

State of a truly concurrent system

In a system with true concurrency (distributed system, multiprocessor computer), what does it mean for the system to have a particular state?

Look at this problem in the context of an important class of states: *stable properties*

$$F \text{ stable} \equiv F \Rightarrow \square F$$

Distributed System

- Set of processes that execute independently.
- Interconnected with some kind of communications network that allow the processes to communicate by *sending* and *receiving* messages.

RPC Deadlock

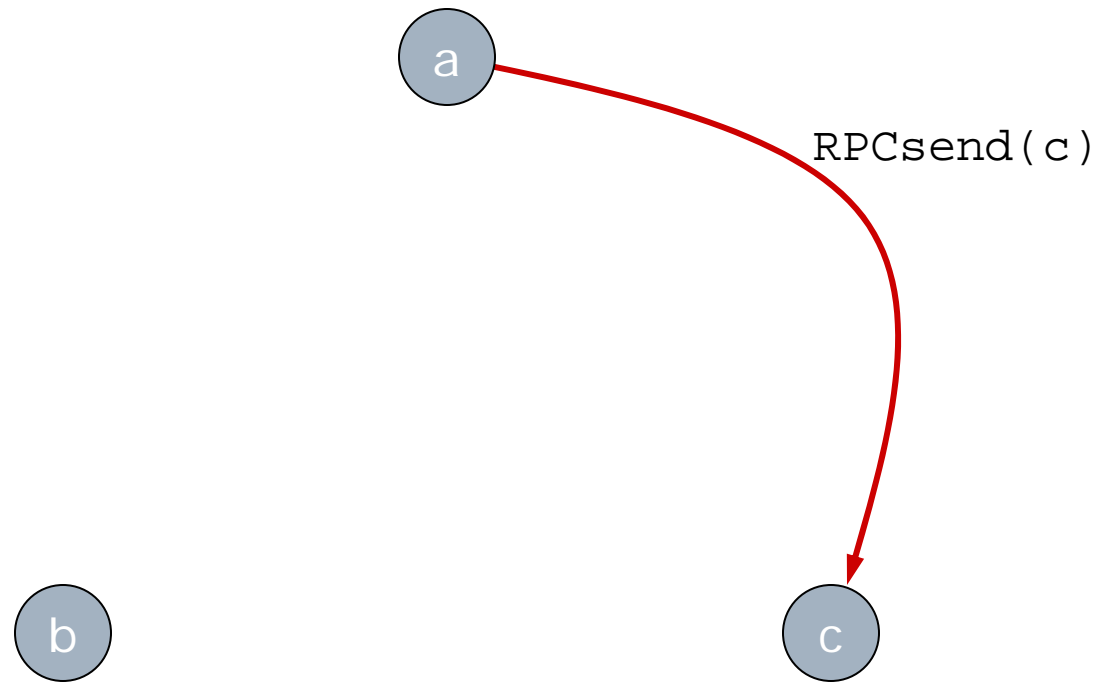
Design a protocol to detect stable property *RPC Deadlock*

