In Proceedings of 2006 IEEE 1588 Conference, October 2-4, 2006, Gaithersburg, MD

PTIDES: Programming Temporally Integrated Distributed Embedded Systems

Yang Zhao, EECS, UC Berkeley
Edward A. Lee, EECS, UC Berkeley
Jie Liu, Microsoft Research

2006 Conference on IEEE-1588
NIST, Gaithersburg, MD, USA
October 2 - 4, 2005.

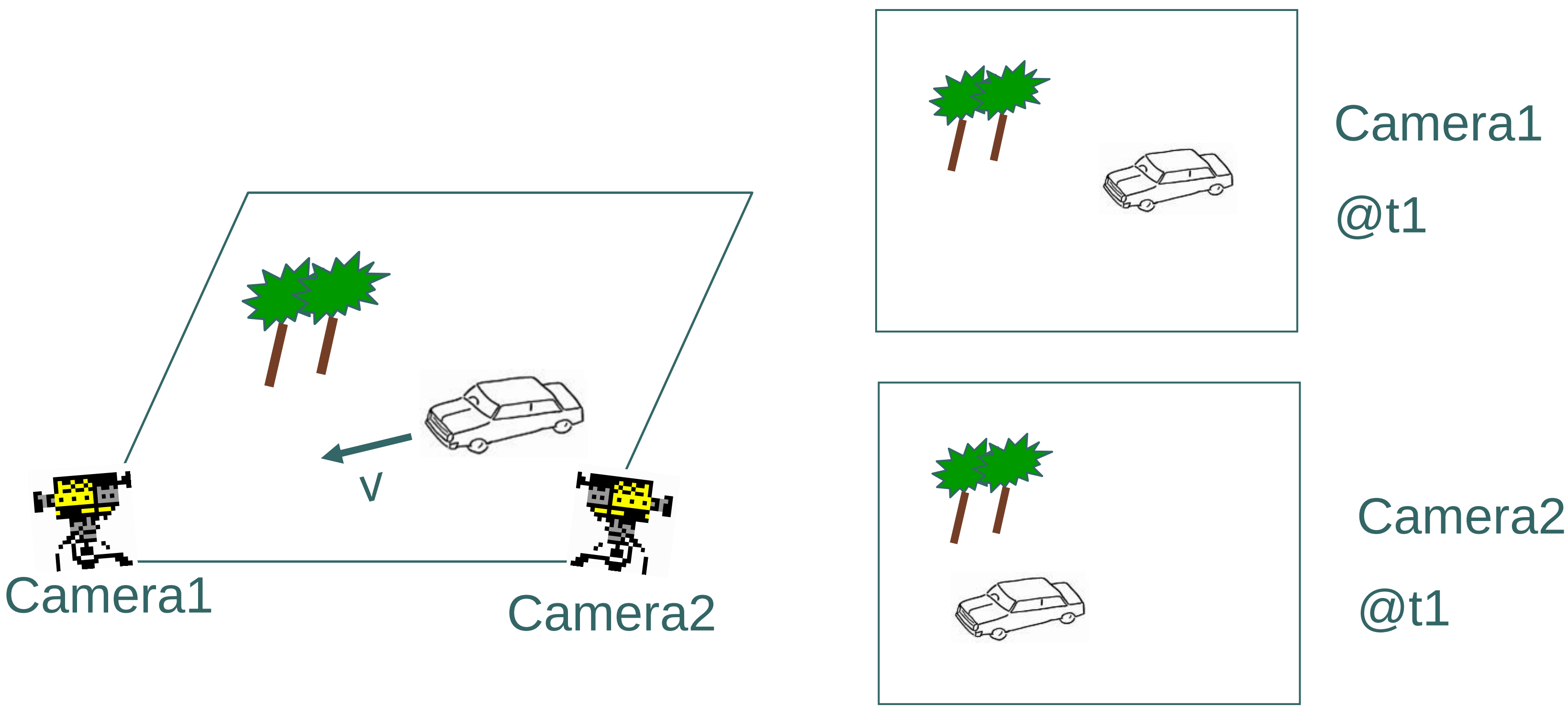
Time in Distributed Systems

- It was true that “A distributed system can be characterized by the fact that the global state is distributed and that a common time base does not exist.” [1]
- Distributed system programming often needs to access information about time.
  - Estimate the time at which events occur
  - Detect process failures
  - Synchronize activities of different systems
- In the past, time synchronization has been a relatively expensive service.
  - Use imprecise clocks.
  - Use logical time/virtual time.

[1] Friedemann Mattern, “Virtual Time and Global States of Distributed Systems”, 1988

Time in Distributed Systems

- A large part of difficulty in programming distributed embedded systems is due to imprecise clocks.

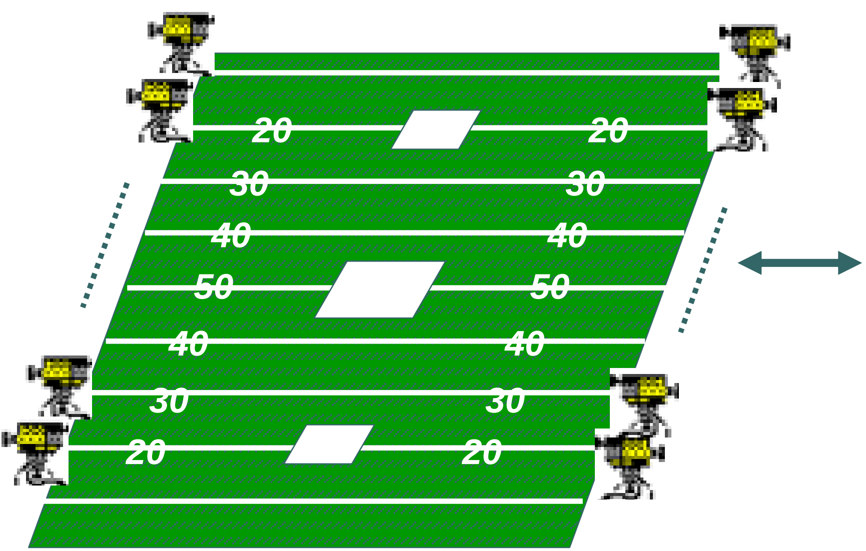


Time in Distributed Systems

- Logical time or virtual time is about ordering of events:
  - *e* < *e'* (event *e* happens before e') if *t(e)* < *t(e')* .
- Now, time synchronization offers a consistent global notion of time.
  - It is meaningful to talk about the metric nature of time: *t(e') – t(e)*
- Time synchronization could greatly change the design of distributed systems!

Motivating Example

- Camera has computer-controlled zoom and focus capabilities.
- Zoom and focus take time to set up, and the camera should not take picture during this period.
- The video of each camera is synchronized and time stamped
  - All the views of some interesting moment can be played back in sequence
  - How often a camera takes picture is also controlled by the computer.



e: zoom camera at t

e': take picture at t'

If $t - t' < \Delta$, then e should be dropped.

