Today

- Reactively detecting properties.
- Generalizing FIFO: Causal delivery.
- Strong clock condition and vector clocks.
  - Some eye-blurring equations.
- Implementing causal delivery.
- Revisiting RPC deadlock.
  - A realistic deadlock detection protocol.

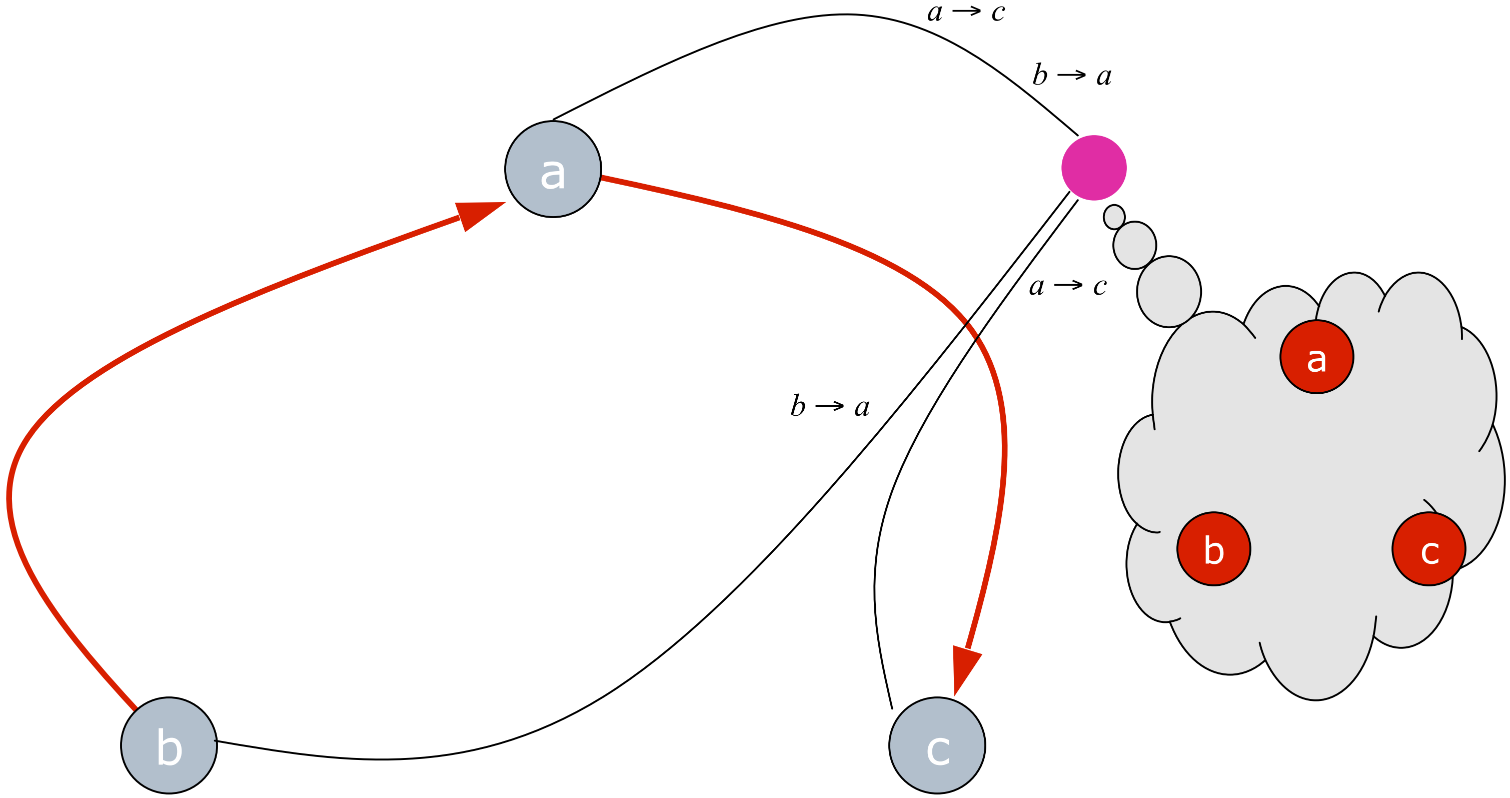
Reactively detecting properties

Rather than periodically constructing a global state to see if a property holds, deconstruct the property into local information, and have a *monitoring process* combine them.

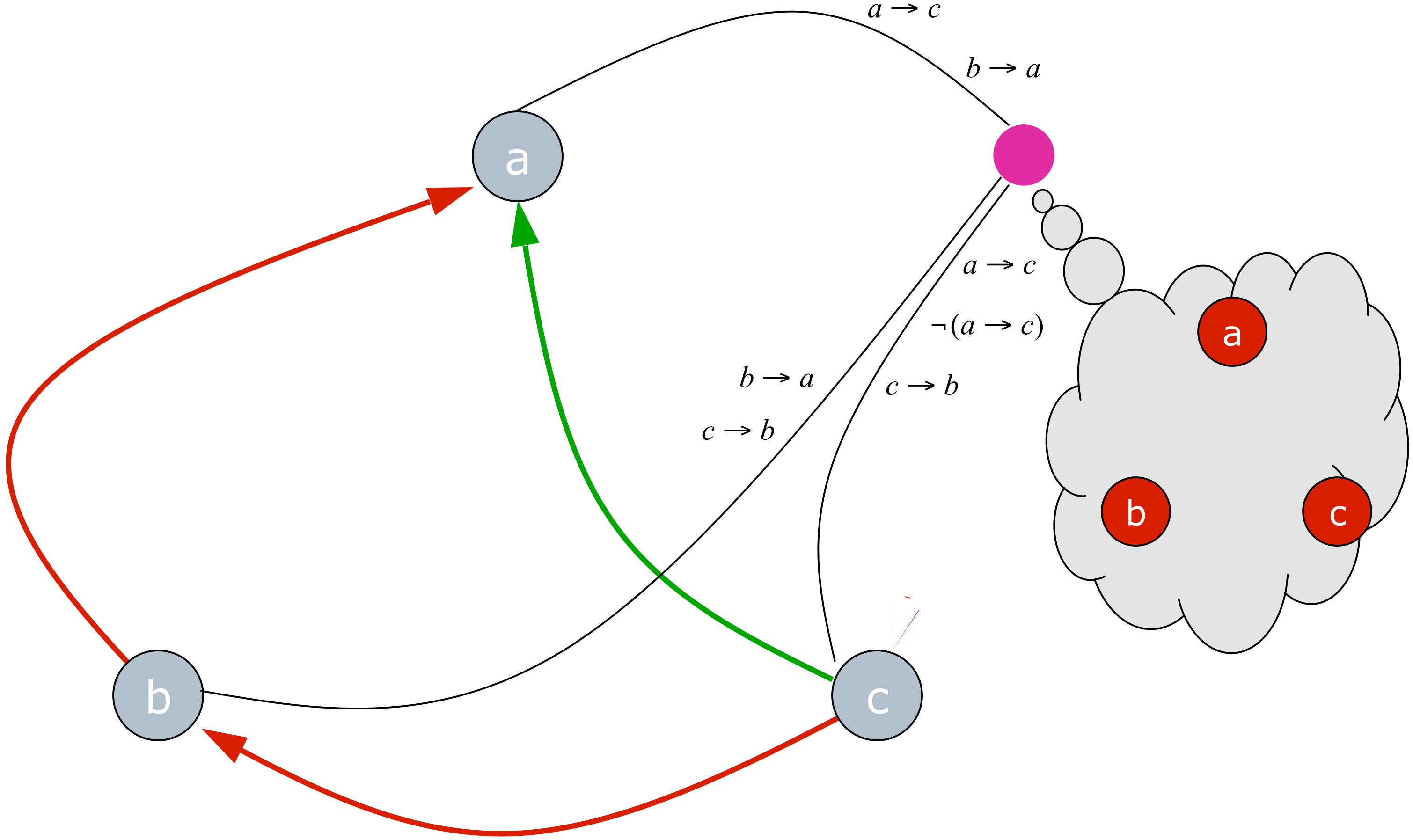
With RPC Deadlock, local information for *p* is the set of edges incident on *p* in the *waits-for* graph.

- When executes `RPCsend`.
- When receives a `RPCsend` message.
- When executes `RPCreply`.