- An application uses processes that communicate via *blocking sends*

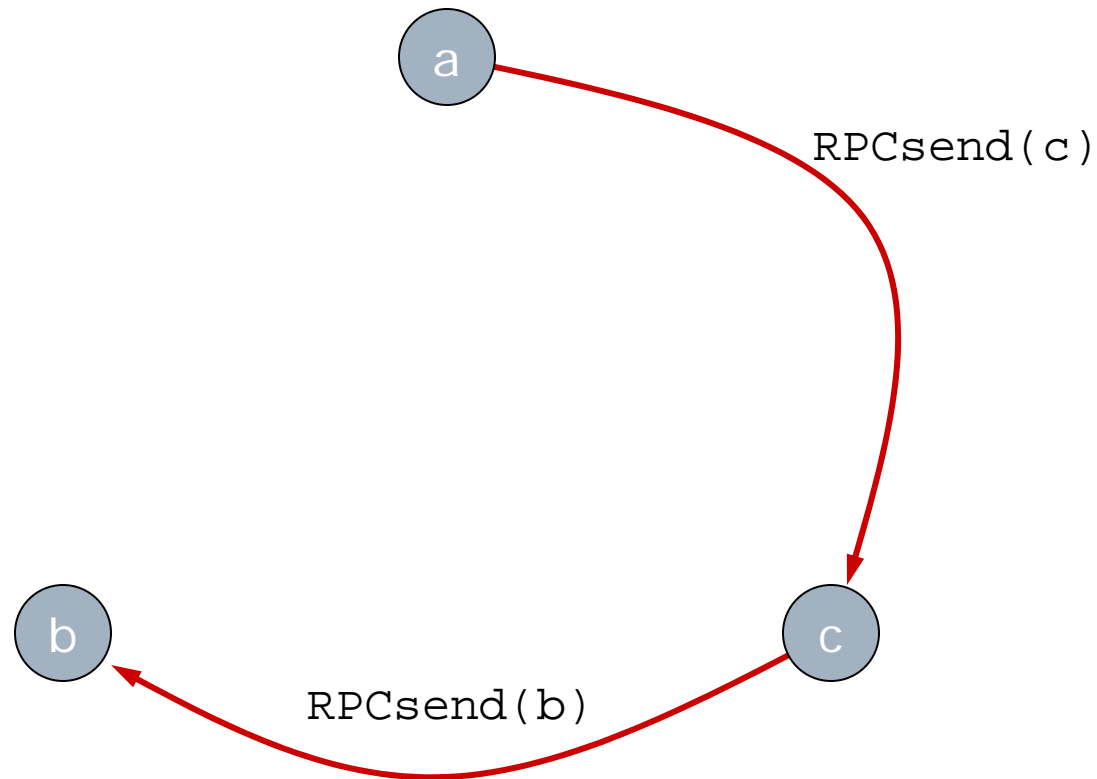
```
    r = RPCsend(p, m);  
    (m, p) = RPCreceive();  
    RPCreply(r);
```

- p waits-for q if p has executed $\text{RPCsend}(q, m)$ and q has not yet executed $\text{RPCreply}(r)$.
- *Deadlock* if cycle in waits-for graph.