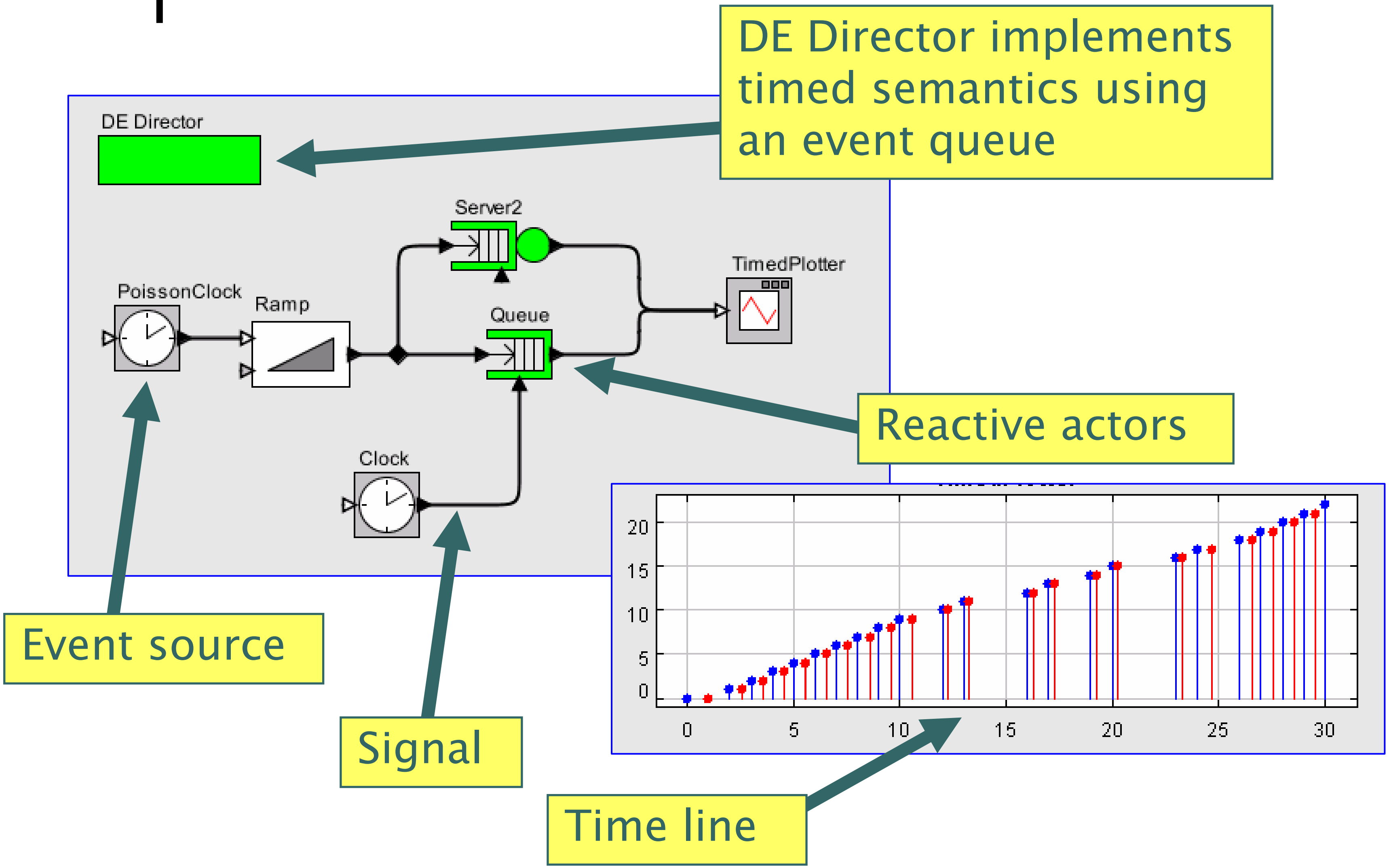
How to Design the Application?

- Challenges include: computation and timing relation between events to be realized in software
- Prevailing software methods abstract away time, replacing it with ordering. Moreover,
  - Order not specified as part of the interface definition.
  - It can be difficult to control the order in concurrent systems.
- Need programming languages that include time and concurrency as first-class properties.
  - Elevating time to the programming language level
    - Time is part of the semantics of programs
  - Augmenting software component interfaces with timing information

Discrete Event Systems

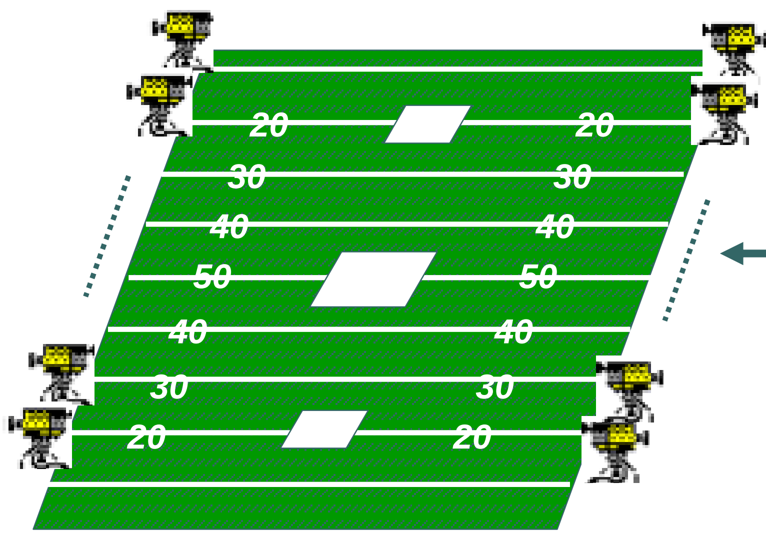
- Dynamic systems that evolve in accordance to events
  - The state of the system changes only when an event occurs
  - Events are associated with time
    - Ex. arrival of a packet, completion of a job, failure of a machine
- DE models have been used for modeling physical systems including:
  - Hardware systems (VHDL, Verilog)
  - Manufacturing systems
  - Communication networks (OPNET, NS-2)
  - Transportation systems
  - Stock market

Discrete Event Modeling in Ptolemy II



Motivating Example

- Camera has computer-controlled zoom and focus capabilities.
- Zoom and focus take time to set up, and the camera should not take picture during this period.
- The video of each camera is synchronized and time stamped
  - All the views of some interesting moment can be played back in sequence
  - How often a camera takes picture is also controlled by the computer.