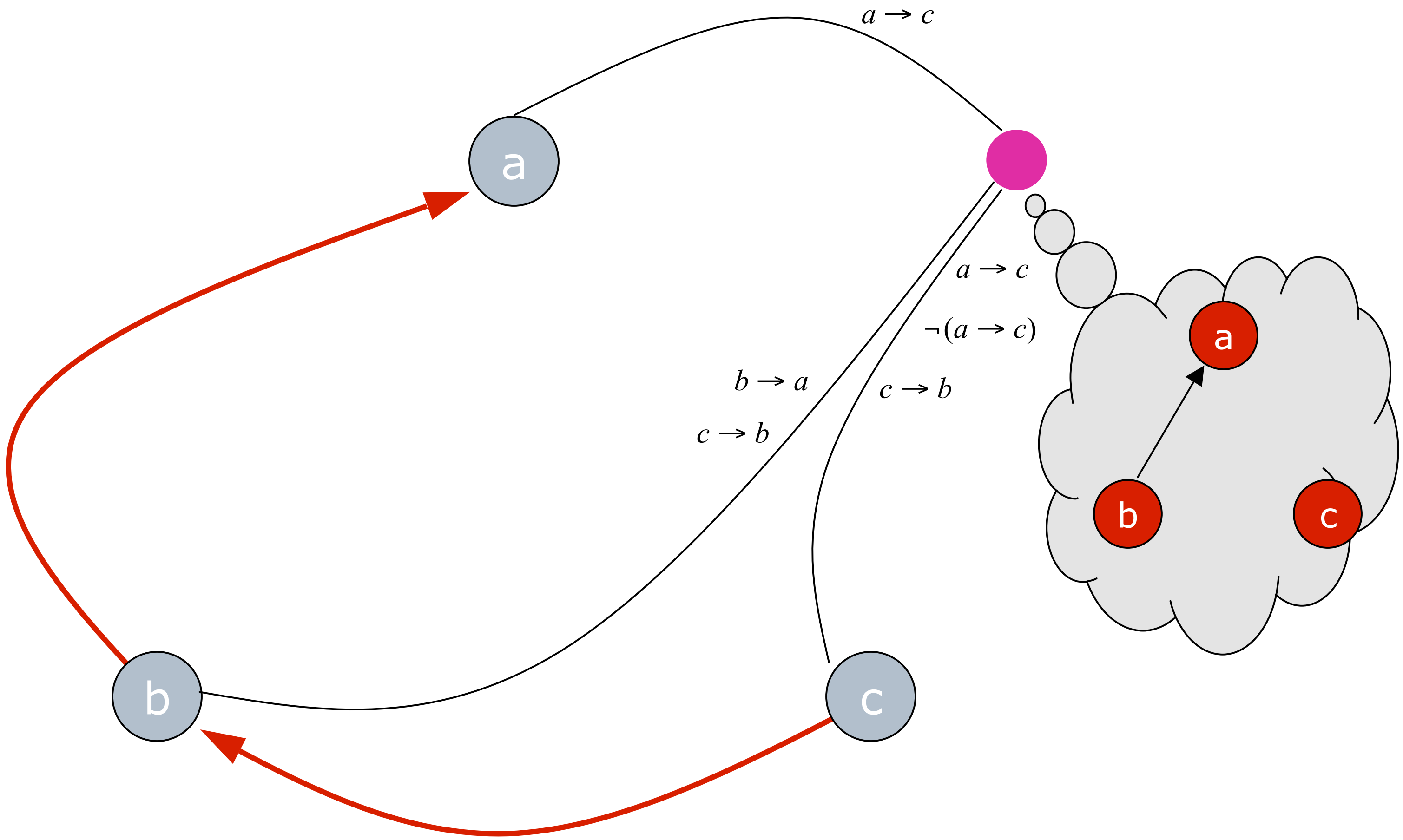
Misdetecting deadlock



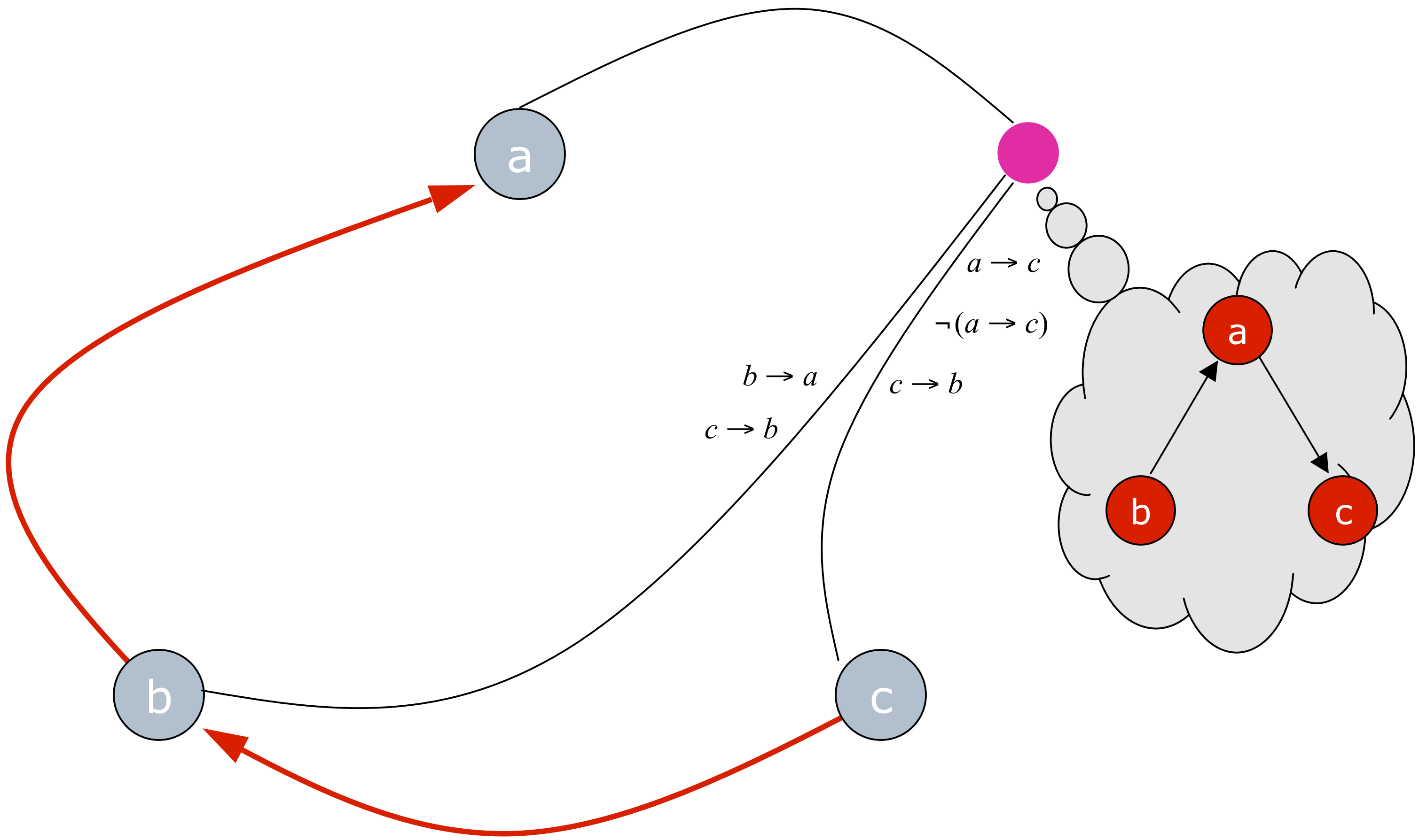
Misdetecting deadlock



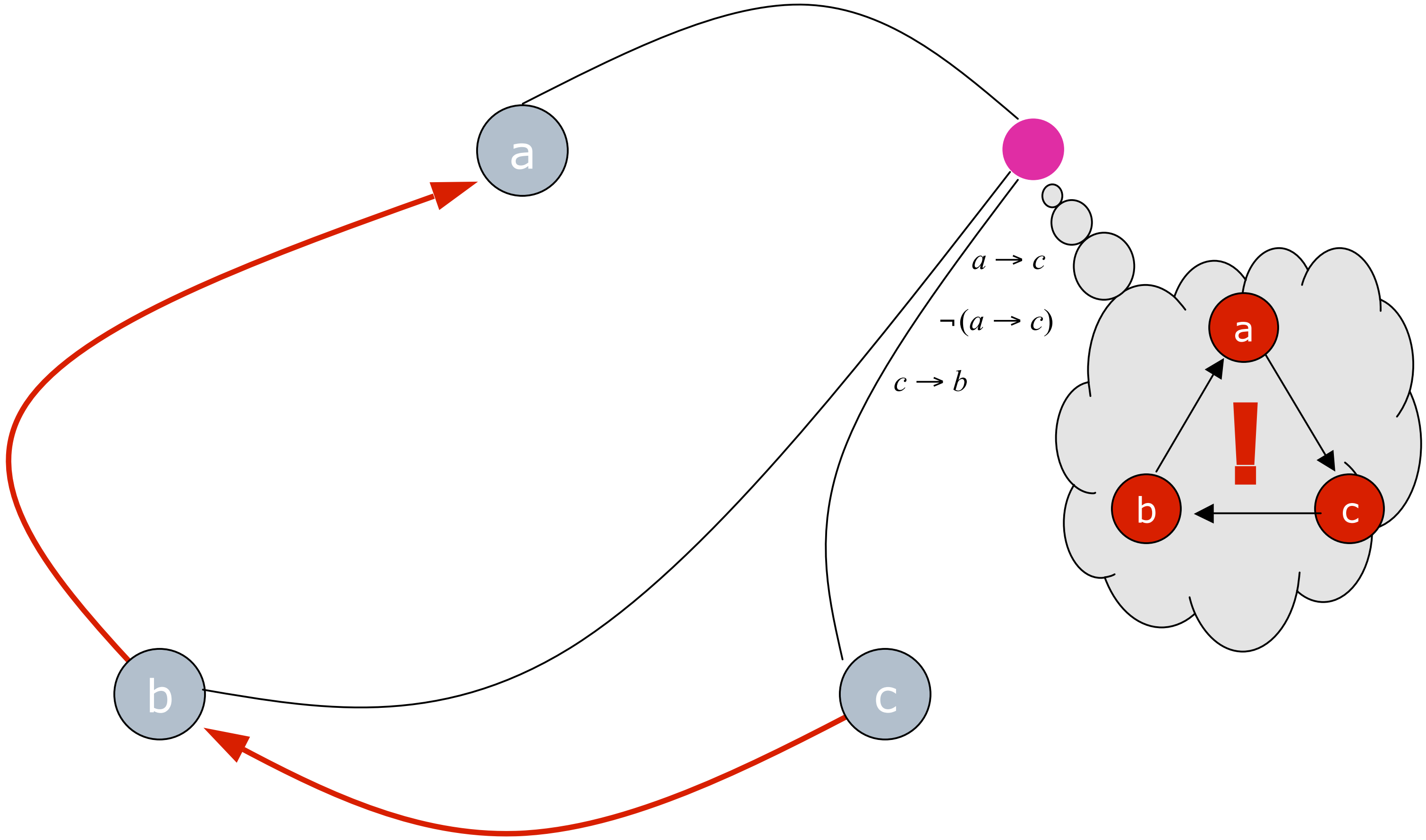
Misdetecting deadlock



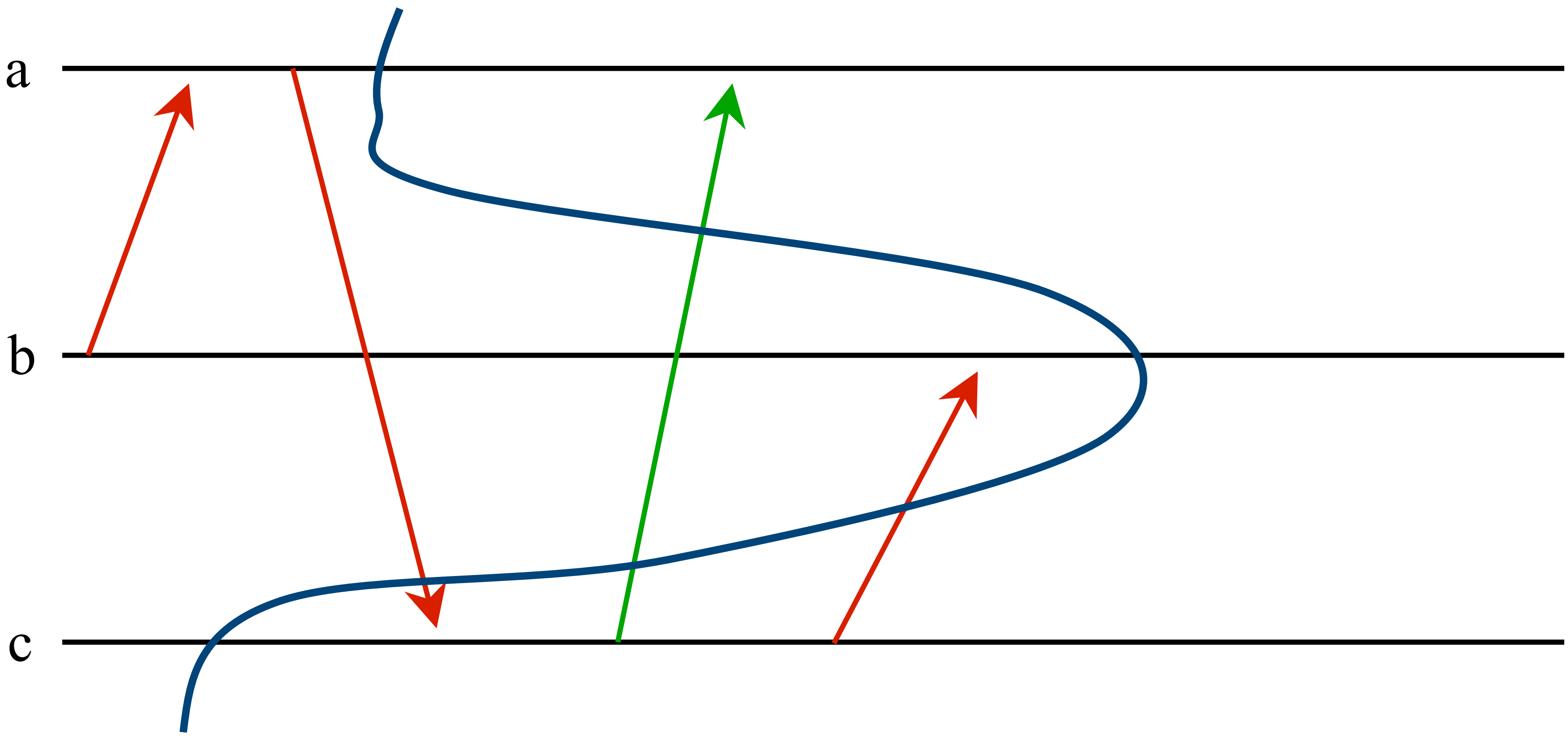
Misdetecting deadlock



Misdetecting deadlock



Misdetecting deadlock

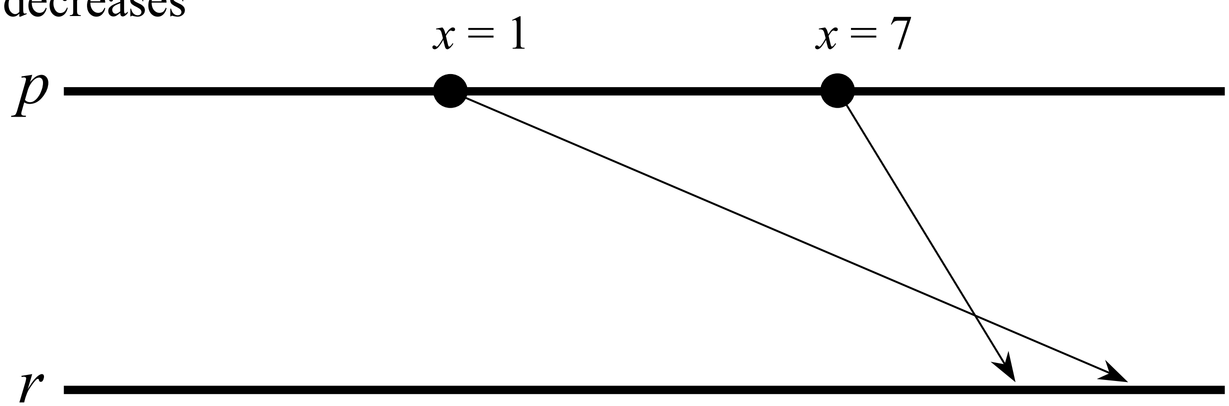


FIFO and Casual Delivery

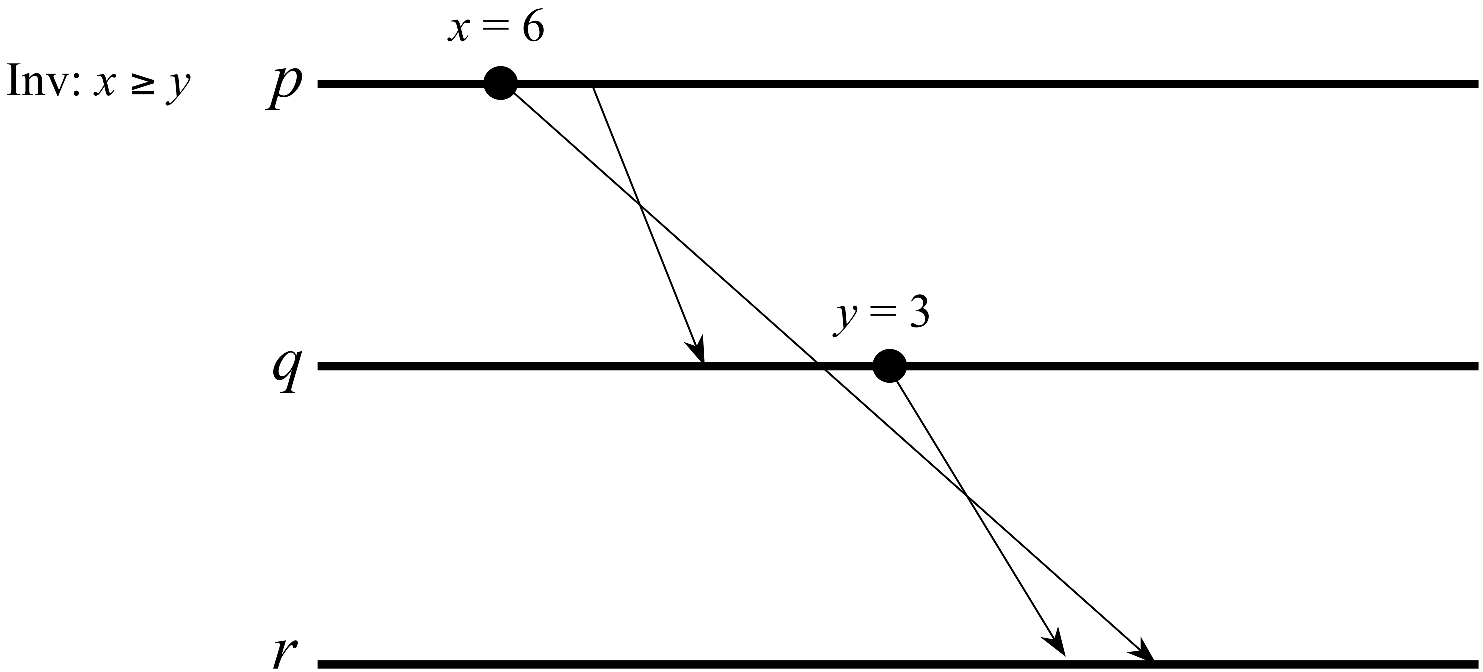
If a single process were to send information about its state, then non-FIFO delivery of messages could lead to an incorrect deduction.

FIFO: If p sends m_1 to r before p sends m_2 to r, then r does not deliver m_2 before m_1.