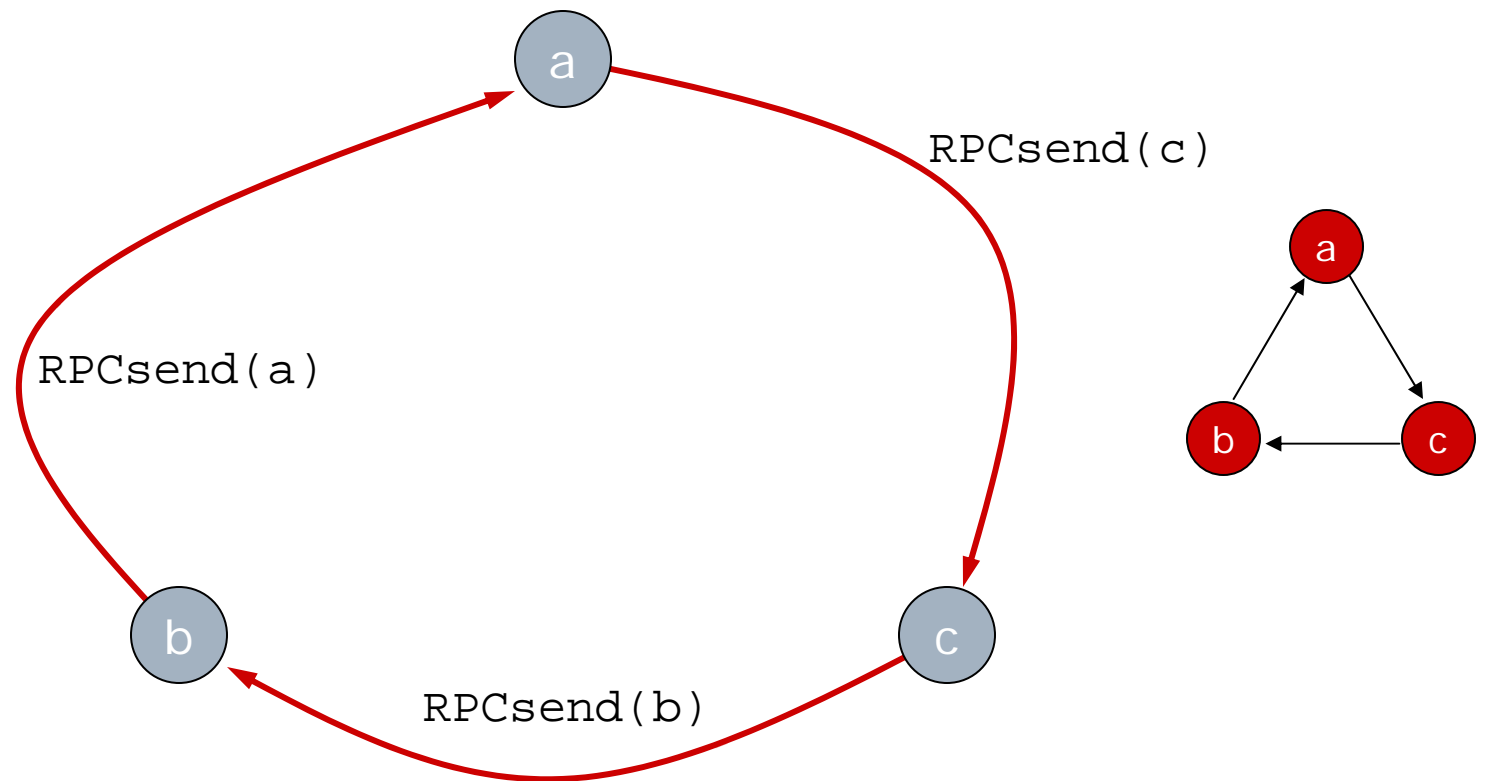
Example of deadlock



Example of deadlock



Example of deadlock



A simple protocol

A separate *monitoring process* samples the states of the processes p .

- Who p is waiting on;
