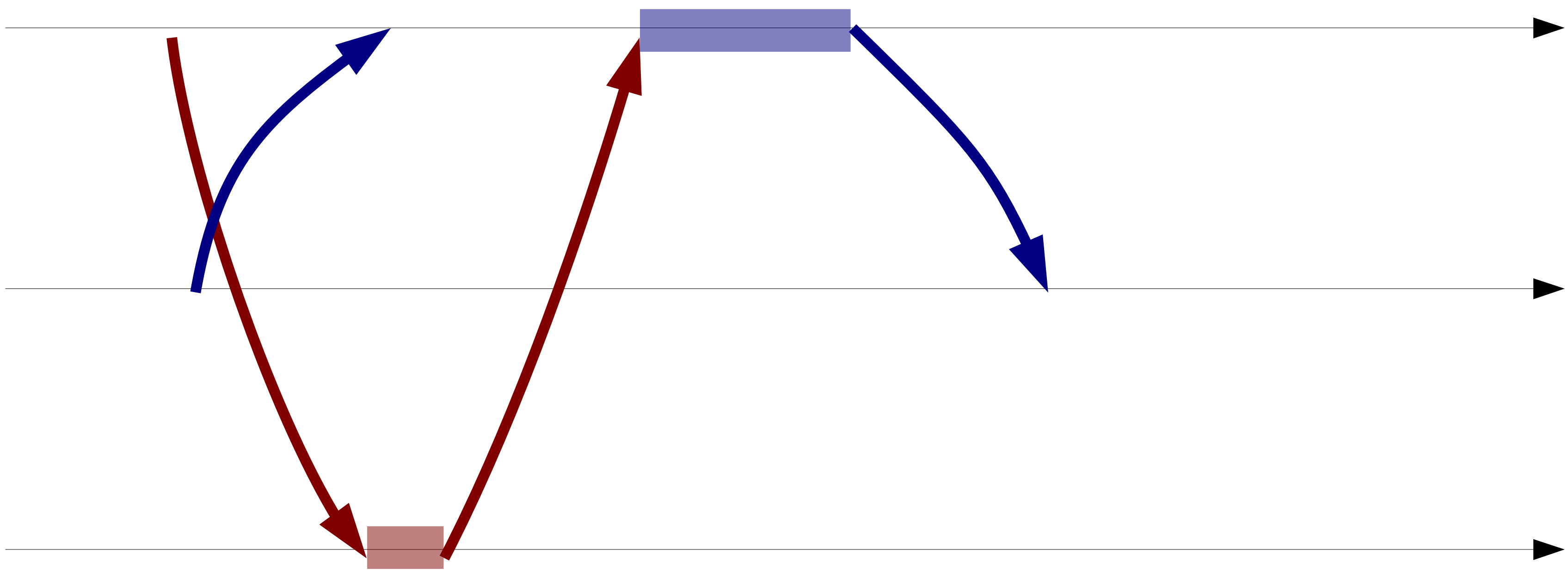


Example: Distributed deadlock

- An example with 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

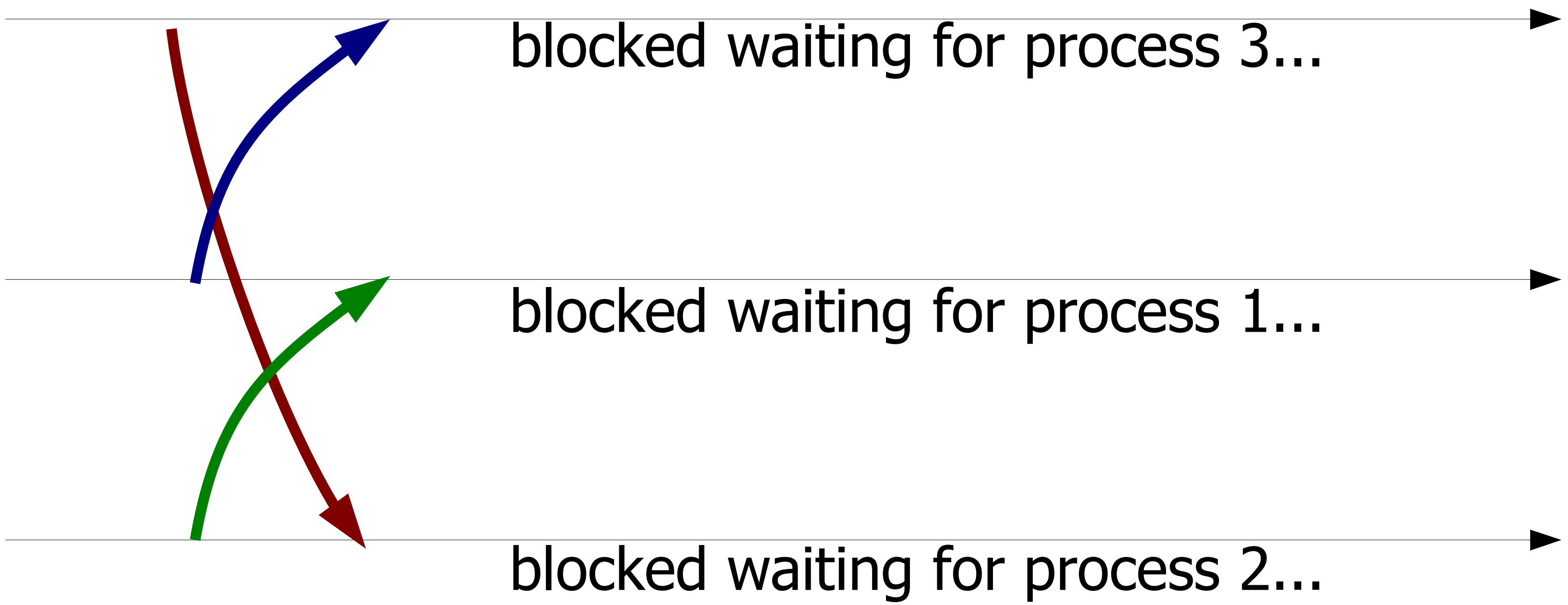
Example: Distributed deadlock

- Deadlock-free run:



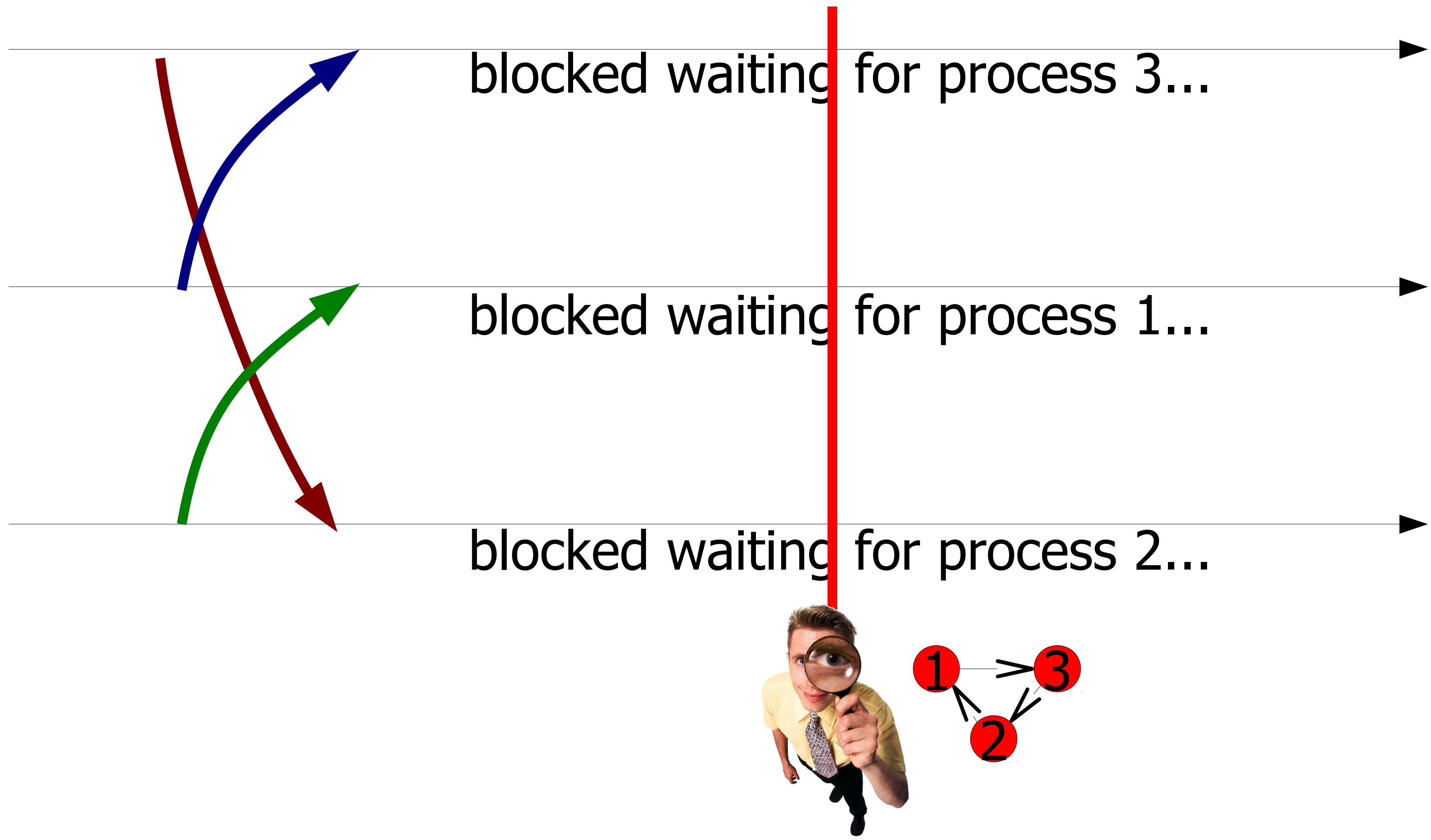
Example: Distributed deadlock

- Distributed deadlock:



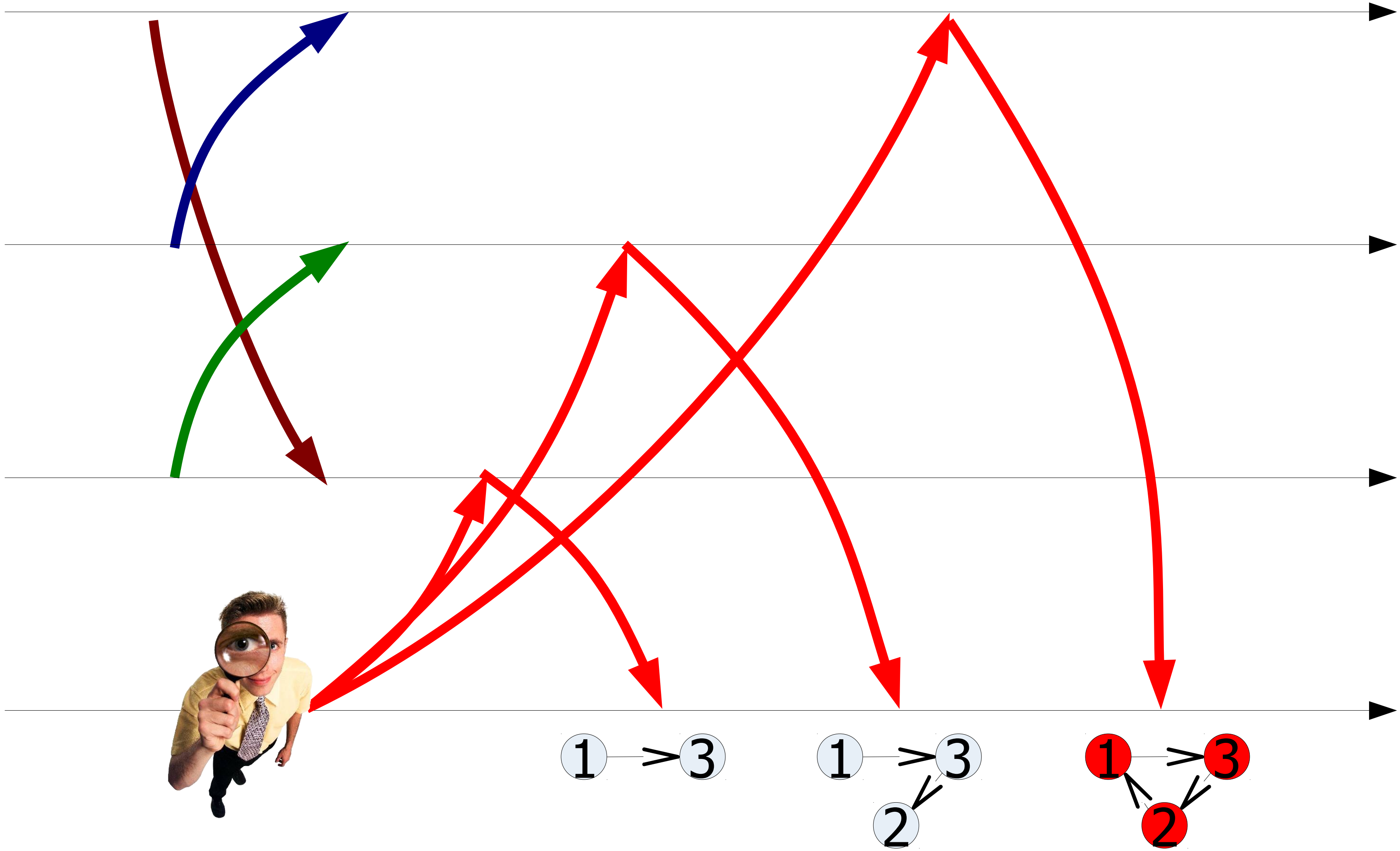
Example: Distributed deadlock

- Instant observation is impossible:



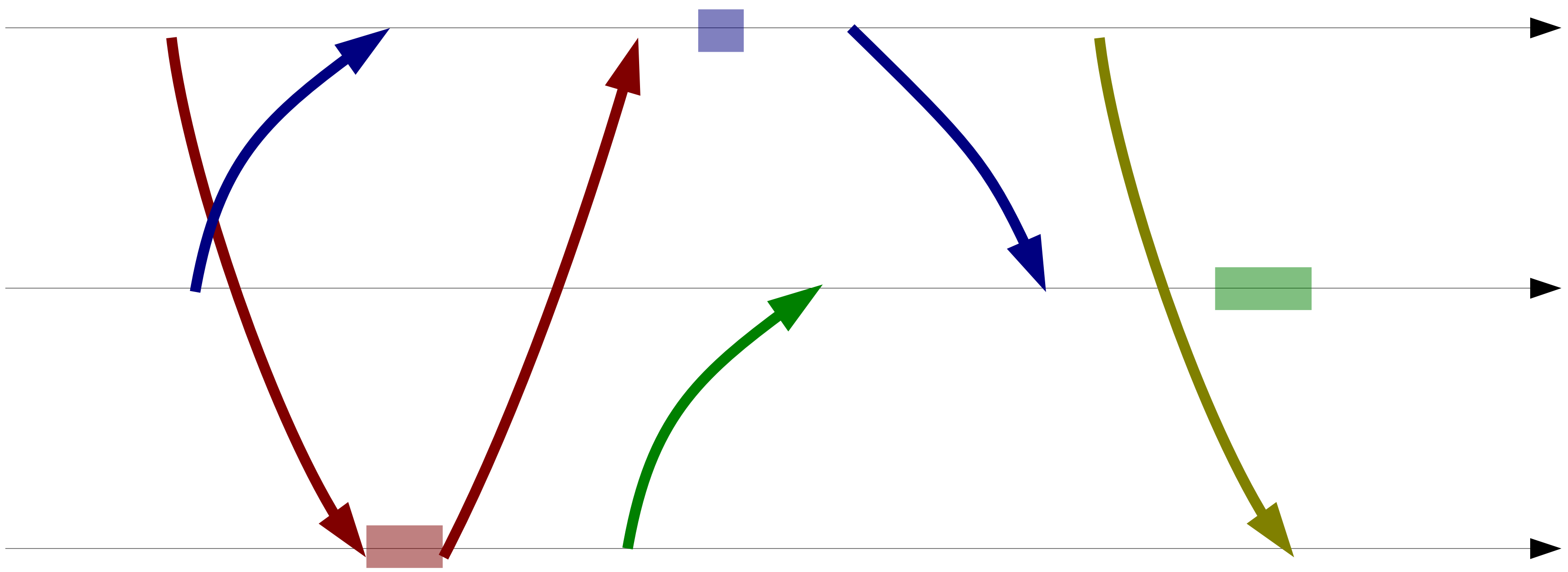
Example: Distributed deadlock

- Deadlock detection with a "wait for" graph:



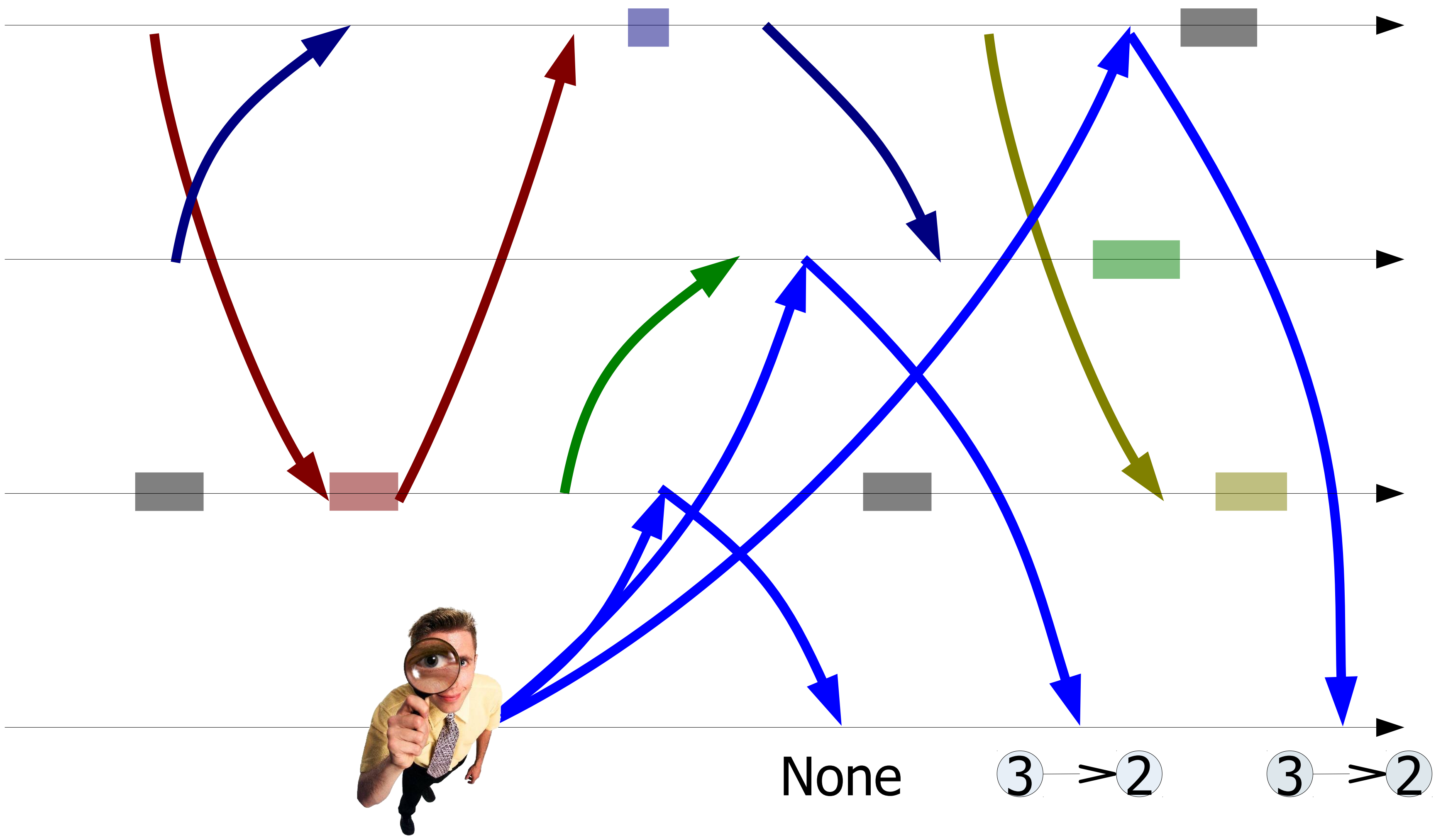
Example: Distributed deadlock

- A more complex deadlock-free run:



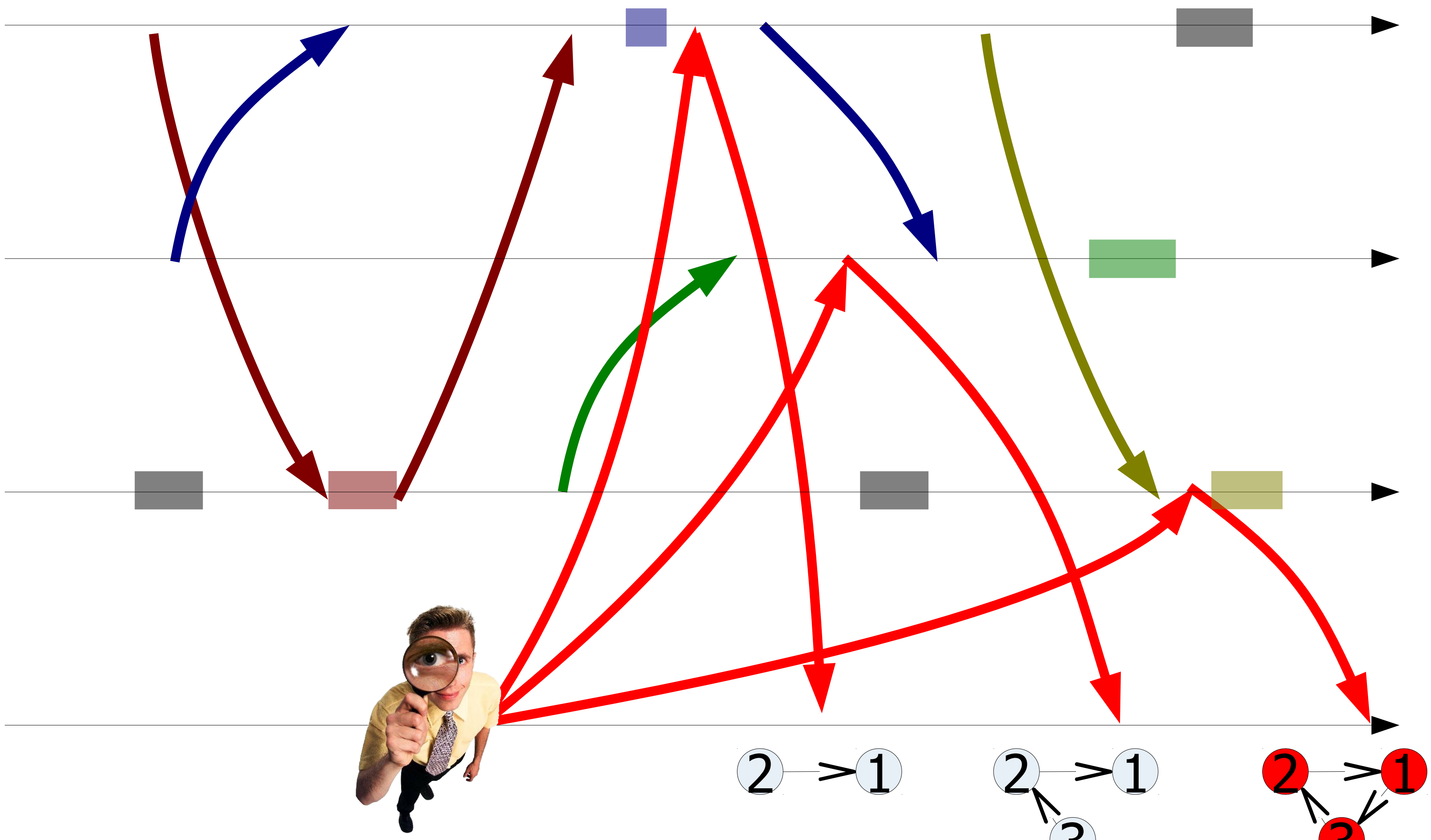
Example: Distributed deadlock

- A deadlock-free WFG:



Example: Distributed deadlock

- A WFG with a ghost deadlock:

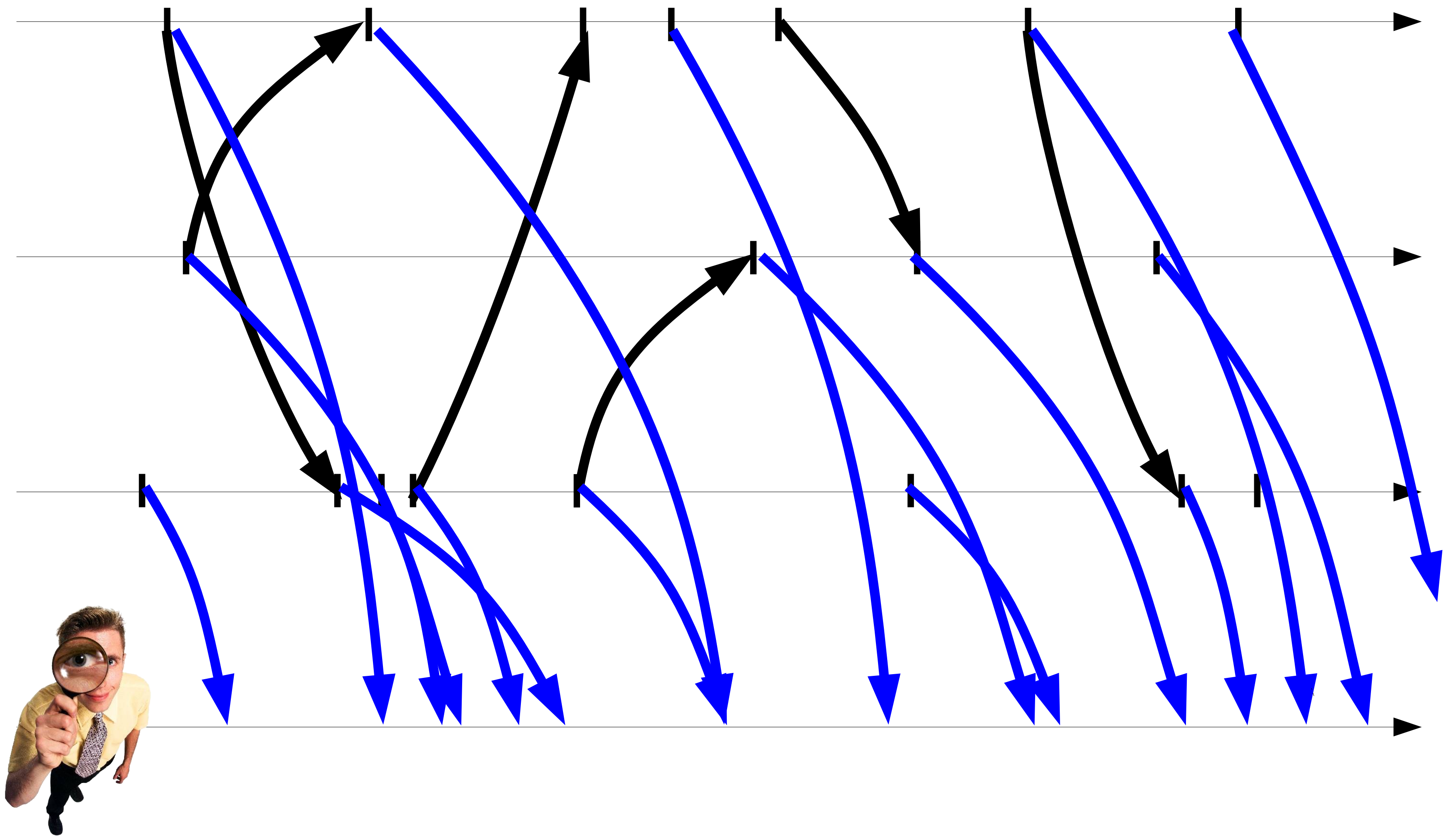


Global Property Evaluation

- This problem is an instance 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