Inv: x never decreases



FIFO and Casual Delivery (continued)



C is *consistent* if, for all events e in C, all events e': $e' \rightarrow e$ are in C

Causal: If p sends m_1 to r before q sends m_2 to r, then r does not deliver m_2 before m_1.

Strong Clock condition

The *Clock condition* is $e' \rightarrow e \Rightarrow C(e') < C(e)$

This is the wrong direction for determining the causal ordering of two events.

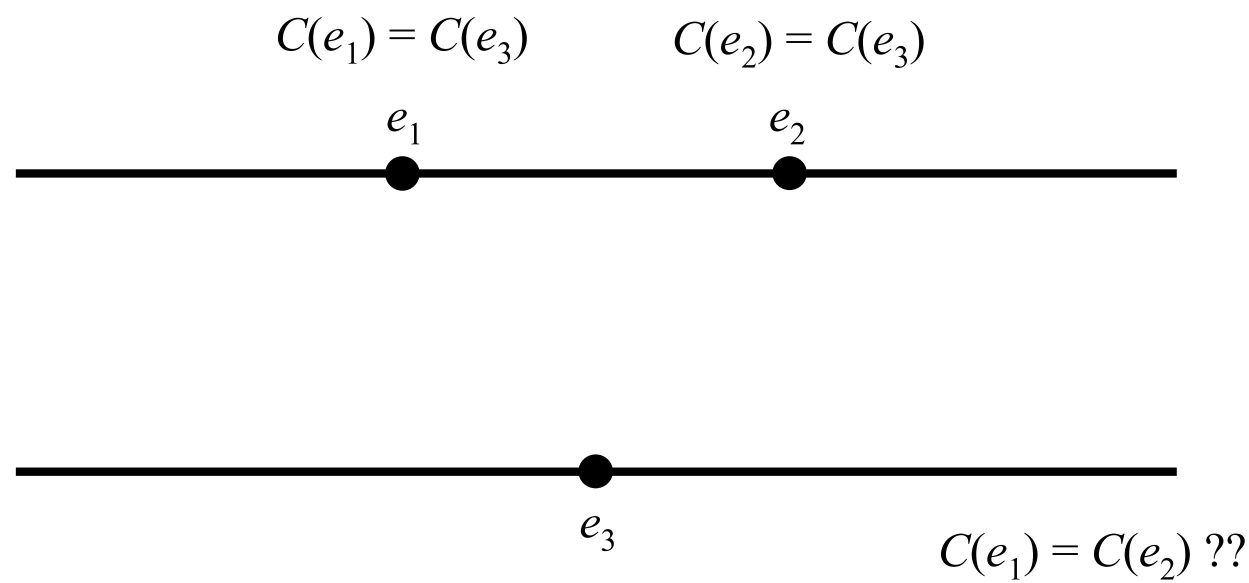
Strong Clock condition: $e' \rightarrow e \equiv C(e') < C(e)$

... which would allow determining causal order from the values of the logical clocks.