- Which RPCsend requests have been received (even if not RPCreceive'd).

This protocol does not use RPC: it uses lower-level nonblocking *send* and asynchronous *receive* primitives.

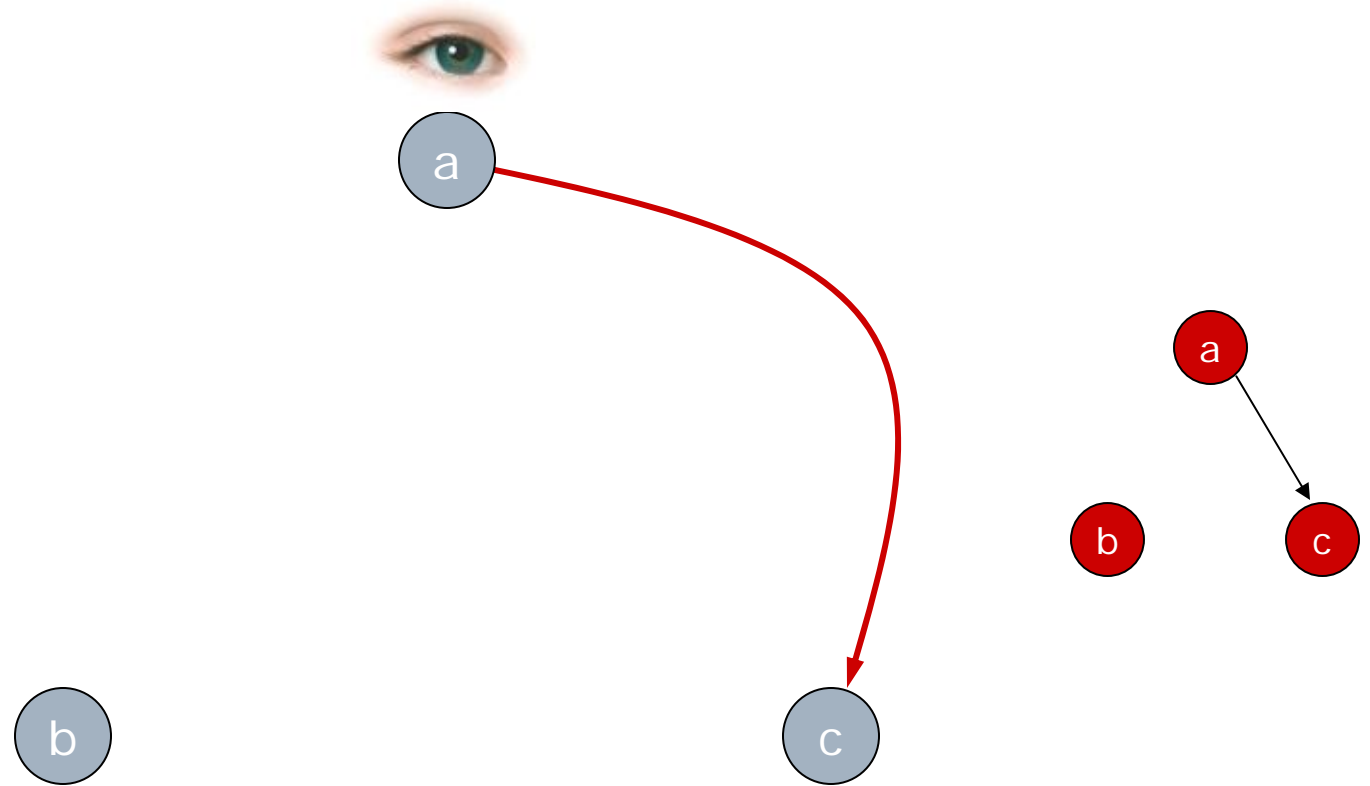
A simple protocol (2)