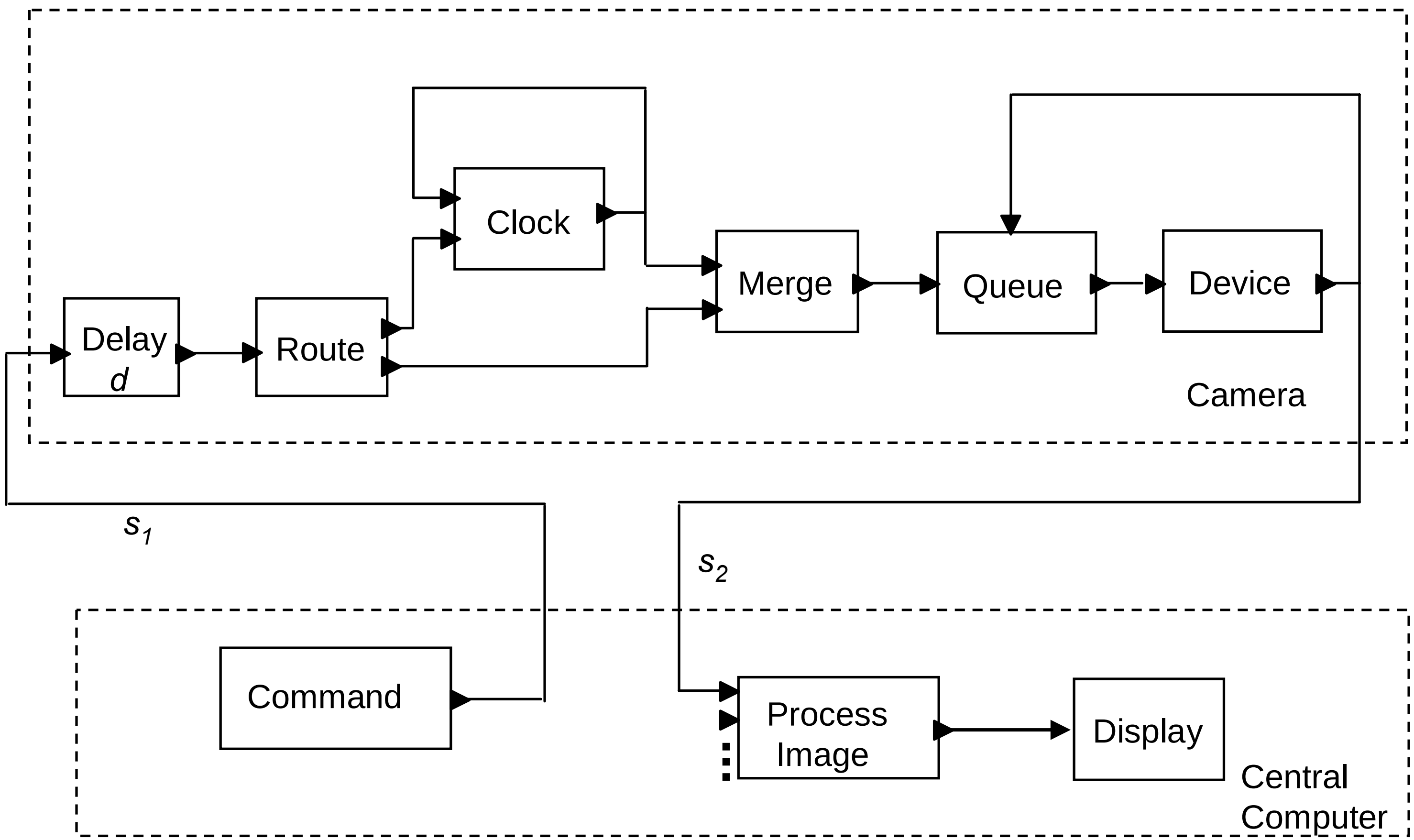


e: zoom camera at t

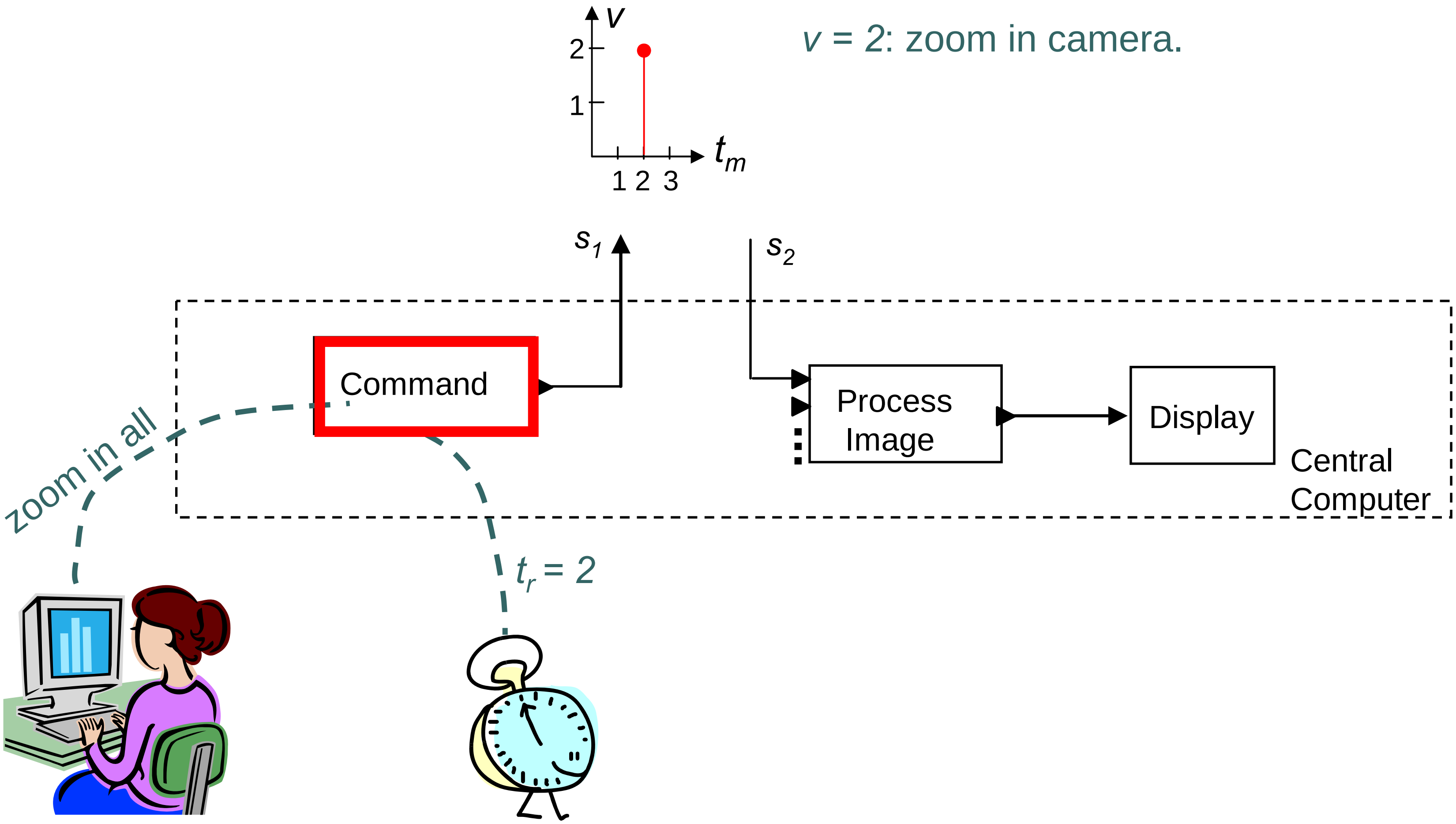
e': take picture at t'

If $t - t' < \Delta$, then e should be dropped.