Strong Clock condition (continued)

The Strong Clock Condition implies

$$e' \parallel e \equiv (C(e') = C(e))$$



A Clock that satisfy the Strong Clock condition

Need a clock that can encode an *irreflexive partial order*.

Given n processes,

$$C_i = [K_1, K_2, ..., K_n]$$

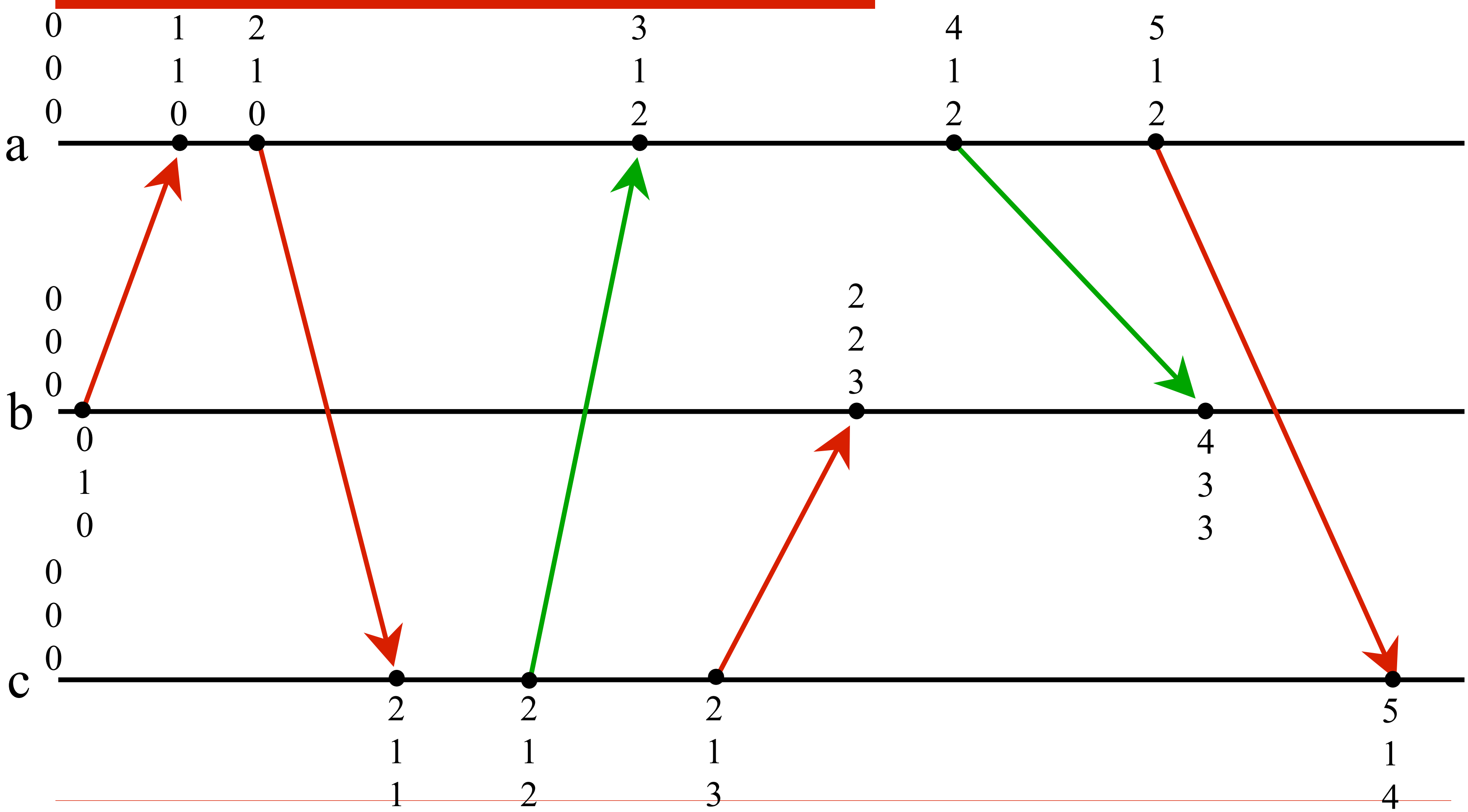
where

- K_i: the number of events p_i has executed.
- K_j, $j \neq i$: the number of events of p_j in p_i's causal past.

Vector clocks

- C_i is initially [0, 0, ..., 0].
- When p_i executes an event, it increments $C_i[i]$.
- When p_i sends a message m to p_j, it piggybacks C_i on m.
- When p_i receives a message m,
  $\forall\, j: 1 \le j \le n, j \ne i: C_i[j] = \max(C_i[j], m.C[j])$
  $C_i[i] = C_i[i] + 1.$

Vector clocks (continued)



Vector clock rules

If e_i is of p_i and $V(e)$ is the clock associated with e:

- $e_i \rightarrow e_j \equiv V(e_j)[i] \geq V(e_i)[i]$
- $e_i \parallel e_j \equiv (V(e_i)[i] > V(e_j)[i]) \wedge (V(e_j)[j] > V(e_i)[j])$

Given two events e and e':

- $e' \rightarrow e \equiv \forall\, i\colon 1 \leq i \leq n\colon V(e')[i] \leq V(e)[i]$
  $\wedge\ \exists\, i\colon 1 \leq i \leq n\colon V(e')[i] < V(e)[i]$
- $e' \parallel e \equiv \exists\, i\colon 1 \leq i \leq n\colon V(e')[i] < V(e)[i]$
  $\wedge\ \exists\, i\colon 1 \leq i \leq n\colon V(e)[i] < V(e')[i]$

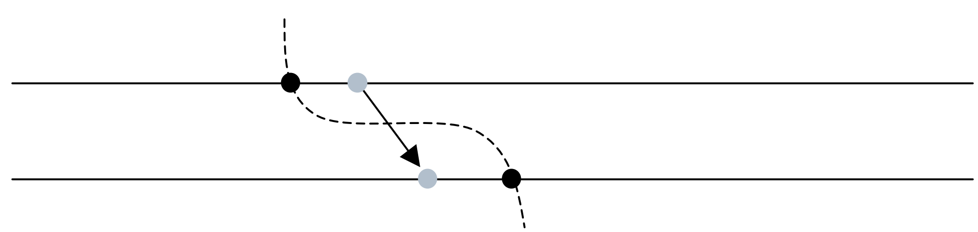
Vector clock rules (continued)

Two events e_i and e_j cannot be in the same consistent cut iff

$$(V(e_i)[j] > V(e_j)[j]) \lor (V(e_j)[i] > V(e_i)[i])$$

The tuple of events $\langle e_1, e_2, ..., e_n \rangle$ is the consistent cut iff

$$\forall\, i: 1 \leq i \leq n: V(e_i)[i] = \max(V(e_1)[i], V(e_2)[i], ... V(e_n)[i])$$