A process periodically runs the following protocol:

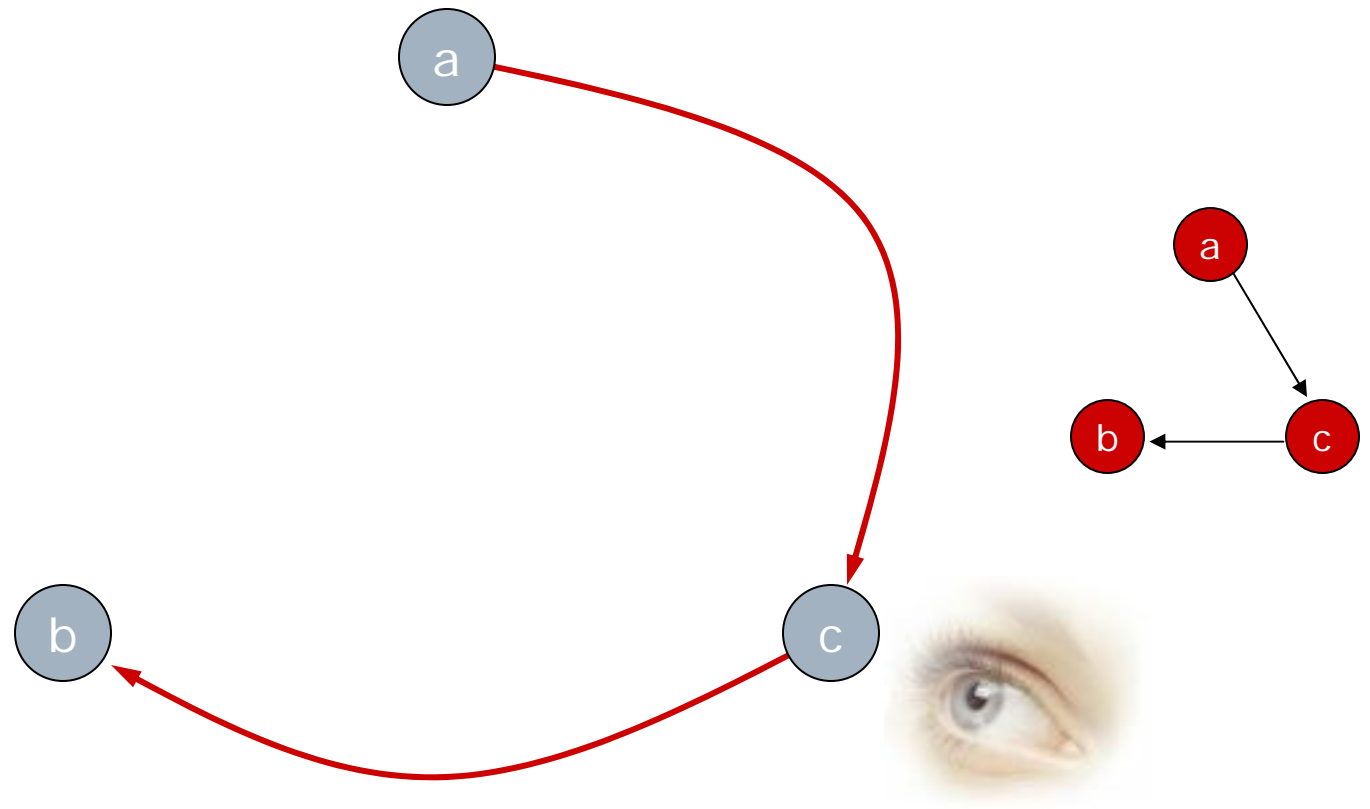
```
wfg = empty;
for (each application process  $p$ ) {
    send message to  $p$  requesting
        on who (if any) it is waiting and who (if any) are waiting on it.
    if ( $p$  waiting on some  $q$ ) add edge to  $wfg$  from  $p$  to  $q$ ;
    for all ( $r$  waiting on  $p$ ) add edge to from  $r$  to  $p$ ;
}
if ( $wfg$  has cycle) detect deadlock;
```