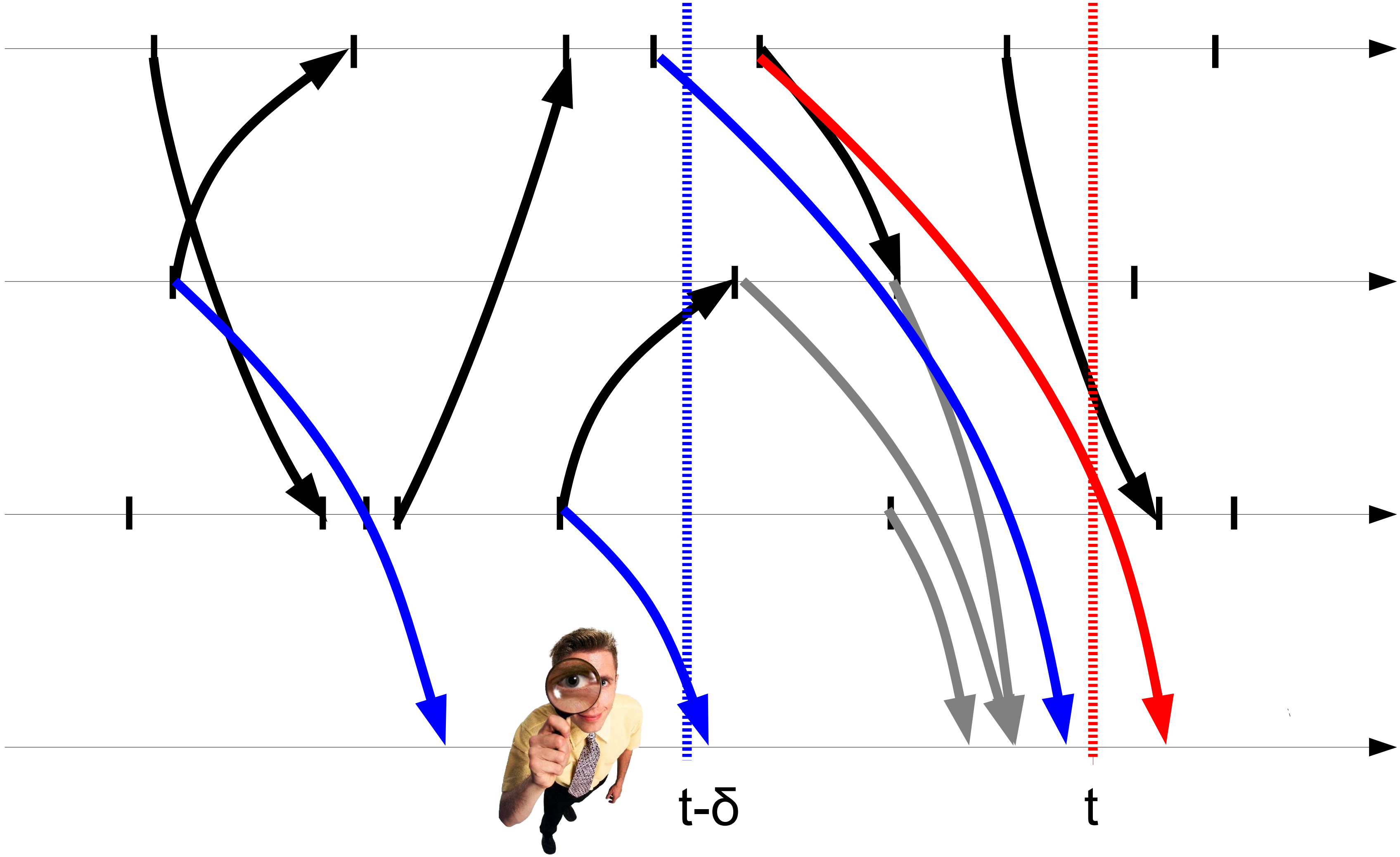
- Report all events to monitor:



First try: Synchronous system

- Global clock, δ upper bound on message delay
- Tag events with real time
- Consider events only up to t-δ