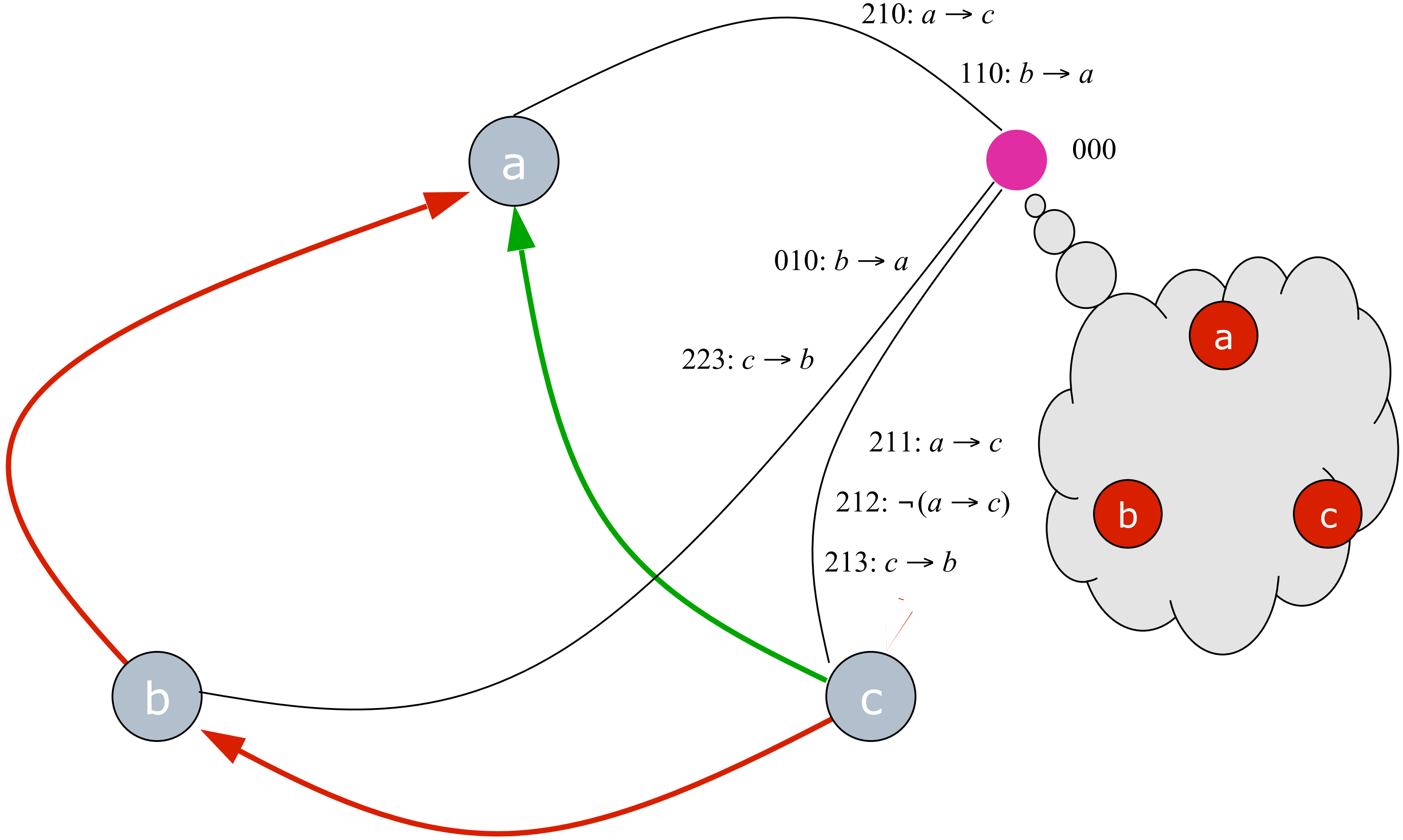


Implementing Causal Delivery

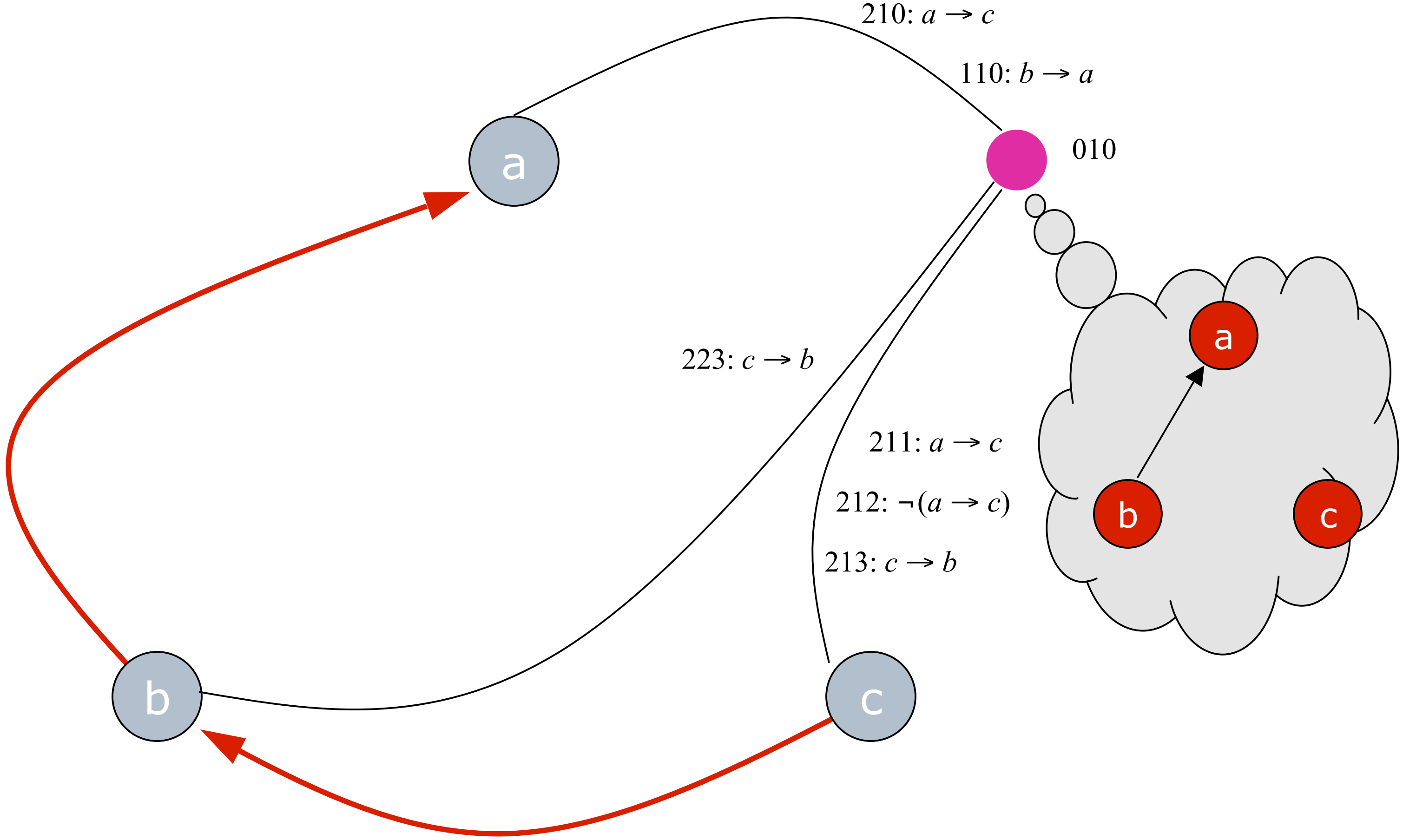
Causal delivery of messages to a particular destination (say p_1):

- Each process increments its local counter of a vector clock when it sends a message to p_1.
- p_1 maintains an array $\mathit{delivered}[1..n]$ where $\mathit{delivered}[i]$ is the number of message p_1 has delivered from p_i.
- When p_1 receives a message m, it adds the message to a set *received*.
- p_1 delivers message $m \in \mathit{received}$ sent by p_i when:
  - $m.C[i] = \mathit{delivered}[i] + 1$
  - $\forall\, j$: $1 \leq j \leq n, j \neq i$: $m.C[j] = \mathit{delivered}[j]$