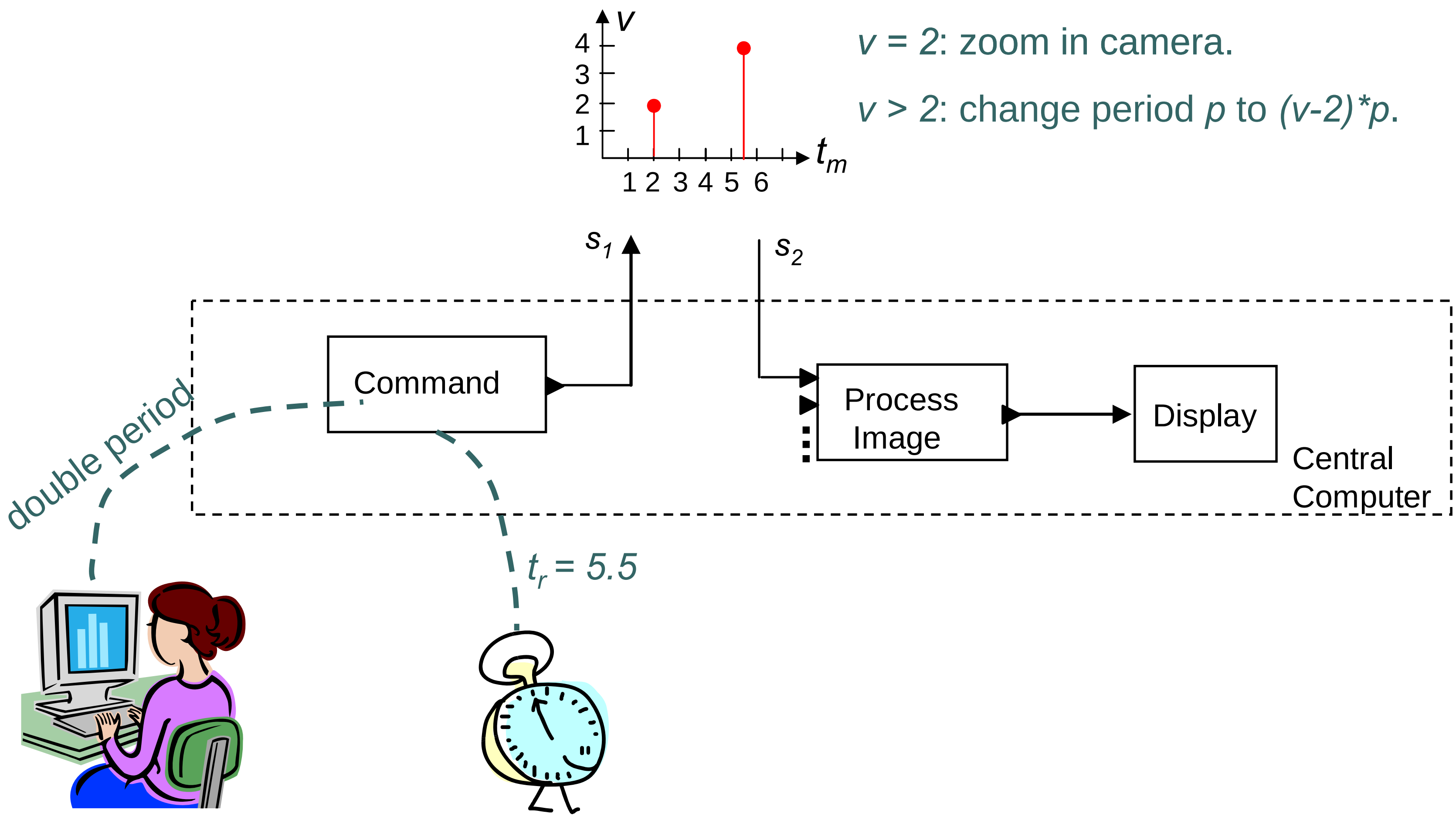
DE Model for the Example



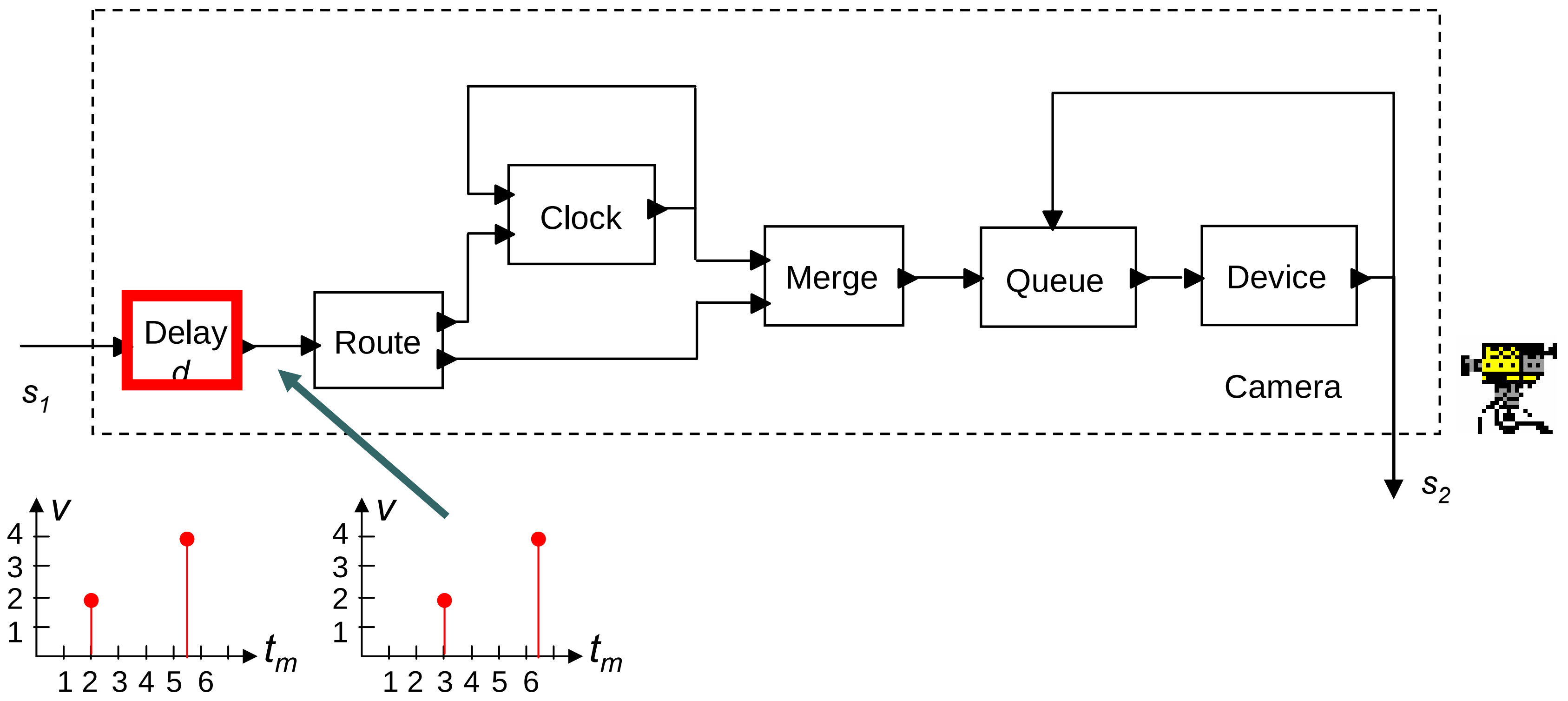
DE Model on the Central Computer



DE Model on the Central Computer

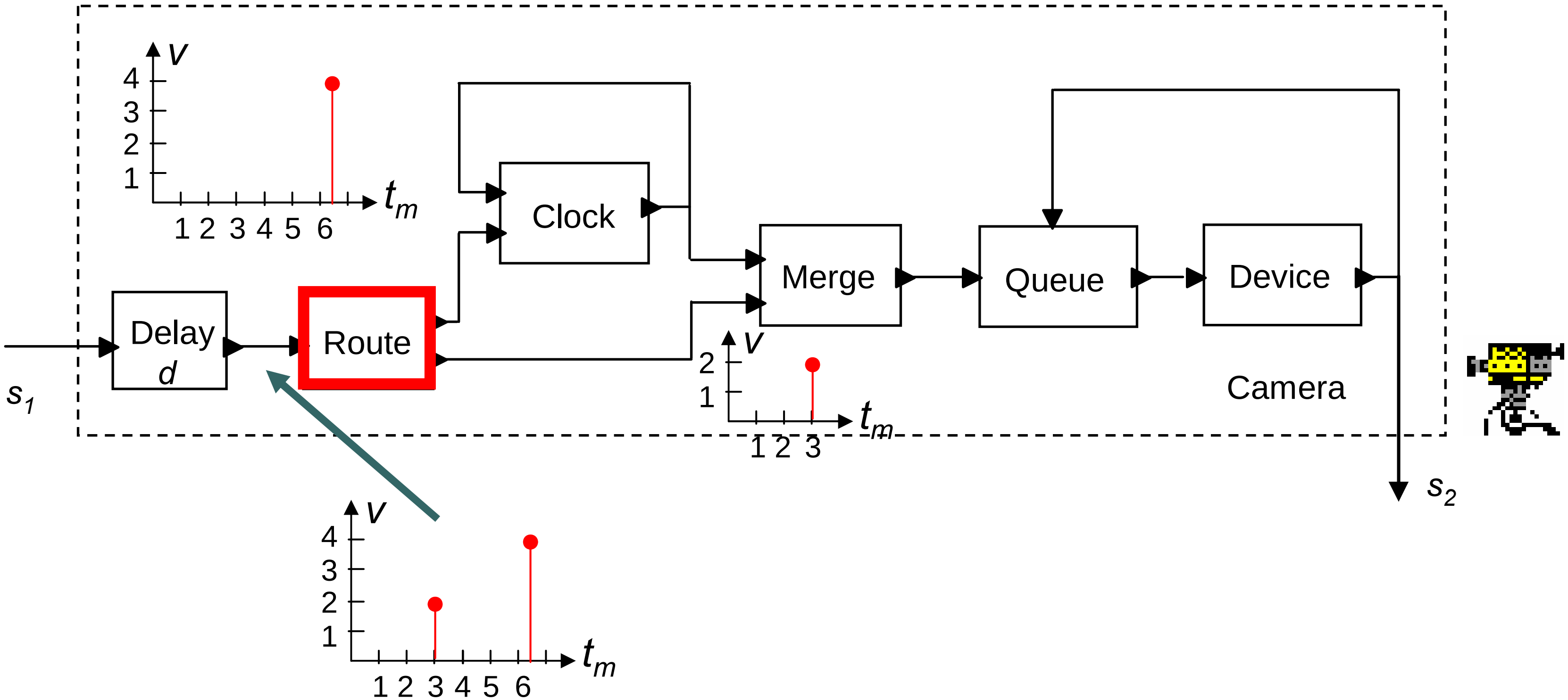


DE Model for the Cameras



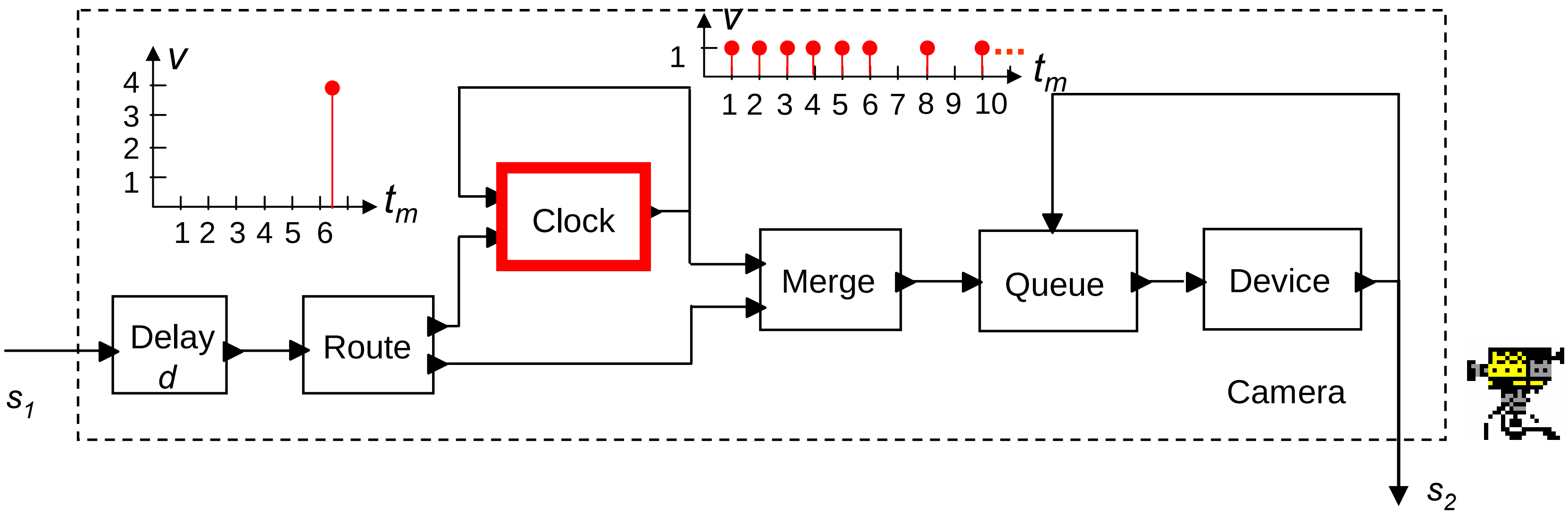