First try: Synchronous system



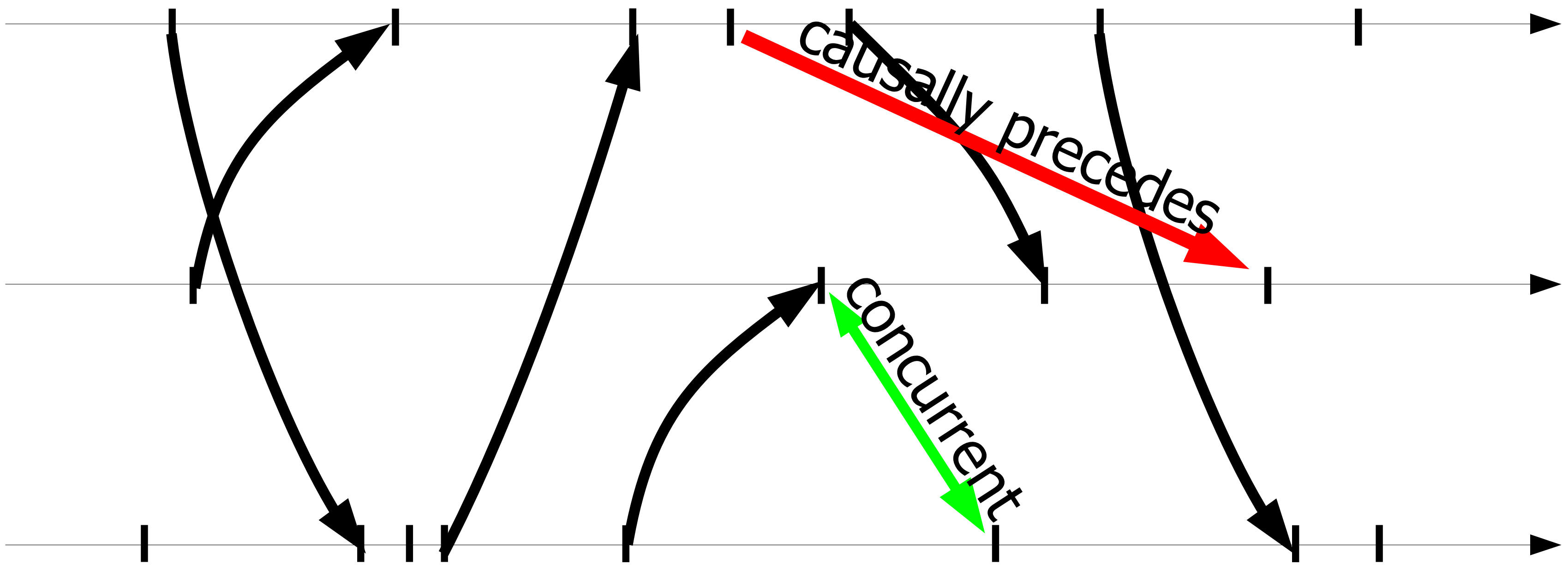
Clock properties

- 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



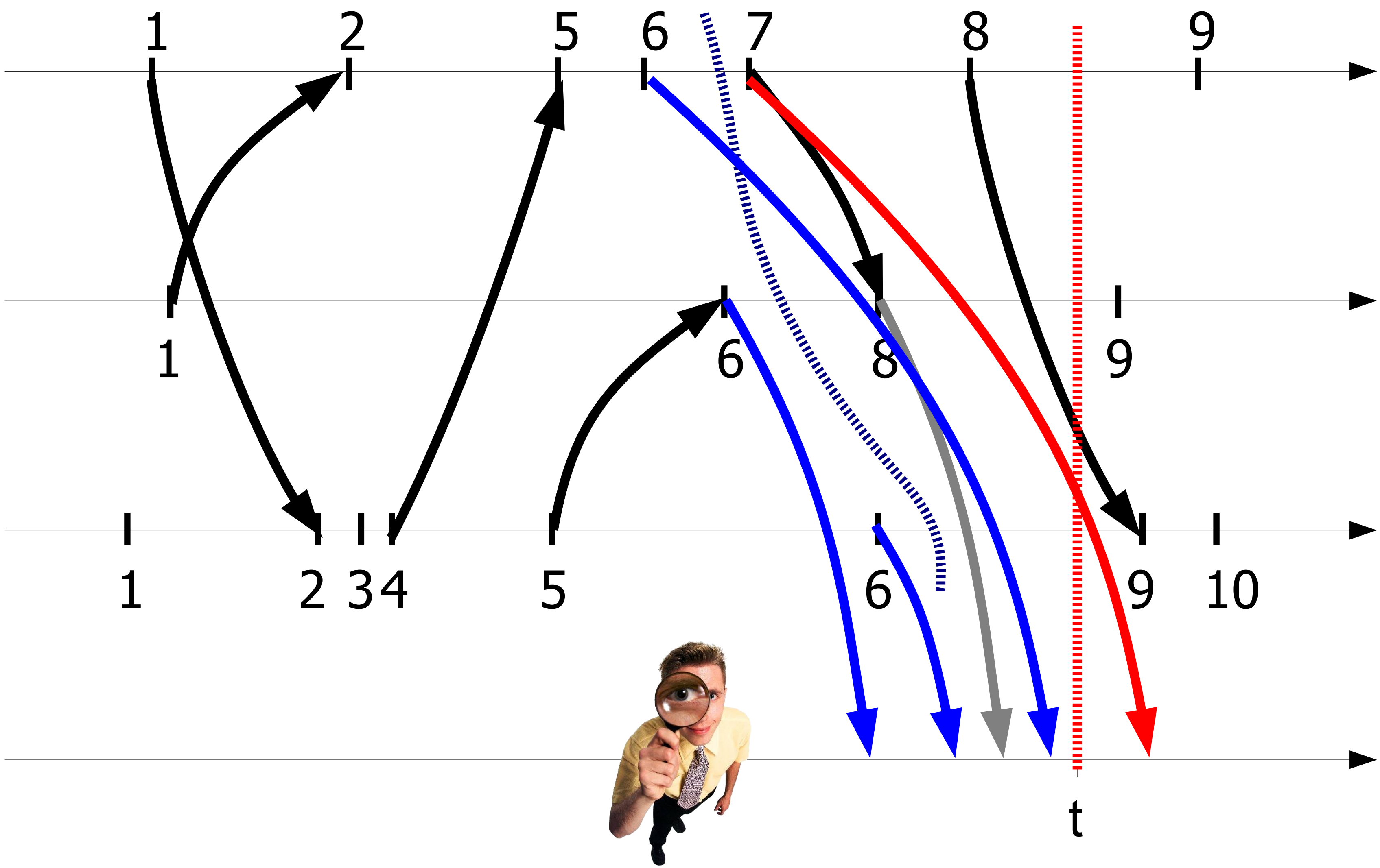
Clock properties

- 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



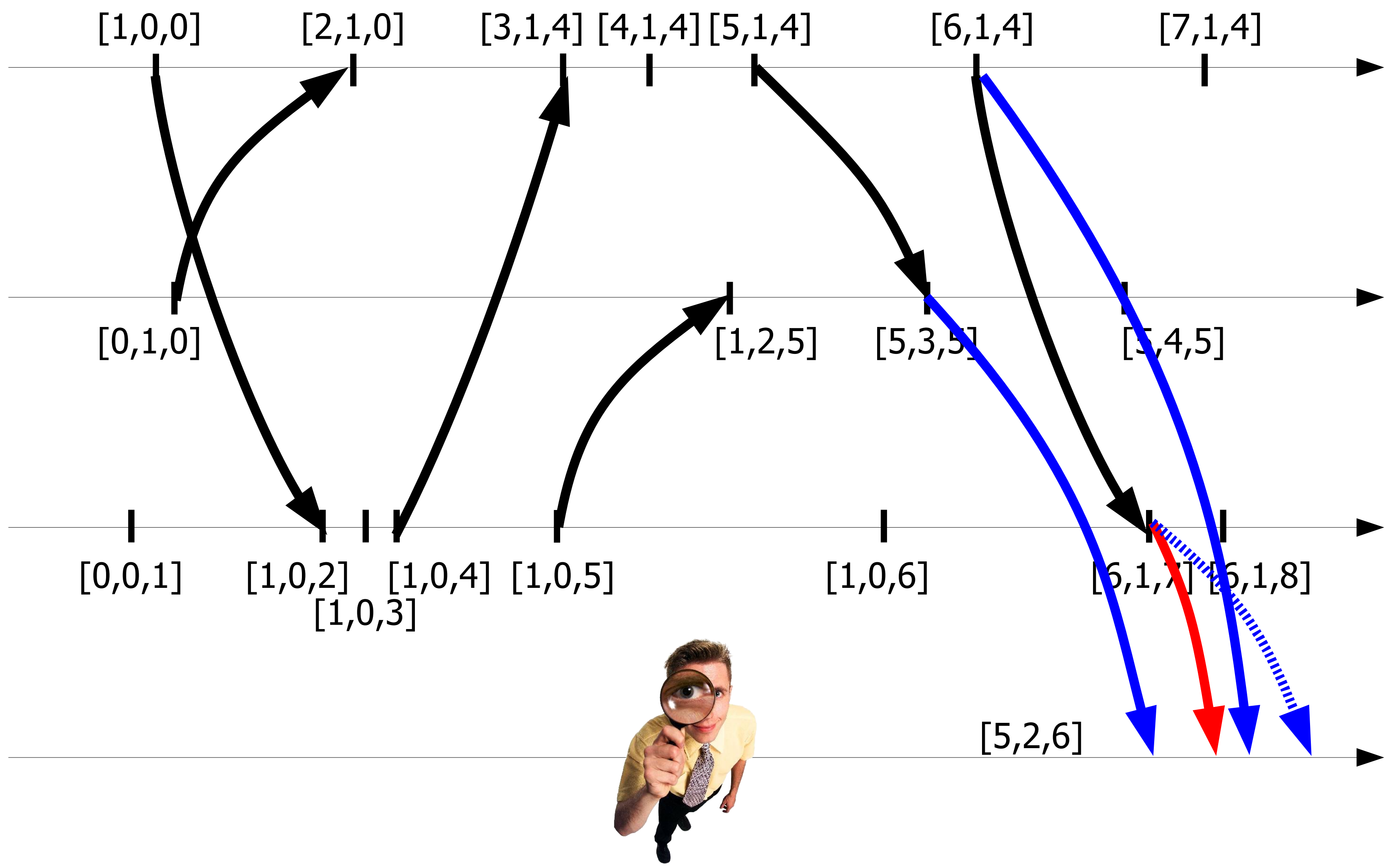
Scalar clocks

- 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

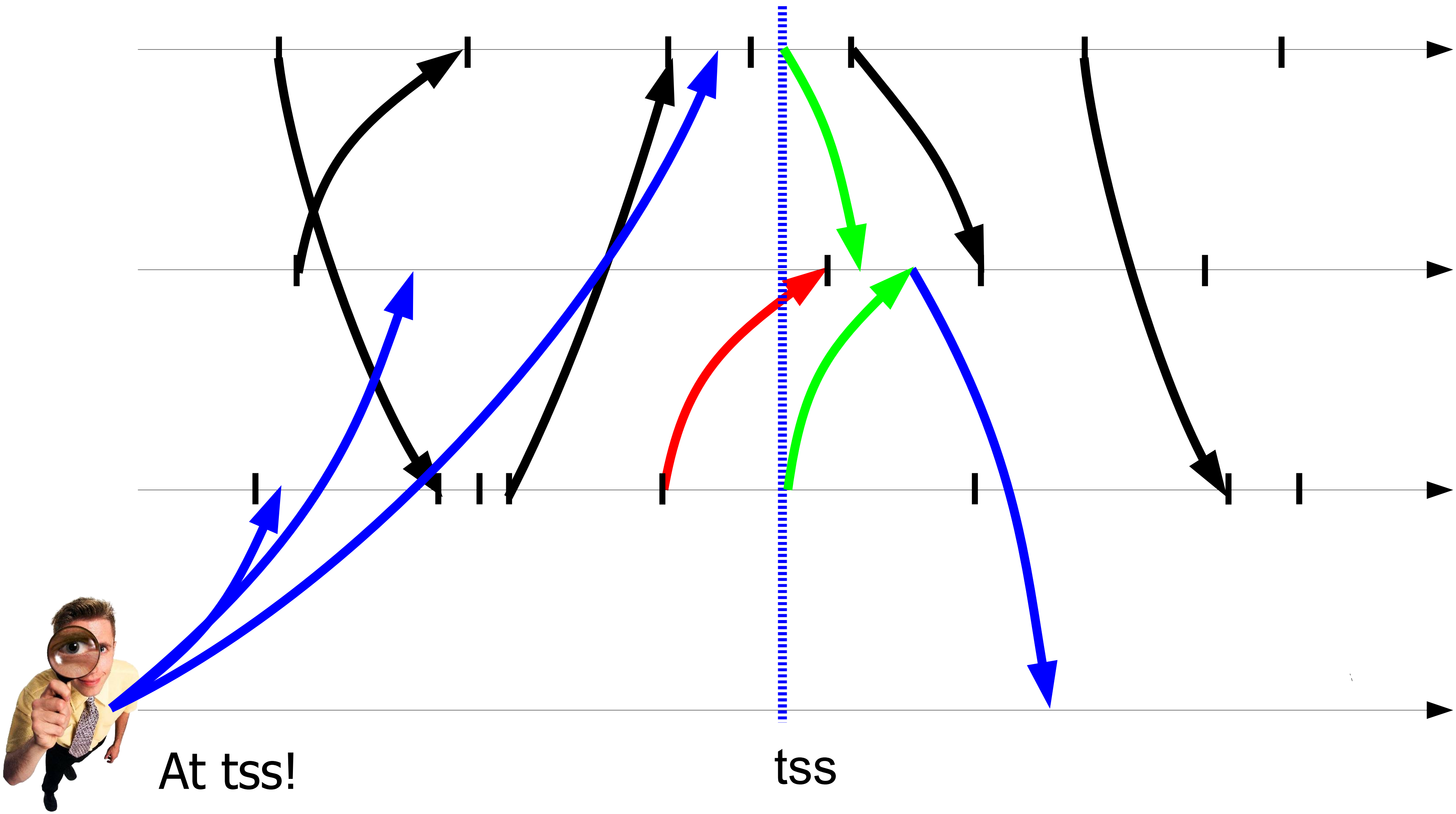
No reporting to monitor process