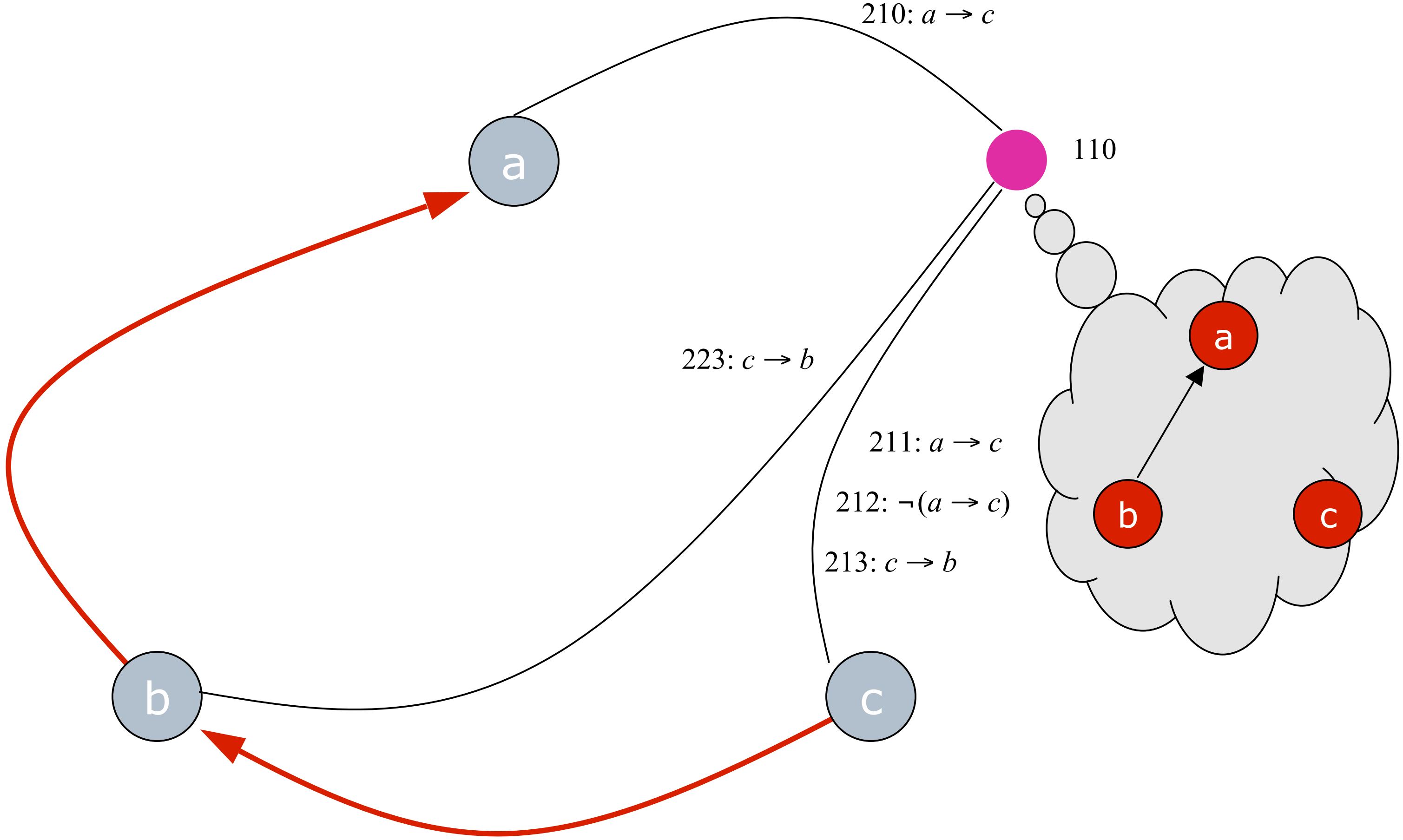
Causal delivery



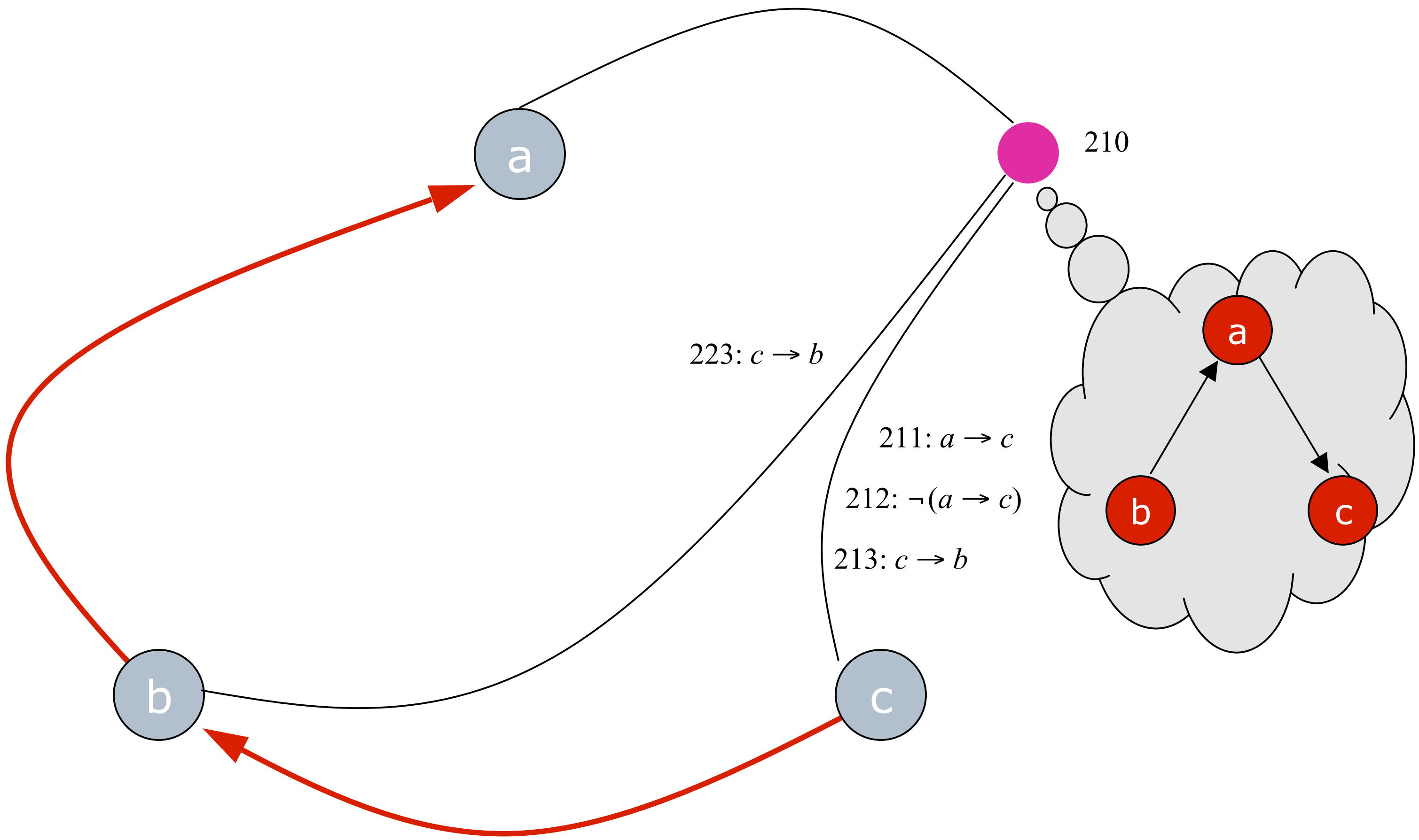
Causal delivery



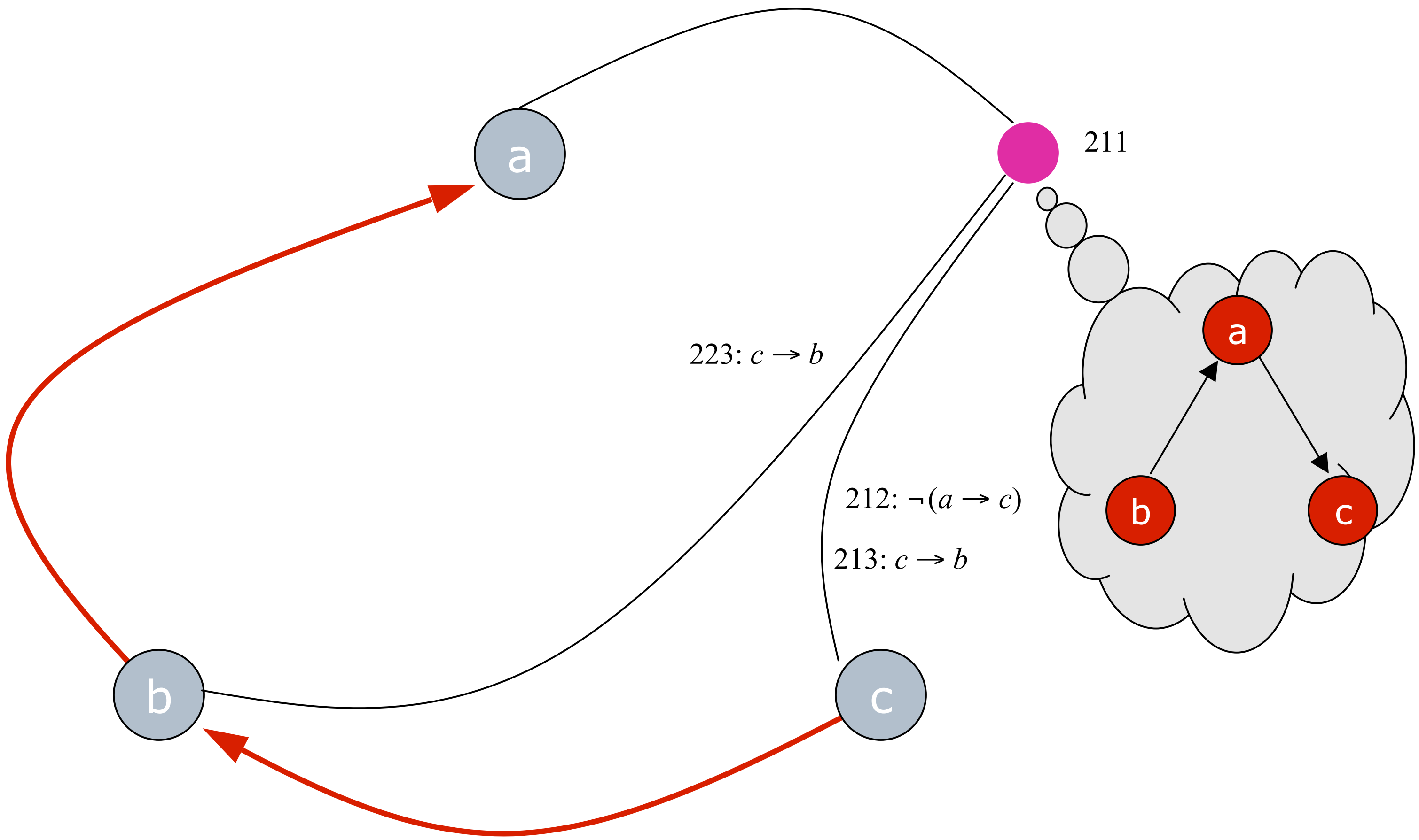
Causal delivery



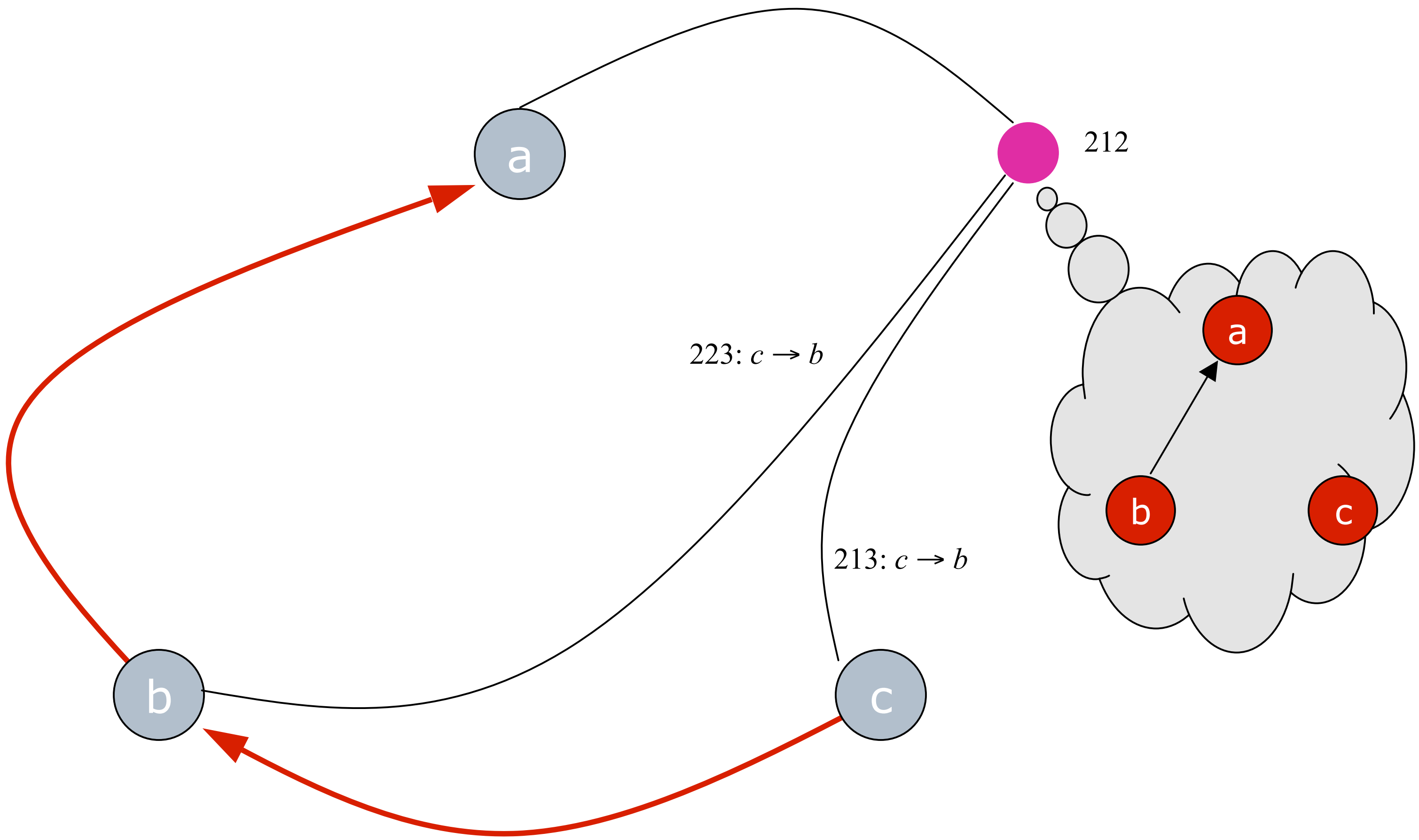
Causal delivery



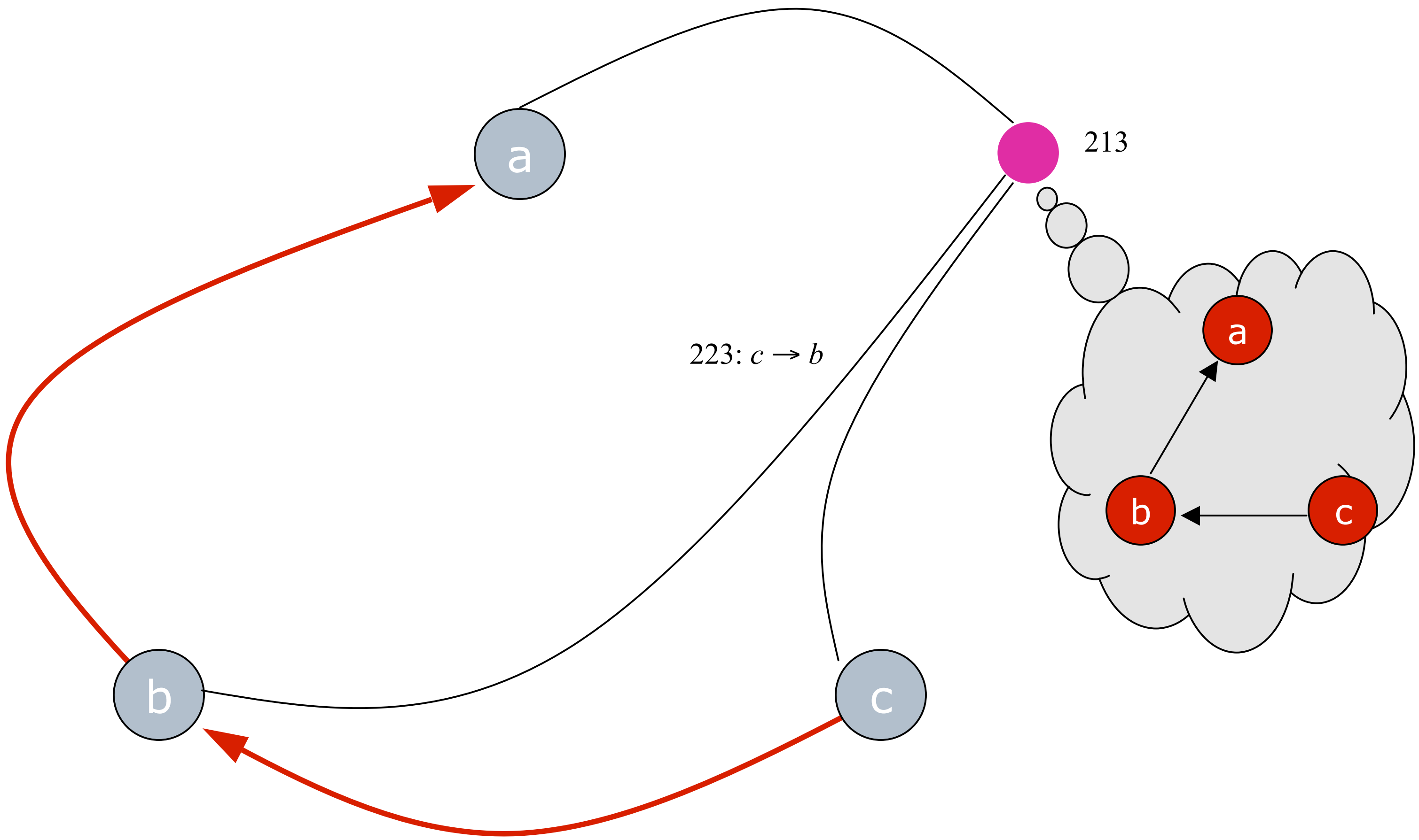
Causal delivery



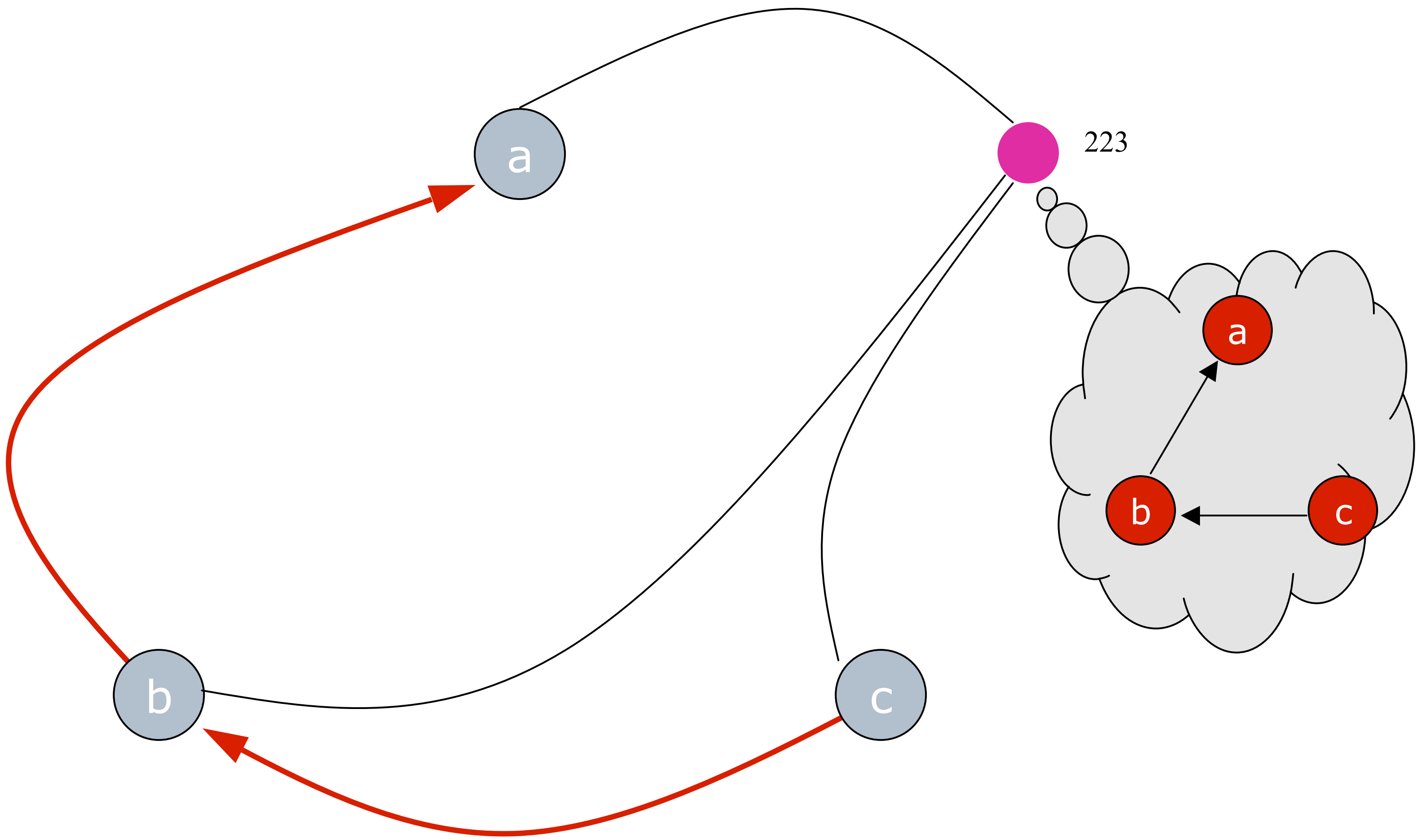
Causal delivery



Causal delivery



Causal delivery



Causal Delivery (continued)

- Implementing causal delivery order for *all* messages sent among processes requires a vector clock for every destination: $O(n^2)$ information.

Revisiting RPC deadlock detection

Neither of these protocol for RPC deadlock detection are very efficient.

A class of stable properties are easier to detect.

Let a *relevant* event be an event that is used in formulating the stable property.

Keith Marzullo and Laura S. Sabel. Efficient detection of a class of stable properties. *Distributed Computing* 8(2): 81-91, October 1994.

Locally stable properties

- A *locally stable* property is a stable property in which a process involved in the stable property will stop executing relevant events.