Assume $d = 1$

DE Model for the Cameras

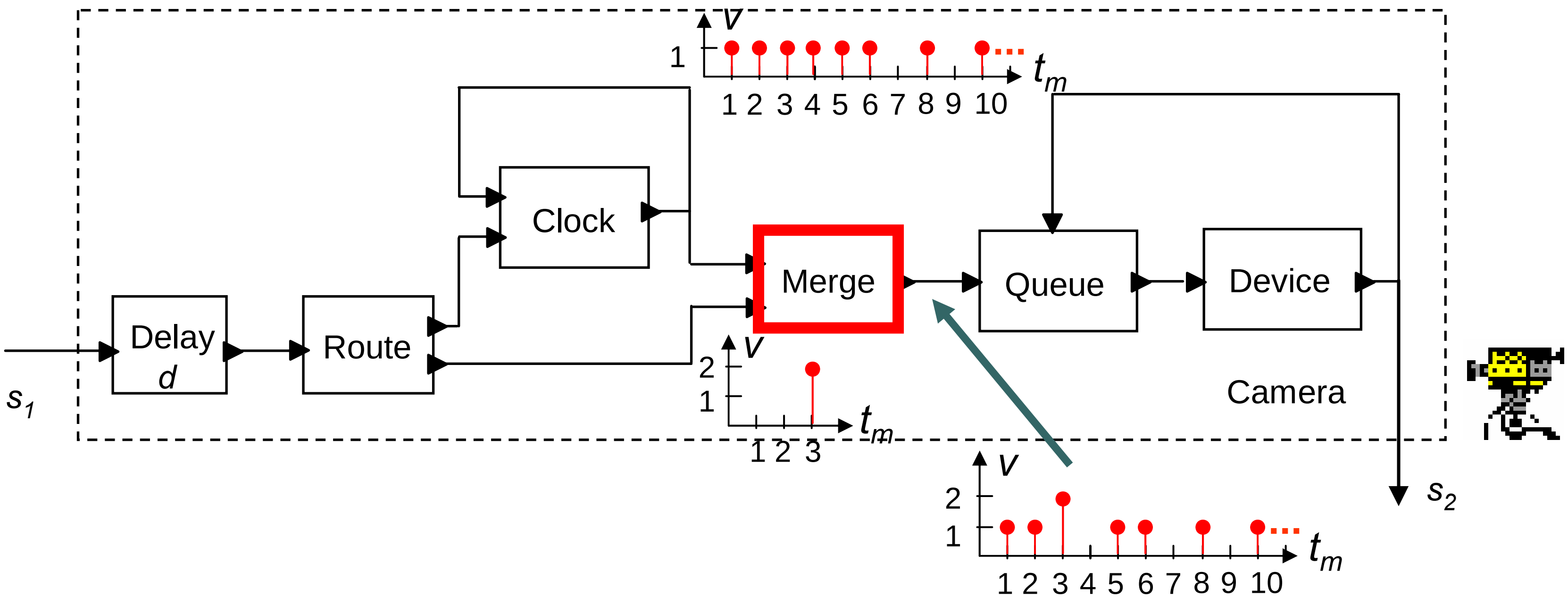


Assume d = 1

DE Model for the Cameras



DE Model for the Cameras

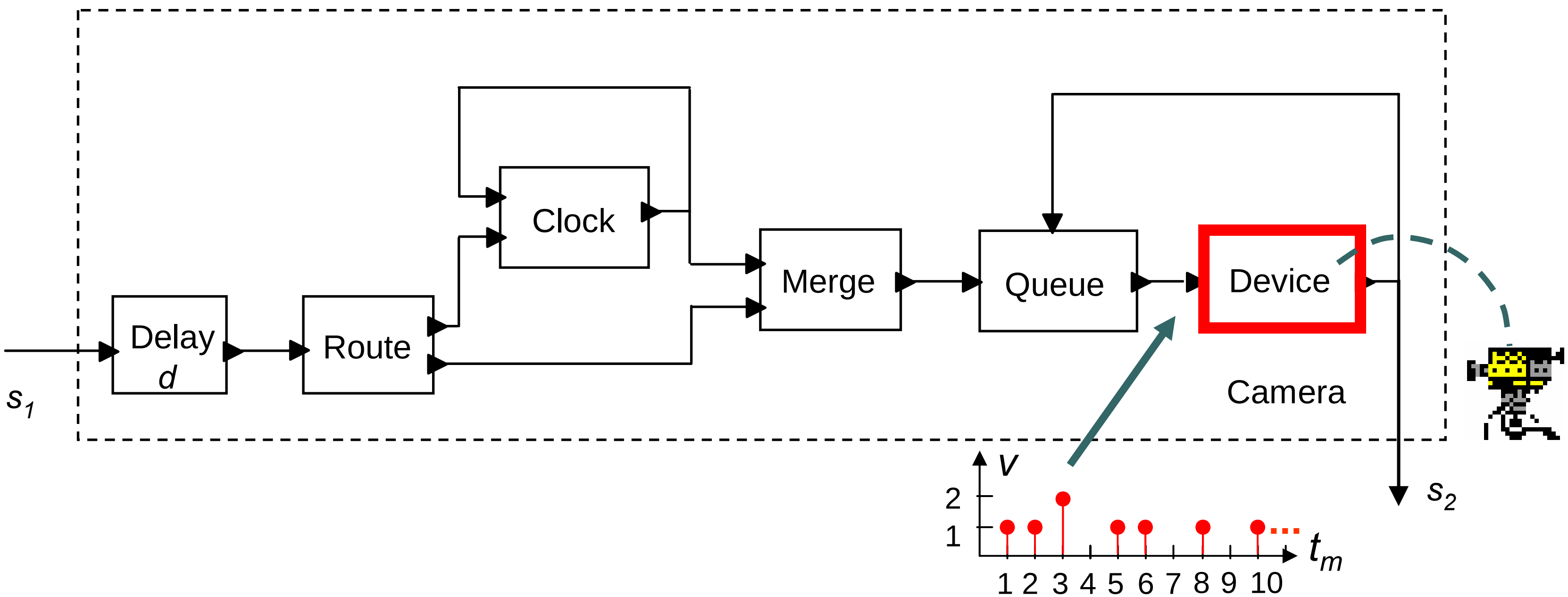


e: zoom camera at t

e': take picture at t'

If t – t' < =1, then e should be dropped.