... NB the process running this can be one of the application processes.

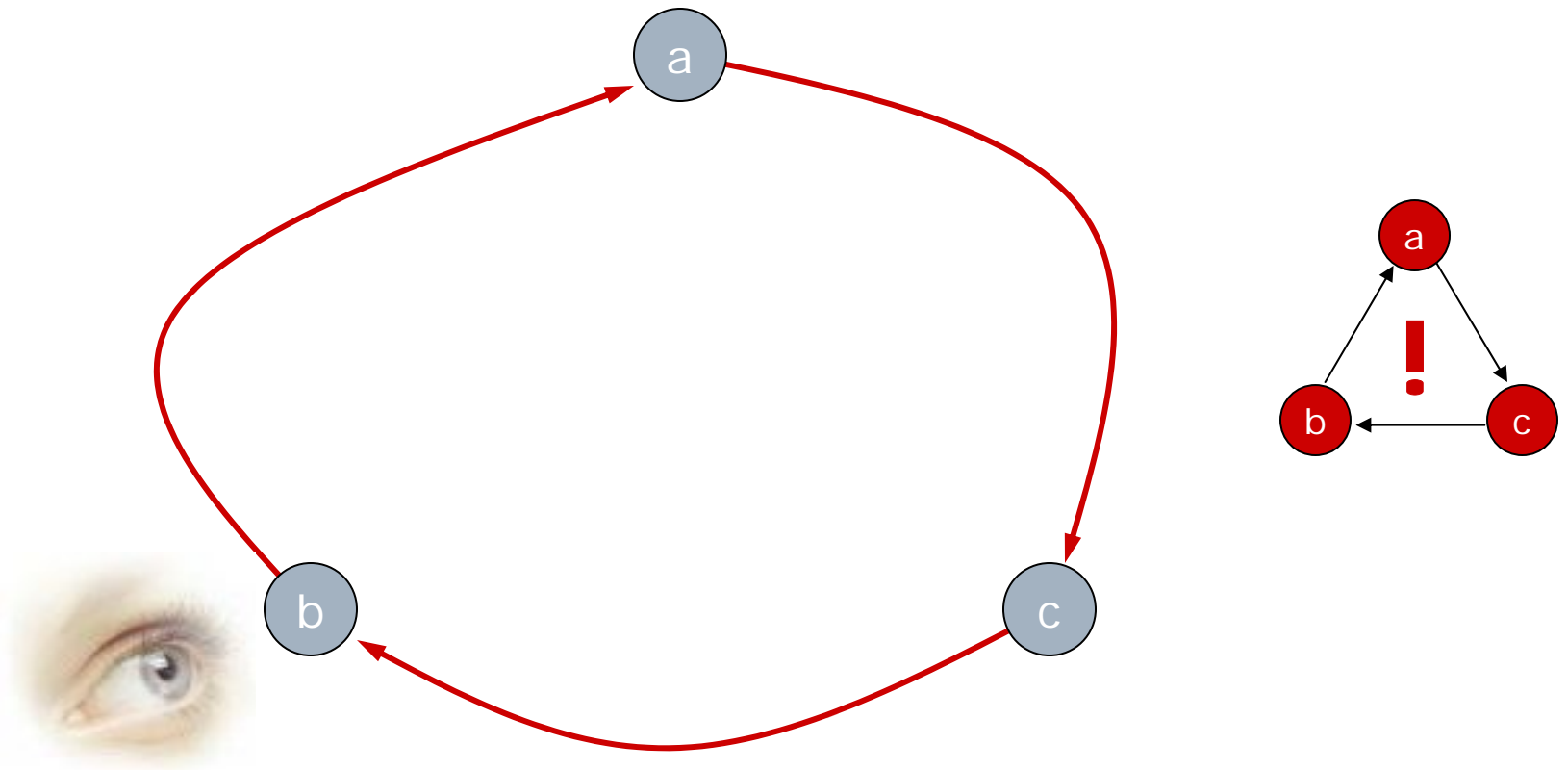