  - Deadlock is locally stable.
  - Lossy token passing, *there are no more than 2 tokens* is not locally stable.

A bit more formally...

Given a stable property P:

- S: a global state.
- S_P: values of variables in S that are in the formulation of P.
- $S \mid A$: subset of S that consists of the states of processes in A (where channel states are encoded in process states).

P is locally stable if, for any S that satisfies P:

- Let A be the set of processes that execute no relevant events in an execution starting with S.
- S can be determined by considering only the variables in $S_P \mid A$.

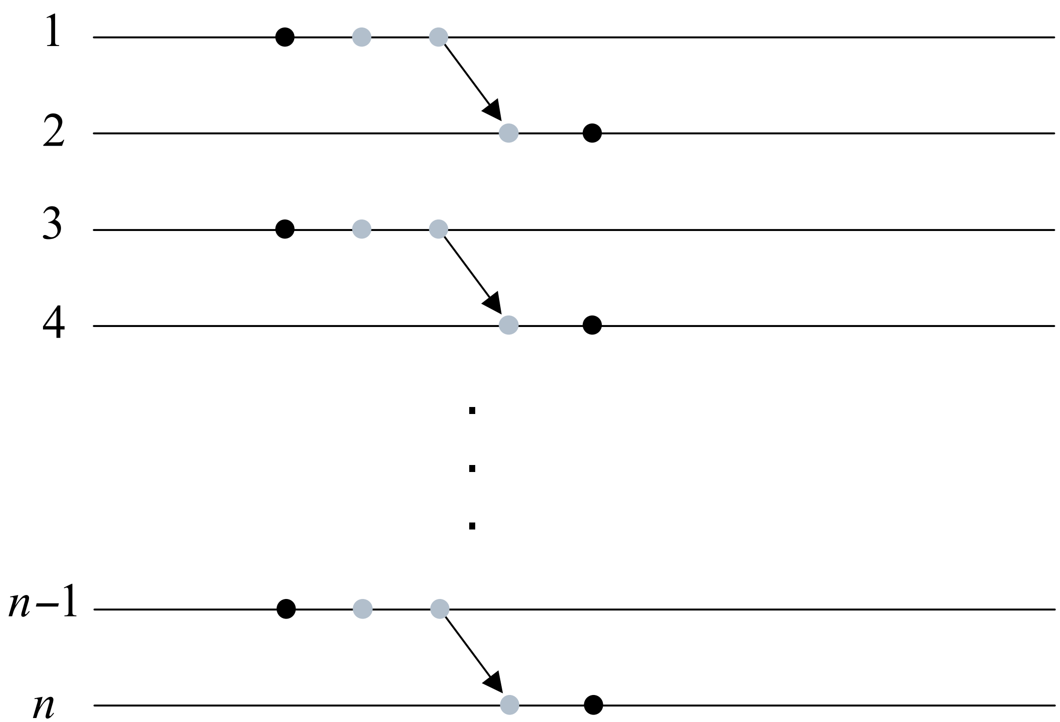
Basic protocol I

- When i executes a relevant events, it records into a buffer B_i its state $B_i.s$ and its vector clock $B_i.C$.
  - We have the vector clock only count relevant events.
  - Since P is locally stable, once P holds, there will be a consistent subcut - i.e., a subset of $\{B_1, B_2, ... B_n\}$whose timestamps indicate they are pairwise consistent - that establishes P.
  - If A is a consistent subcut, then $A' \subseteq A$ is also a consistent subcut.