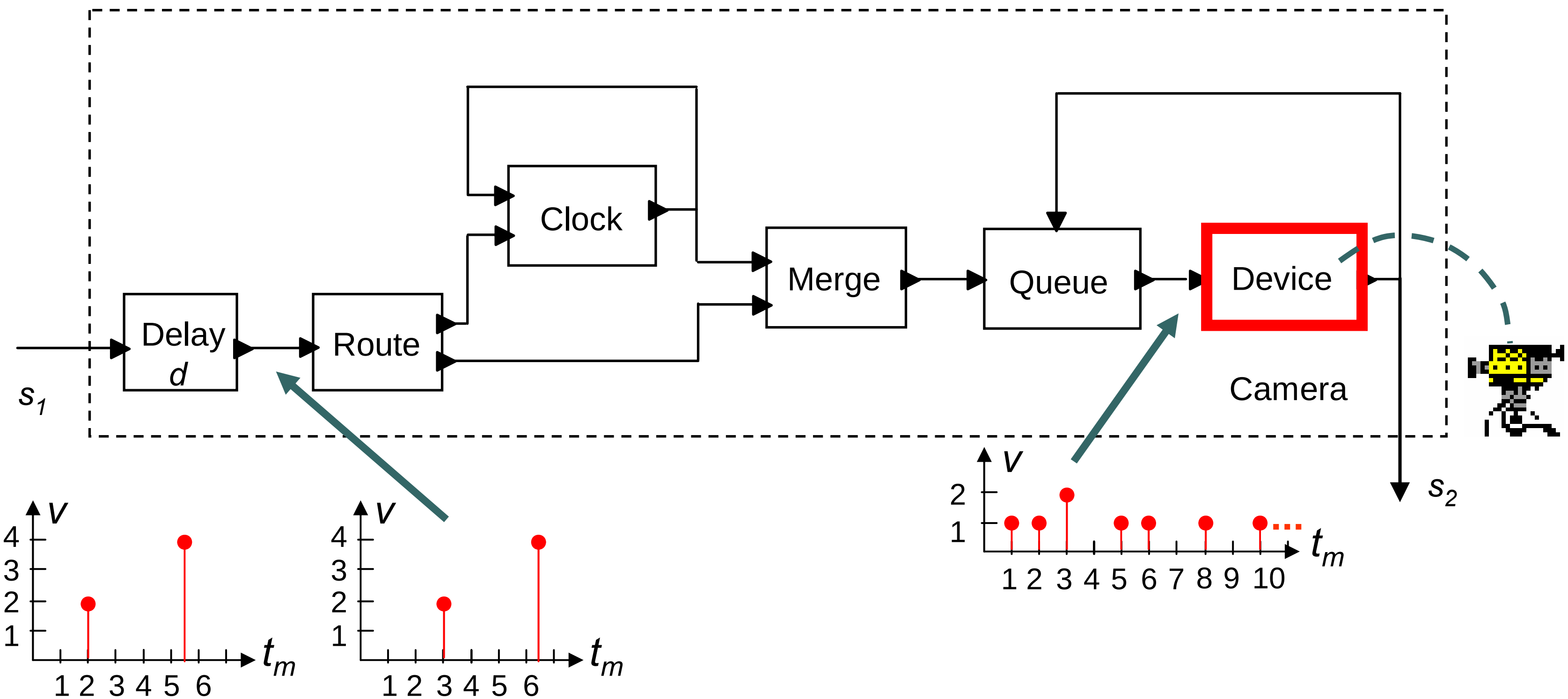
DE Model for the Cameras



Ex. $v = 1$, $t_m = 1$: take a picture at $t_r = 1$.

$v = 2$, $t_m = 3$: zoom in camera at $t_r = 3$.

DE Model for the Cameras



Event at s_1 is received at real time $t_m < t_r <= t_m + D$

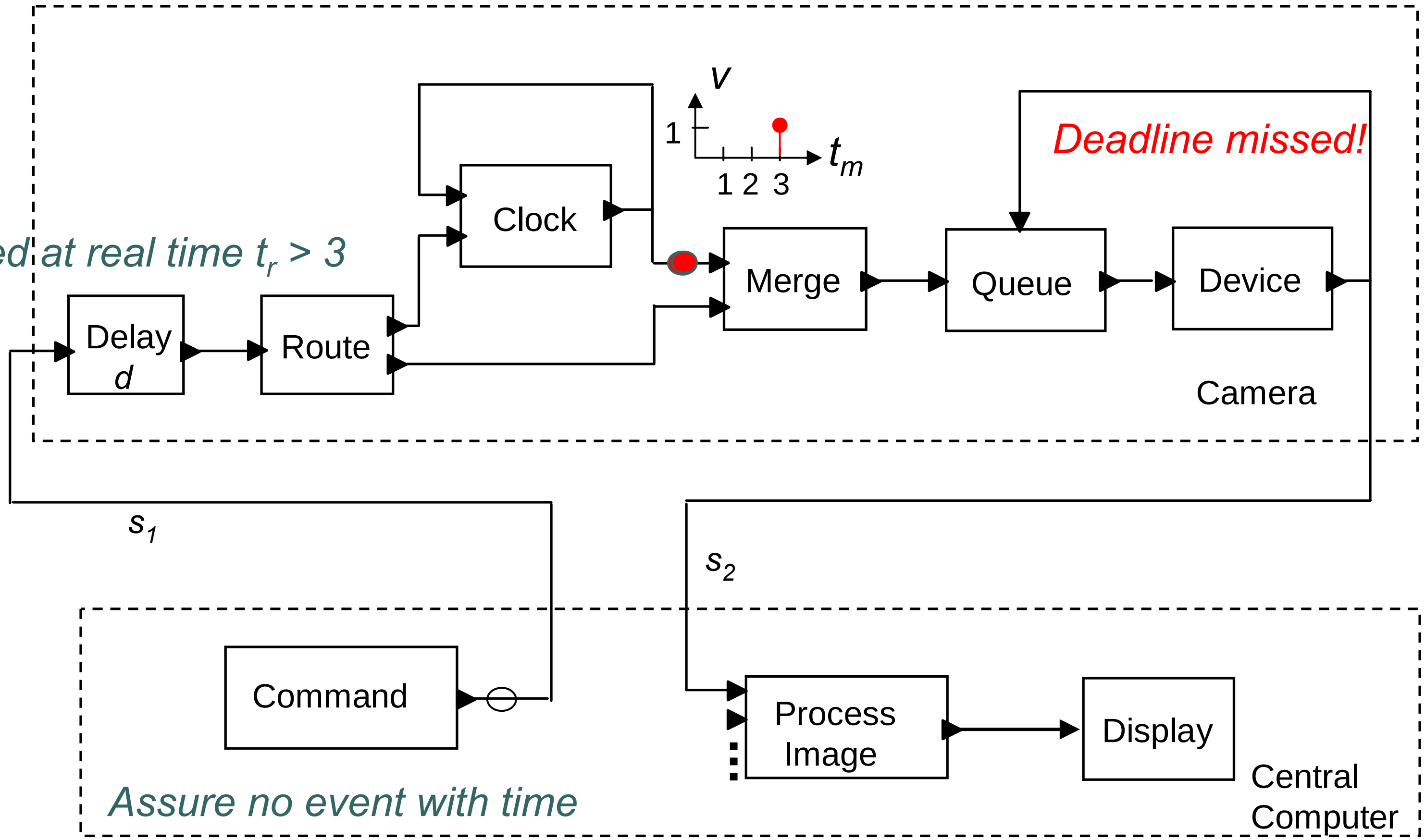
D is the up-bound of network delay

d should greater than D

Challenges in Executing the Model

- Not be practical nor efficient to use a centralized event queue to sort events in chronological order.
- Do the techniques developed for distributed DE simulation work?
  - Conservative? Optimistic?
- Our approach: events only need to be processed in time-stamp order when they are causally related.