Detecting deadlock



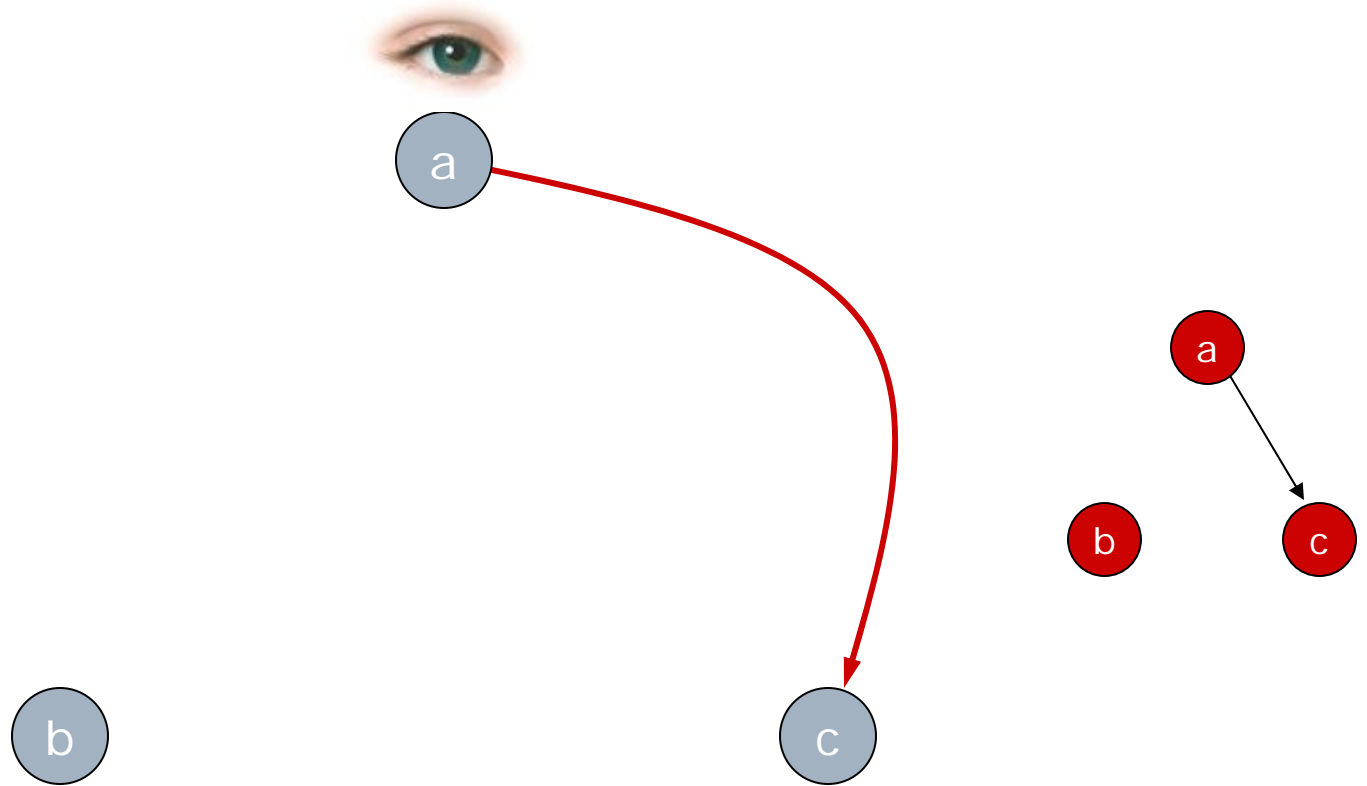
Detecting deadlock



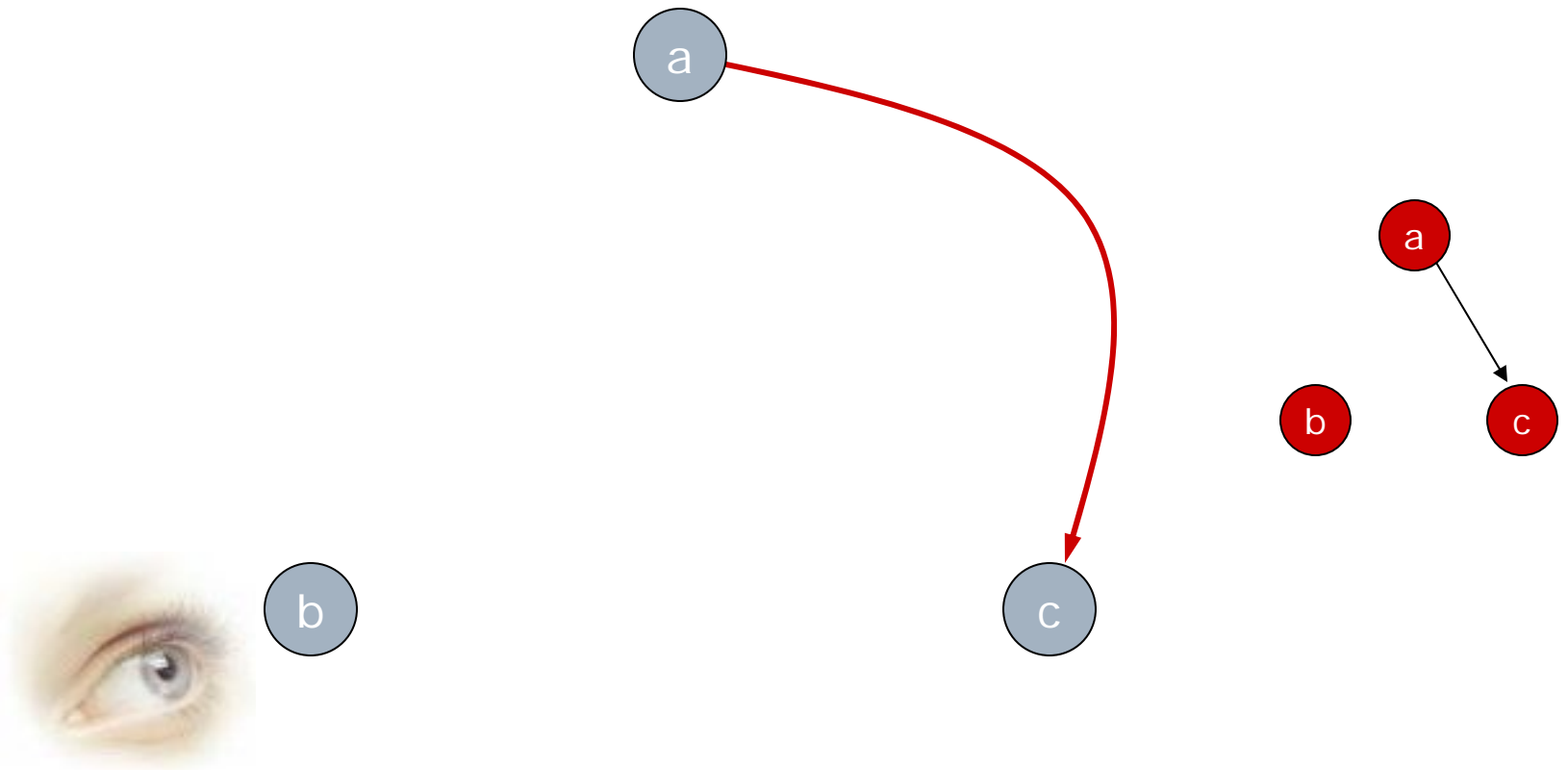