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

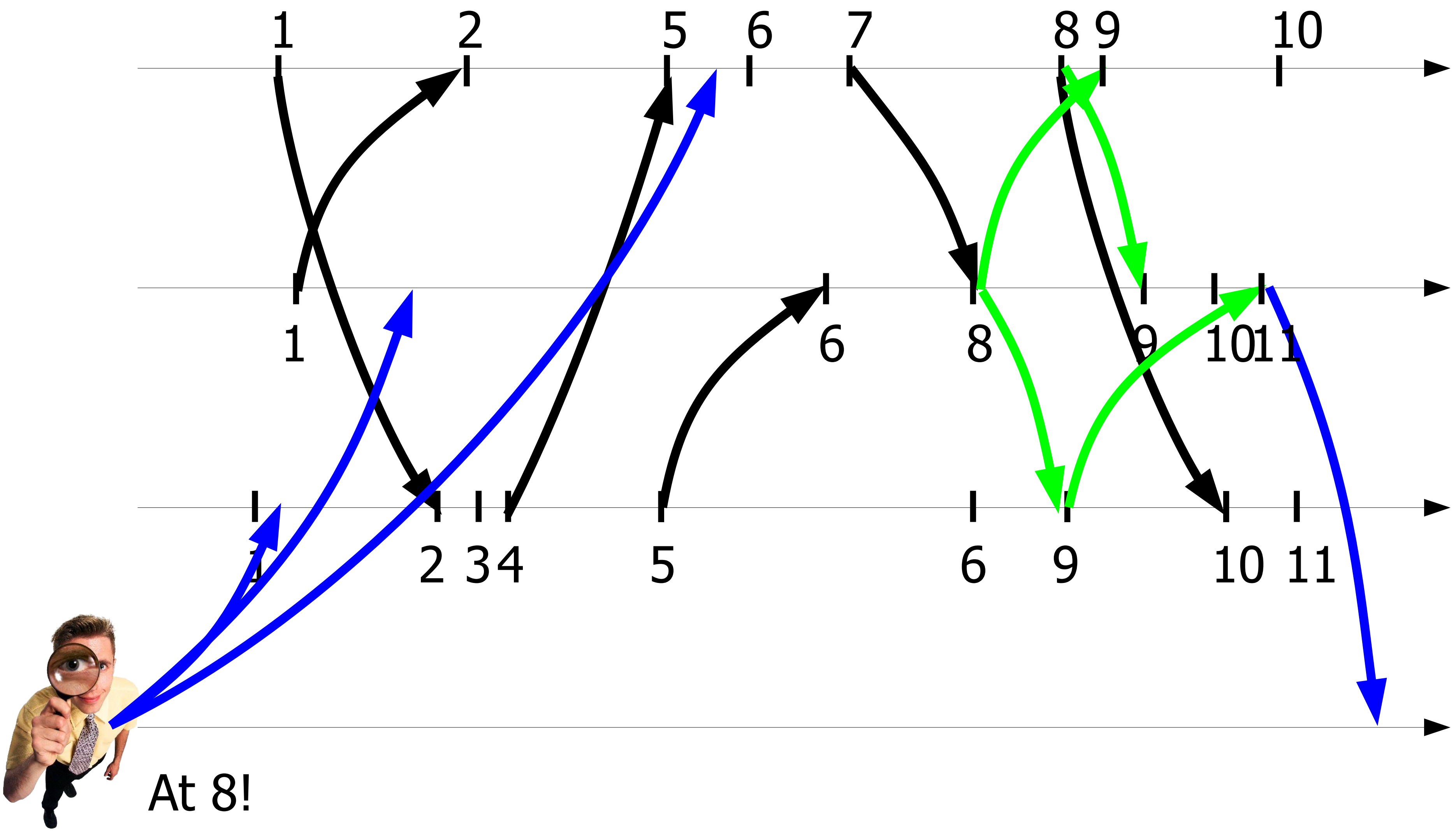
Fourth try: No reporting, synchronous

- 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



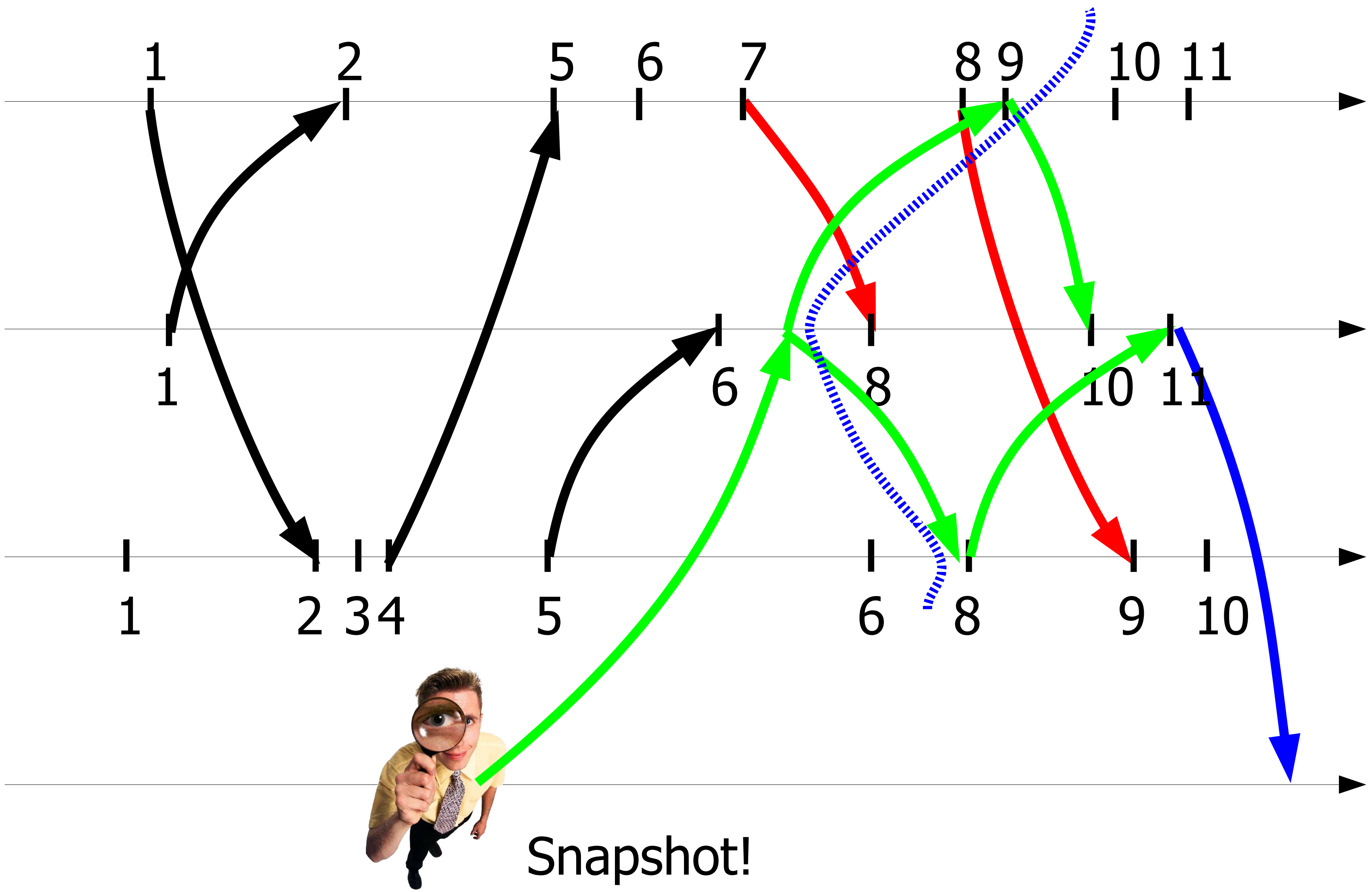
Fifth try: No reporting, logical clock



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



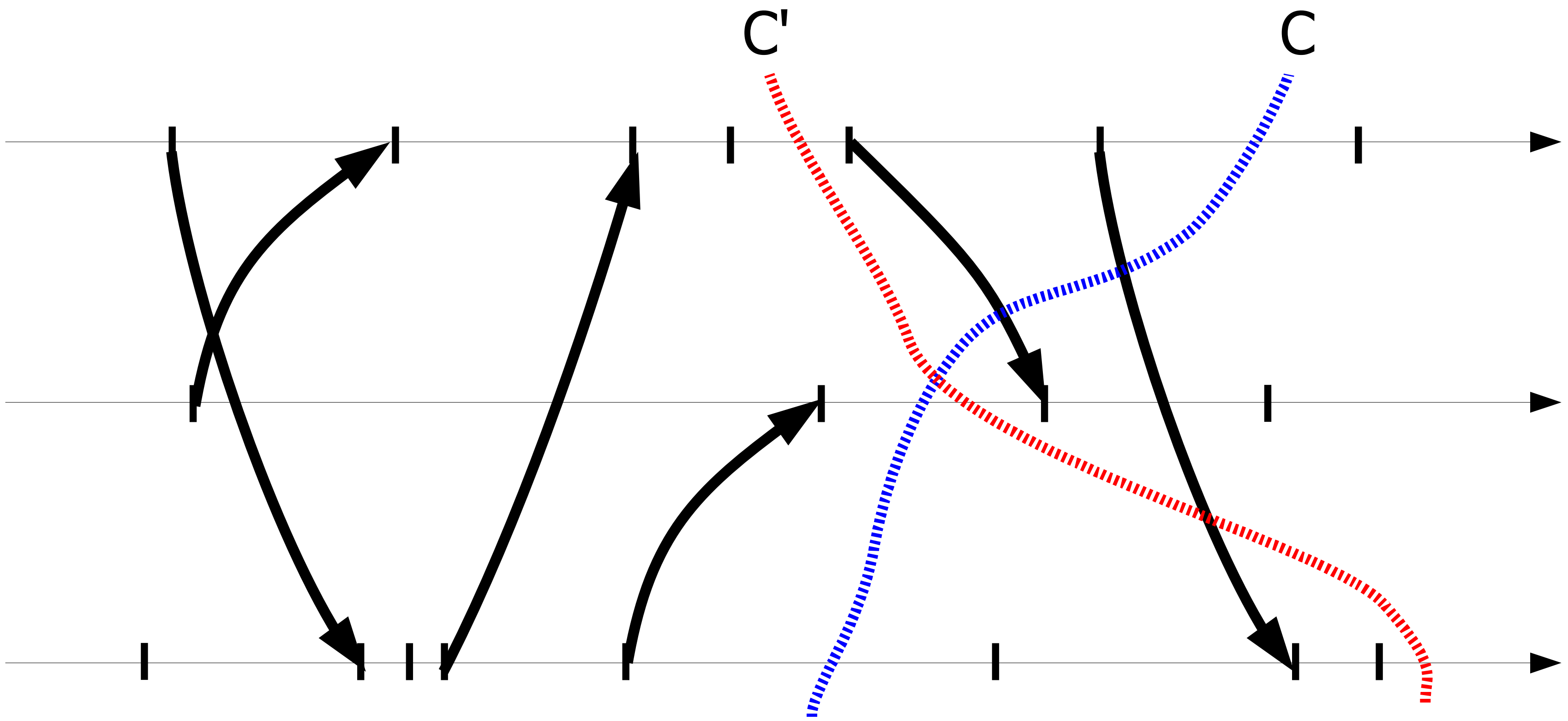
Global Property Evaluation

- What properties can be evaluated?
- Which method for each property?