DE Model for the Example



Received at real time $t_r > 3$

Assure no event with time stamp $t_m < 3$ at $t_r = 3$

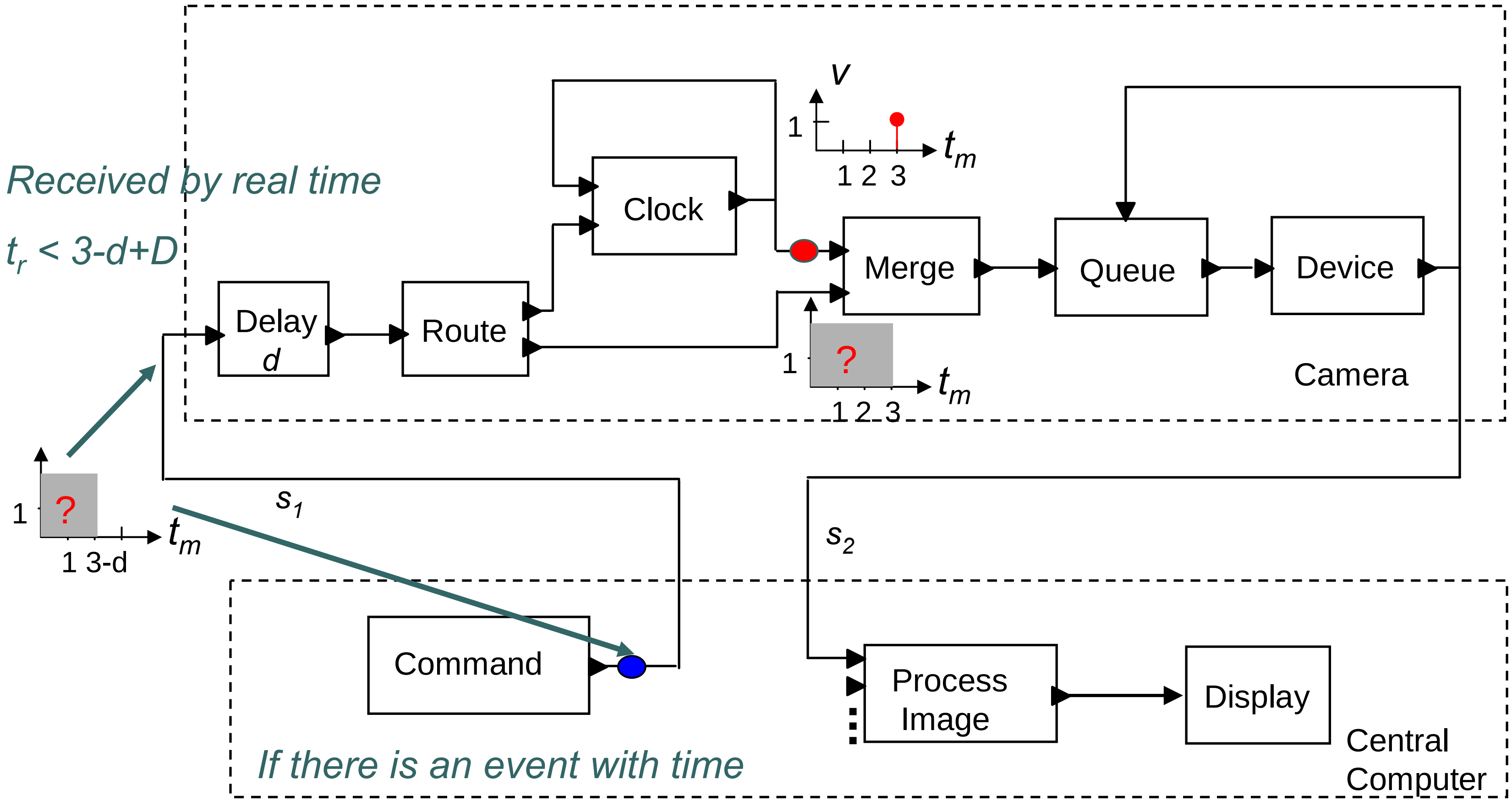
Challenges in Executing the Model

- Not be practical nor efficient to use a centralized event queue to sort events in chronological order.
- Do the techniques developed for distributed DE simulation work?
  - Conservative? Optimistic?
- Our approach: events only need to be processed in time-stamp order when they are causally related.

Intuition on Out of Order Execution

Received by real time

$t_r < 3\text{-}d+D$

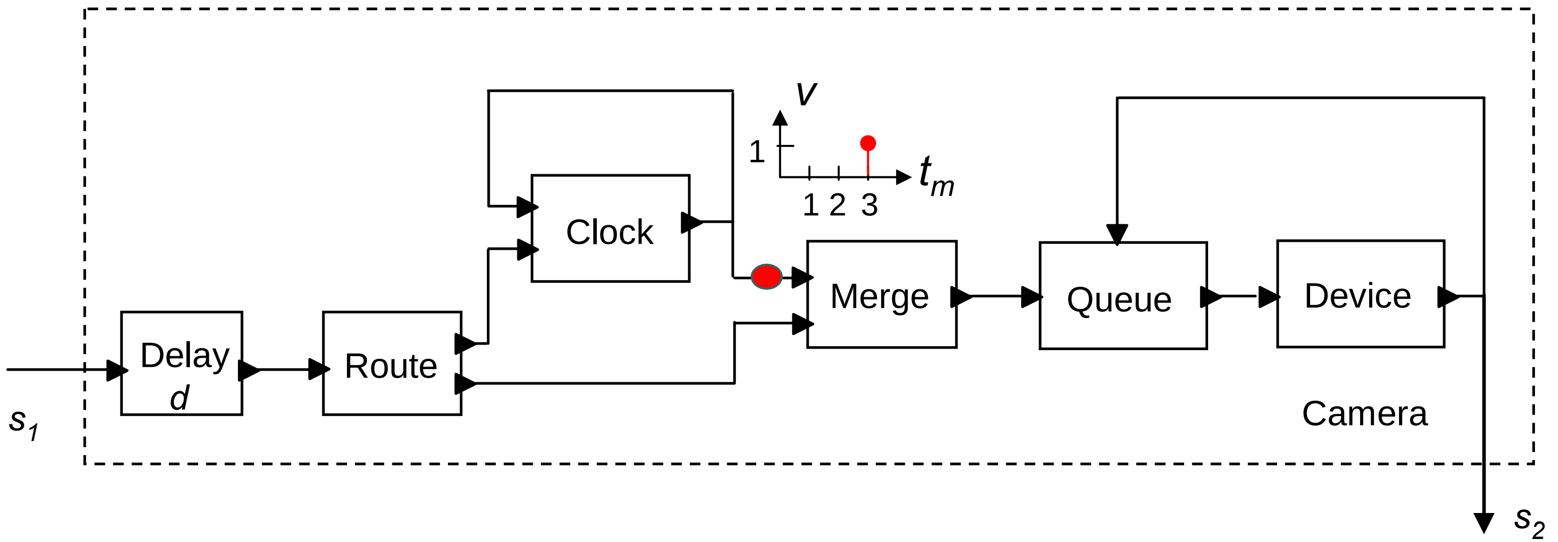


If there is an event with time stamp $t_m \le 3\text{-}d$ at $t_r \le 3\text{-}d$

D : is the up bound of network delay; d > D

Intuition on Out of Order Execution

We can always safely process an event e at the first input of Merge by $t_r > t_m - d + D$



D : is the up bound of network delay; d > D

Relevant Dependency Analysis

- Relevant dependency analysis gives a formal framework for analyzing causality relationships to determine the minimal ordering constraints on processing events.
- It capture the idea that events only need to be processed in time-stamp order when they are causally related.
- Can preserve the deterministic behaviors specified in DE models without paying the penalty of totally ordered executions.

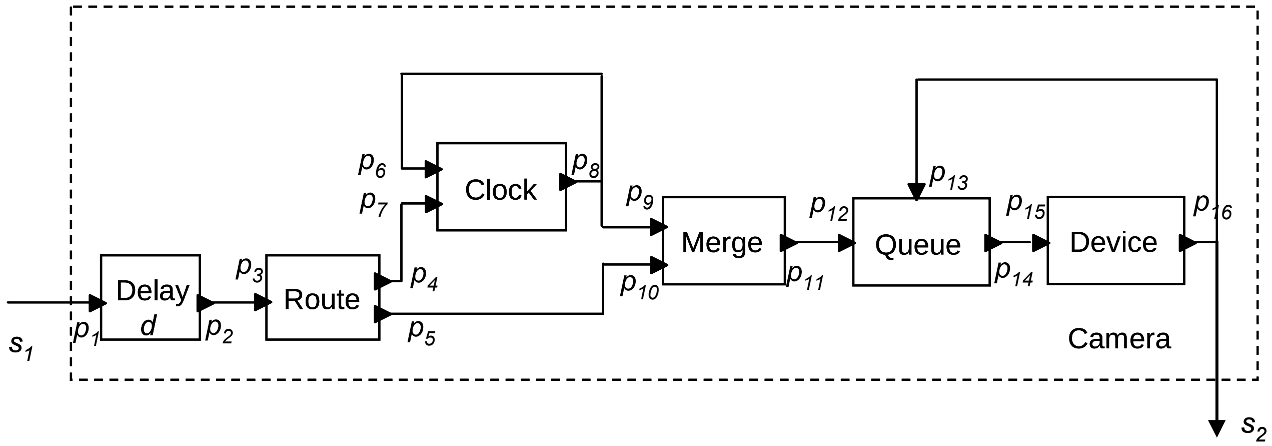
Yang Zhao, Edward A. Lee and Jie Liu "Programming Temporally Integrated Distributed Embedded Systems'', UCB/EECS-2006-82, May 28, 2006

Causality Interface

[Zhou--Lee]

- Causality interface of a component declares the dependency between input and output.