Basic protocol II

- Periodically a process *i* collects $\{B_1, B_2, \ldots B_n\}$ and determines whether there exists a *maximal consistent subcut* in which P holds.
  - Safety: if P holds in this consistent subcut, it holds now.
  - Liveness: from P being locally stable.

Maximal subcuts



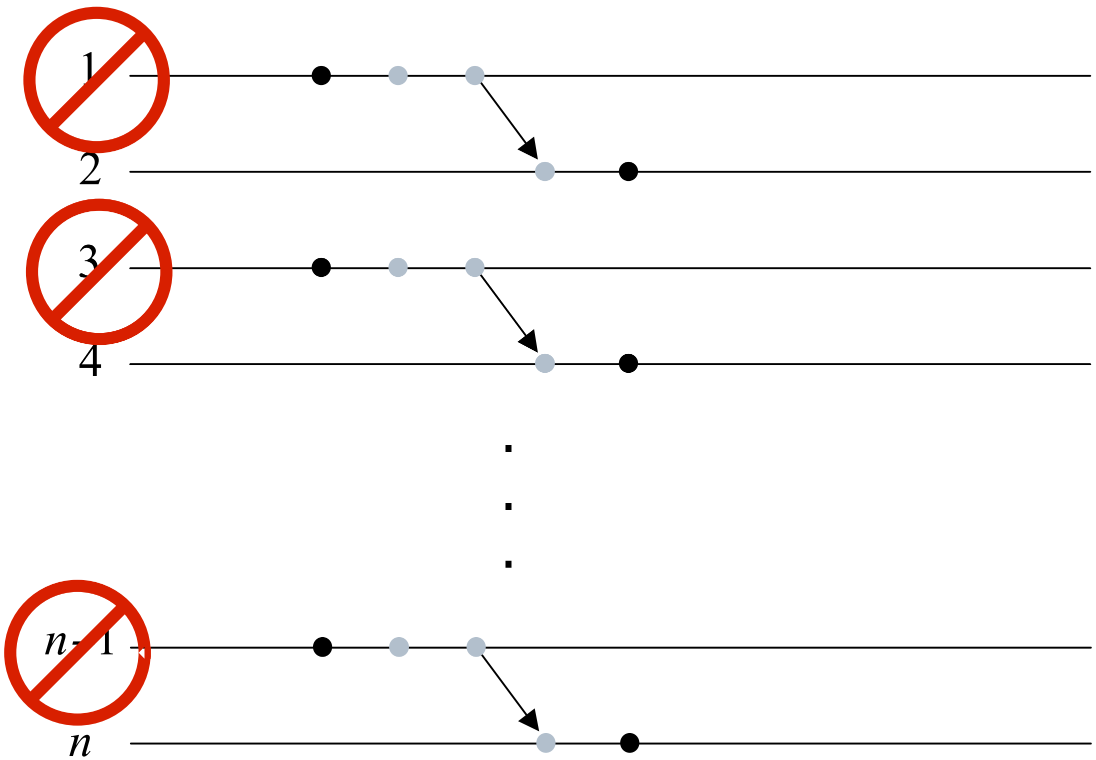
$C(e_2)[1] > C(e_1)[1]$

$C(e_4)[3] > C(e_3)[3]$

$C(e_n)[n-1] > C(e_{n-1})[n-1]$

$2^{n/2}$ maximal subcuts

Latest subcut



$C(e_2)[1] > C(e_1)[1]$

$C(e_4)[3] > C(e_3)[3]$

$C(e_n)[n-1] > C(e_{n-1})[n-1]$

$$\{\forall\ i\ B_i.s\text{: } \forall\ j\text{: } B_i.C[i] \geq B_j.C[i]\}$$

Centralized protocol

- When i executes a relevant events, it records into a buffer B_i its state $B_i.s$ and its vector clock $B_i.C$.
- Periodically a process i collects $\{B_1, B_2, \ldots B_n\}$ and extracts the subset $\{B_i.s: \forall\, j: B_i.C[i] \geq B_j.C[i]\}$.
- If P holds on this subset, then detect P.

Decentralization

Define a token $K = \langle D_1, D_2, ..., D_n \rangle$ where D_i is either a pair $\langle B_i.s, B_i.C[i] \rangle$ or $\perp$.

- A process i generates an empty token $\langle \perp, \perp, ..., \perp \rangle$, sets $K.D_i$ to $\langle B_i.s, B_i.C[i] \rangle$, and passes it to another process.
- When j receives K:
  - set $K.D_j$ to $\langle B_j.s, B_j.C[j] \rangle$
  - for all D_k: $D_k \neq \perp \wedge D_k.C[k] < B_j.C[k]$, set $K.D_k$ to $\perp$.
  - If P holds on non-$\perp$ values of K, then detect condition else forward to some p_m: $D_m = \perp$
    - or, discard it.

RPC Deadlock

State:

- $rs_i[i \in 1..n]$: number of `RPCreply` i sent to j
- $rr_i[i \in 1..n]$: number of `RPCreply` i received from j
- wf_i: process i (if any) is waiting on.
  - Relevant events are those that change these variables.

i waits-for j when $(wf_i = j) \wedge (rs_j[i] = rr_i[j])$

deadlock iff cycle in waits-for graph

RPC Deadlock: Protocol I

When $wf_i = j$ for unexpected time, i:

- creates new token K
- sets $K.D_i$ to $\langle rr_i[j], B_i.C[i]\rangle$
- sends K to j
  - Note we send only two integers!
    - First to determine if i waits on j.
    - Second as part of the protocol.
- Still maintaining vector clocks that count relevant events, buffering relevant event state, etc.

RPC Deadlock: Protocol II

When i receives a token K from j:

if $K.D_i = \perp$

if $((wf_i \neq \emptyset) \wedge (K.D_j.rr_j[i] = B_i.rs_i[j])$
$\wedge\ (\forall\ k: K.D_k \neq \perp: K.D_k.C[k] \geq B_i.C[k]))$

then

$K.D_i = \langle B_i.rr_i[B_i.wf_j], B_i.C[i]\rangle$;

pass K to $B_i.wf_j$;

else drop K;

else

if $(K.D_j.rr_j[i] = B_i.rs_i[j])$ then *detect deadlock*;

else drop K;

haven't visited i yet

RPC Deadlock: Protocol II

i is blocked

j is blocked

When i receives a token K from j:

if $K.D_i = \perp$

if $((wf_i \neq \emptyset) \wedge (K.D_j.rr_j[i] = B_i.rs_i[j])$
$\wedge\ (\forall\ k\text{: } K.D_k \neq \perp\text{: } K.D_k.C[k] \geq B_i.C[k]))$

then

state of i consistent with other process states

$K.D_i = \langle B_i.rr_i[B_i.wf_j], B_i.C[i] \rangle$;

pass K to $B_i.wf_j$;

else drop K;

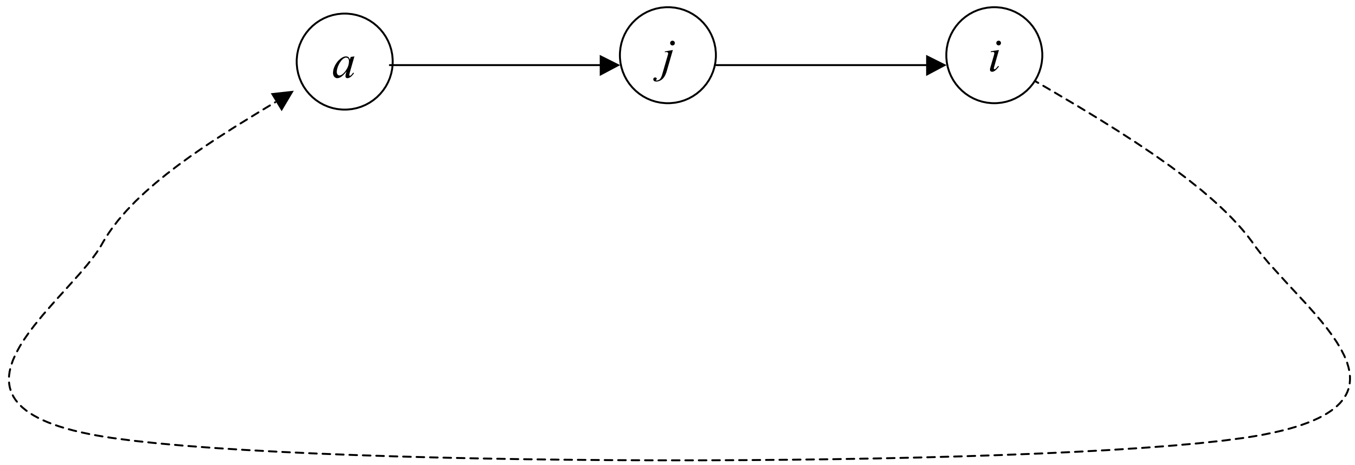
j waits for i: there's a cycle

else

if $(K.D_j.rr_j[i] = B_i.rs_i[j])$ then *detect deadlock*;

else drop K;

Why compare timestamps?



- If $K.D_j.rr_j[i] = B_i.rs_i[j]$ then j hasn't subsequently executed a relevant event.
- Since j hasn't executed a relevant event, neither has a.
- … and so on along the loop.
  - Don't need vector clocks.
  - Don't need separate event buffer.

RPC Deadlock: final protocol

When $wf_i \neq \emptyset$ for unexpected time
i sends $(rr_i[wf_i], i)$ to wf_i.

When i receives (s, a) from j:

if $(a \neq i)$

if $((wf_j \neq \emptyset) \wedge (s = rs_i[j]))$
then send $(rr_i[wf_i], a)$ to wf_i;

else

if $(s = rs_i[j])$ then *detect deadlock*;

K. M. Chandy, J. Misra, and L. Haas. Distributed deadlock detection. *ACM TOCS* 1(2):144-156 (1983).