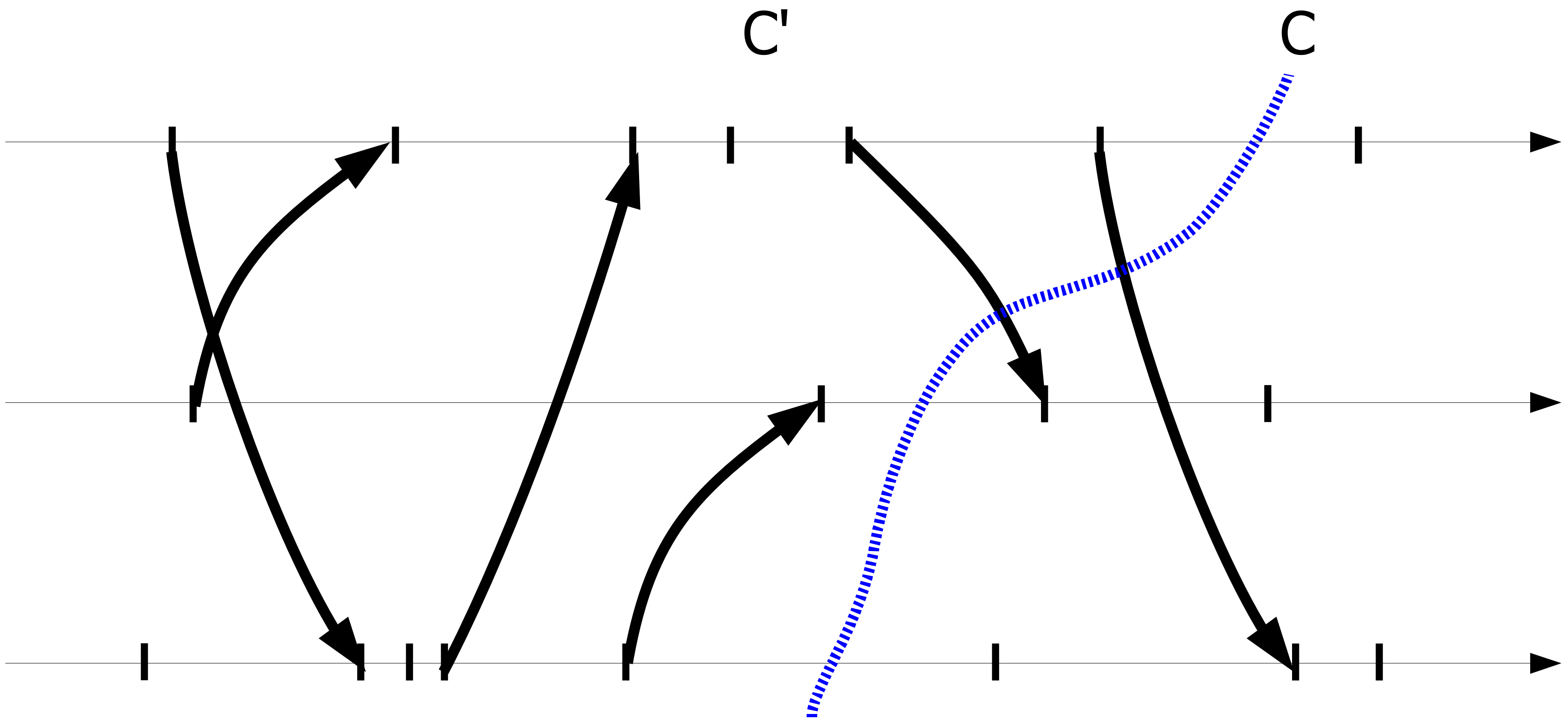
Cuts and consistency

- 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

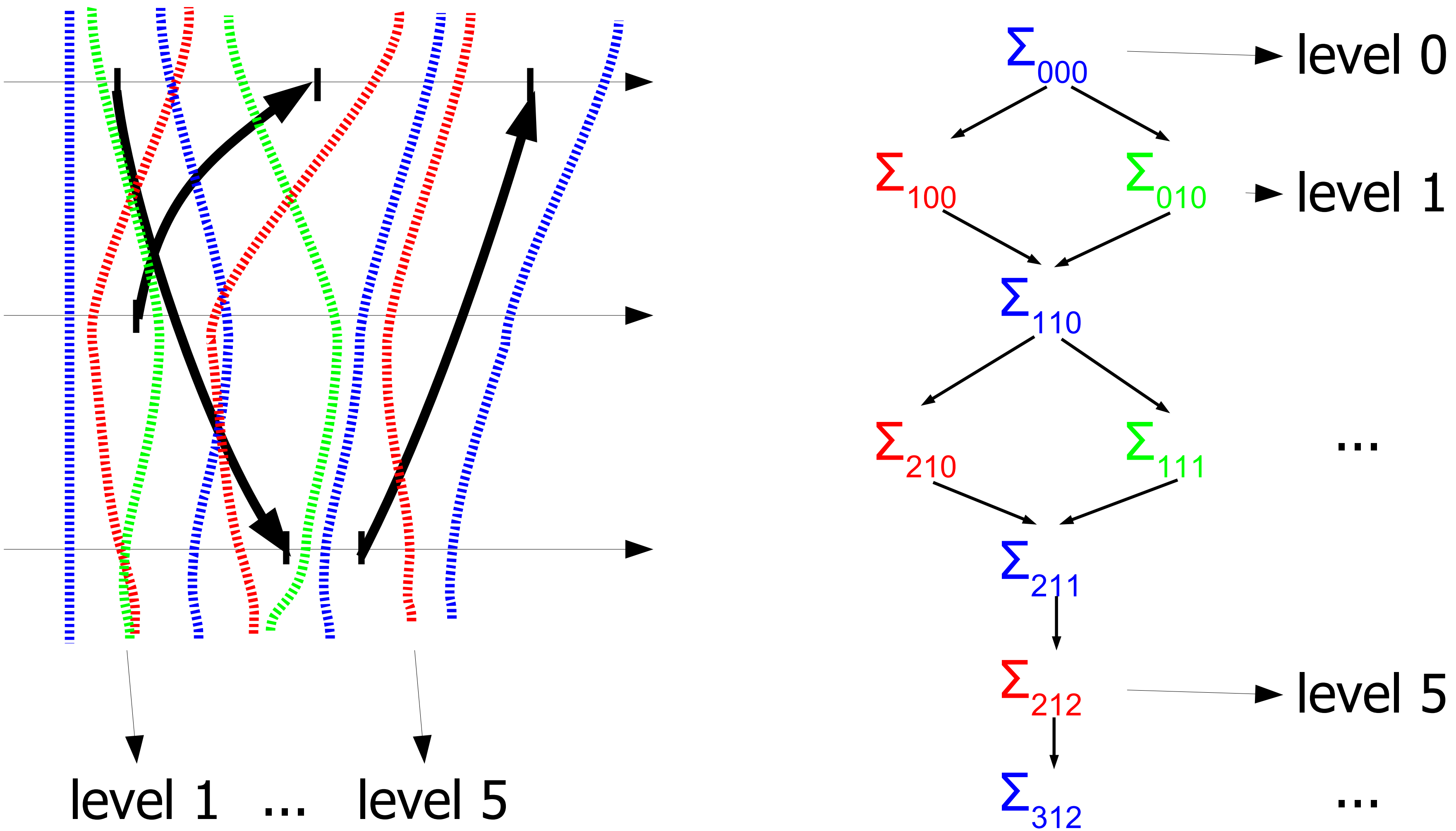


Consistent cuts and global states



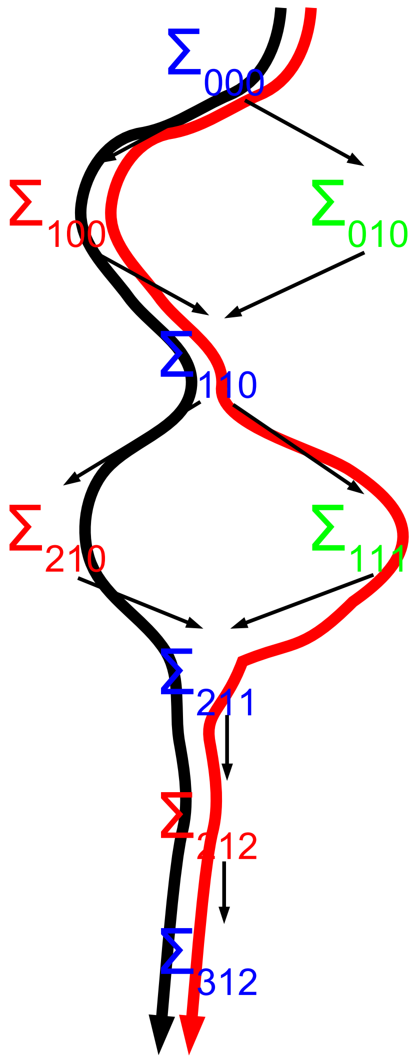
- Notation Σ_{625} means state after 6 events in first process, 2 events in second, ...
- This is a *level 13* state (after 6+2+5 events)

State lattice



State lattice

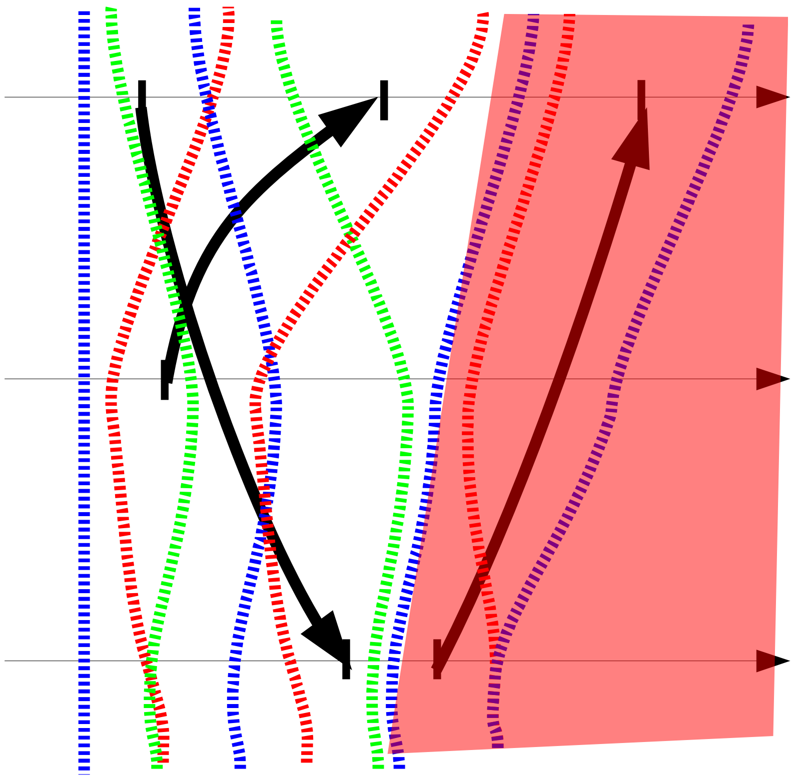
- Includes the true sequence of states in the system
- An observer within the system cannot deny any of the possible paths