$$\delta_a : P_i \times P_o \rightarrow D$$



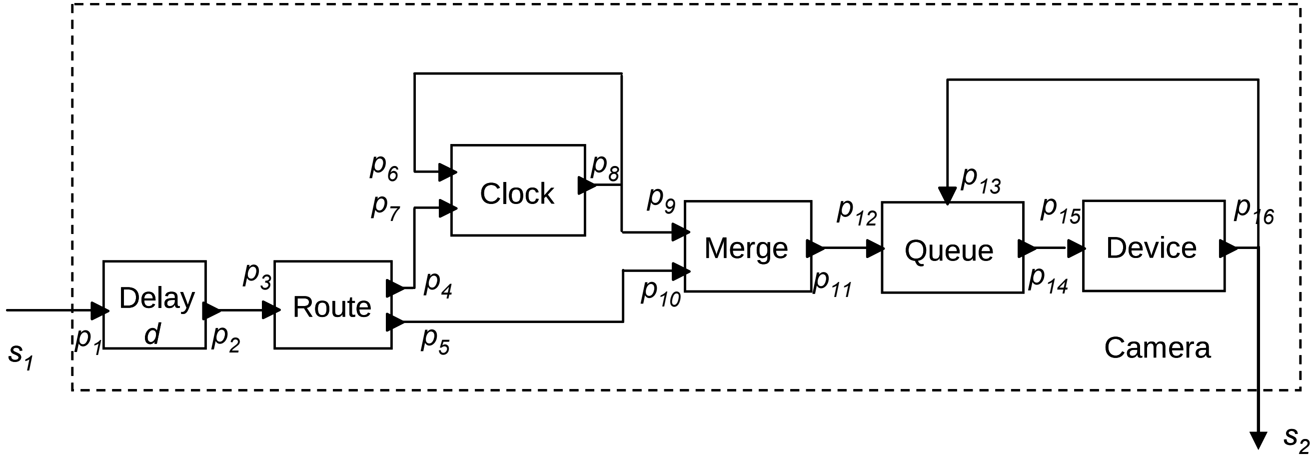
$$\delta_a(p_1, p_2) = d$$

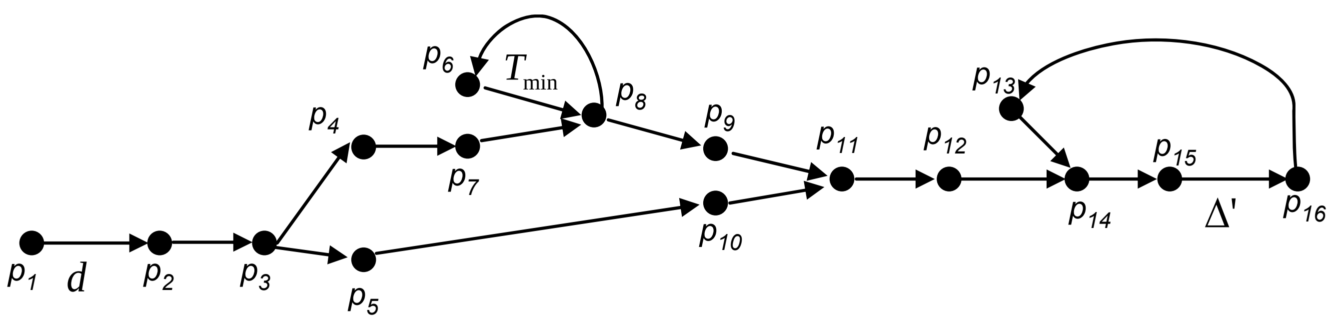
$$\delta_a(p_6, p_8) = T_{\min}$$

$$\delta_a(p_7, p_8) = 0$$

$$\delta_a(p_{15}, p_{16}) = \Delta'$$

Causality Interface Composition

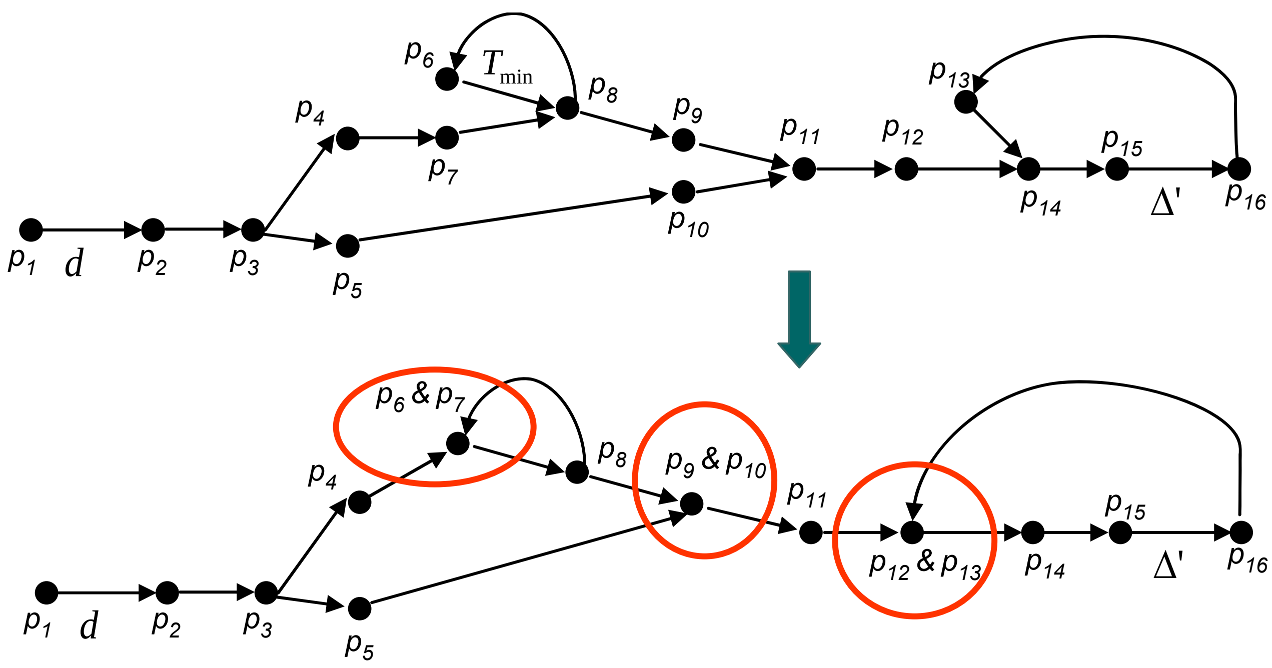




$$\delta(p_1, p_9) = d$$

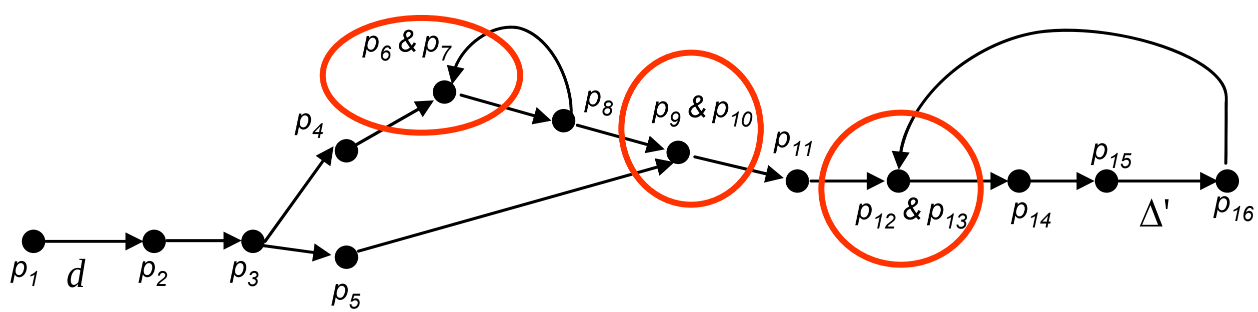
Relevant Dependency

- Relevant dependency on any pair of input ports p1 and p2 specifies whether an event at p1 will affect an output signal that may also depend on an event at p2.



Relevant Dependency

- $d(p_1, p_6) = d$ means any event with time stamp t at p_2 can be processed when all events at p_1 are known up to time stamp $t - d$.



Relevant Order

- Relevant dependencies induce a partial order, called the relevant order, on events.
- $e_1 <_r e_2$ means that e1 must be processed before e2.
- If neither $e_1 <_r e_2$, nor $e_2 <_r e_1$, i.e. $e_1 ||_r e_2$, then e_1, e_2 can be processed in any order.
- This technique can be adapted to distributed execution.

Conclusion

- Time synchronization can greatly change the way distributed systems are designed.
- Discrete-event model can be used as a programming model to explicitly specify and manipulate time relations between events.
- It is challenging to design distributed systems to make sure they are executable.
- Causality analysis can be used to determine when events can be processed out of order to improve executability.
- Work in progress:
  - statically check whether a system design is executable.
  - Implementing a runtime environment on P1000 by Agilent.