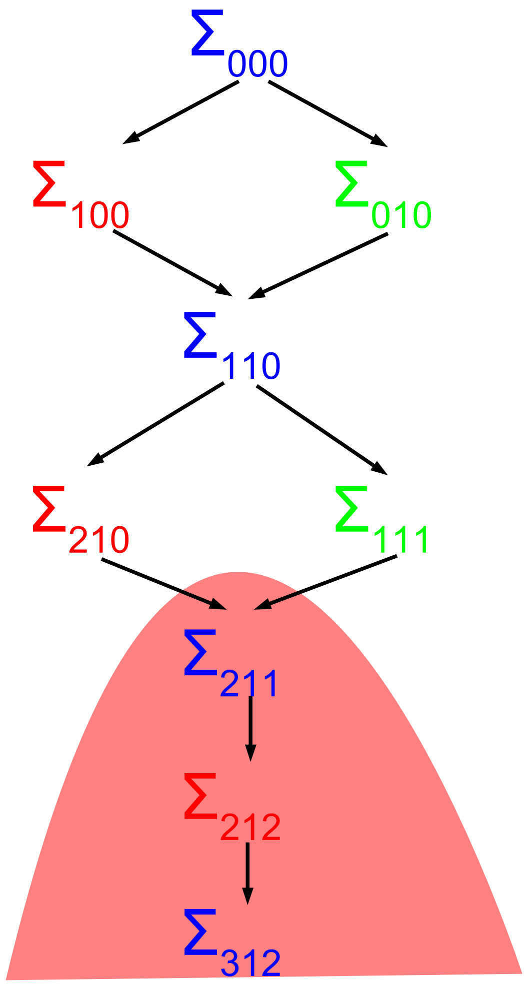


Stable predicates

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

Stable predicates



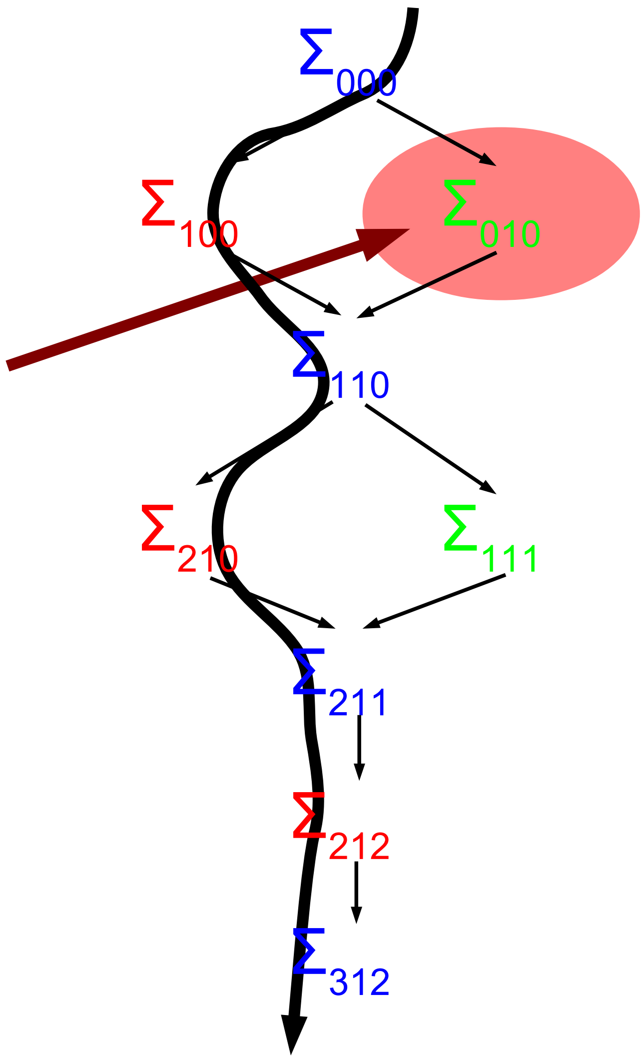


Non-stable predicates

- 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

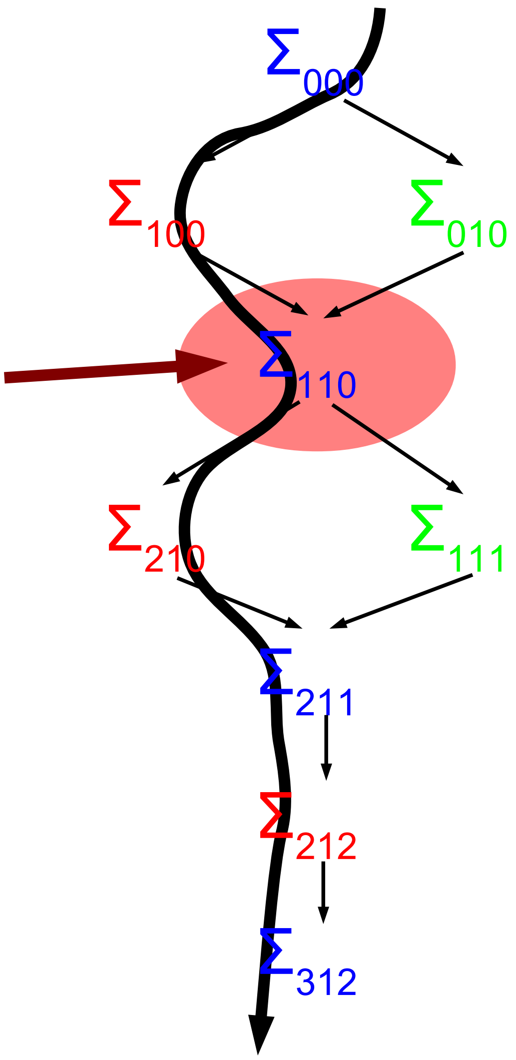
Non-stable predicates

- 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 definitely true: an observer in the system is sure of having been true
- The predicate holds on all possible paths

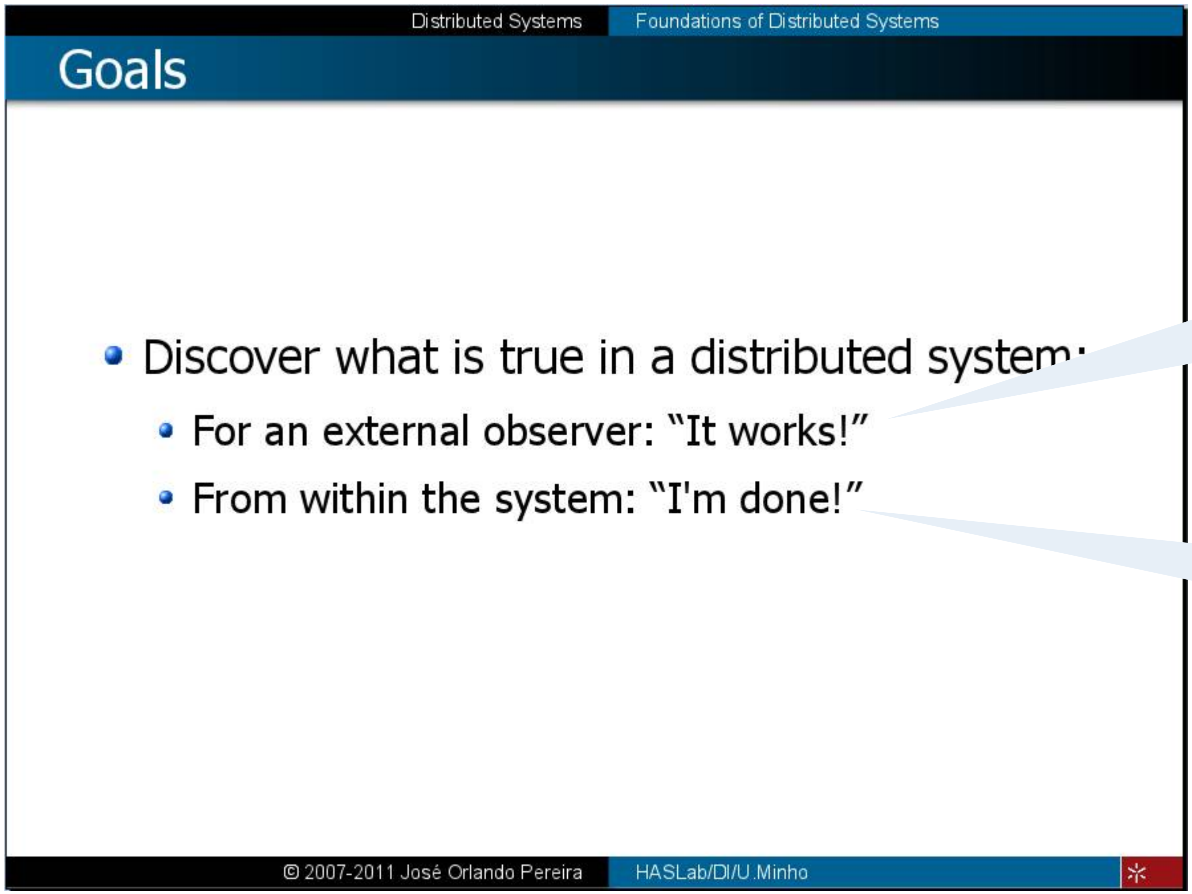


Non-stable predicates

- Start with level n=0
- Loop while more states can be found:
  - Generate all level n states (by selecting all messages that can be accepted in state n-1)
  - If true in <u>all</u> of these states:
    - return **DEFINITLY TRUE**
  - If true in <u>any</u> of these states:
    - return **POSSIBLY TRUE**
  - Increment n
- return **FALSE**

Conclusion

- Both goals achieved:



State machines
and asynchronous models

Logical time and
consistent cuts