Detecting deadlock



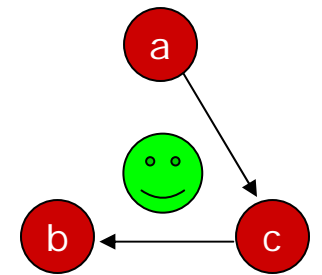
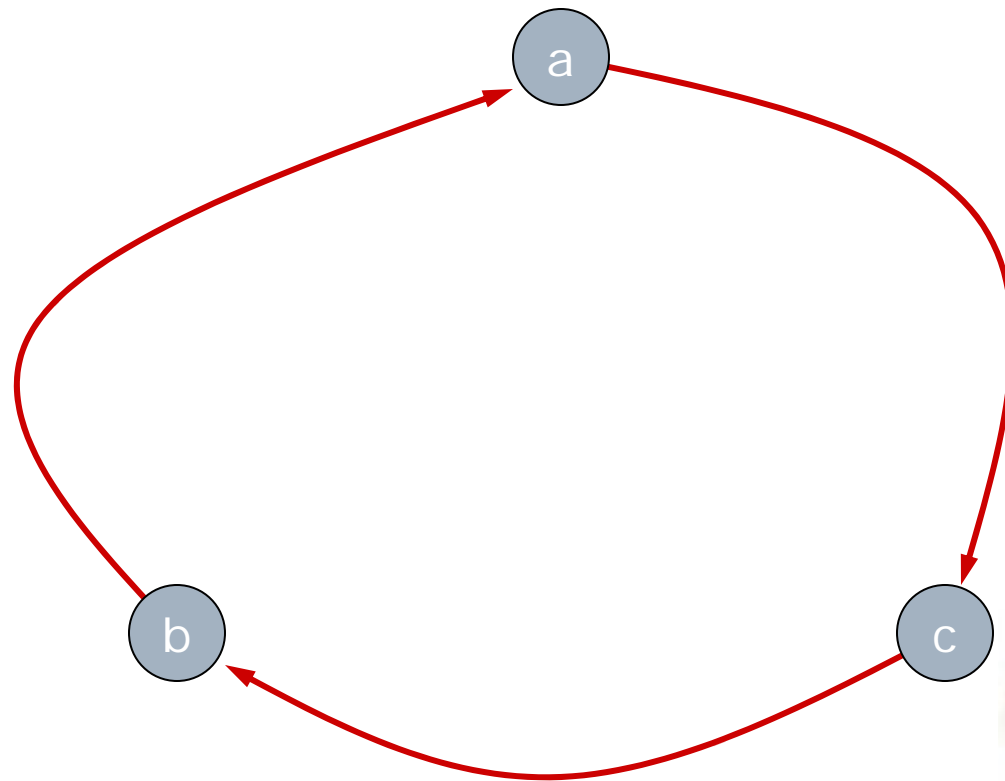
Missing deadlock



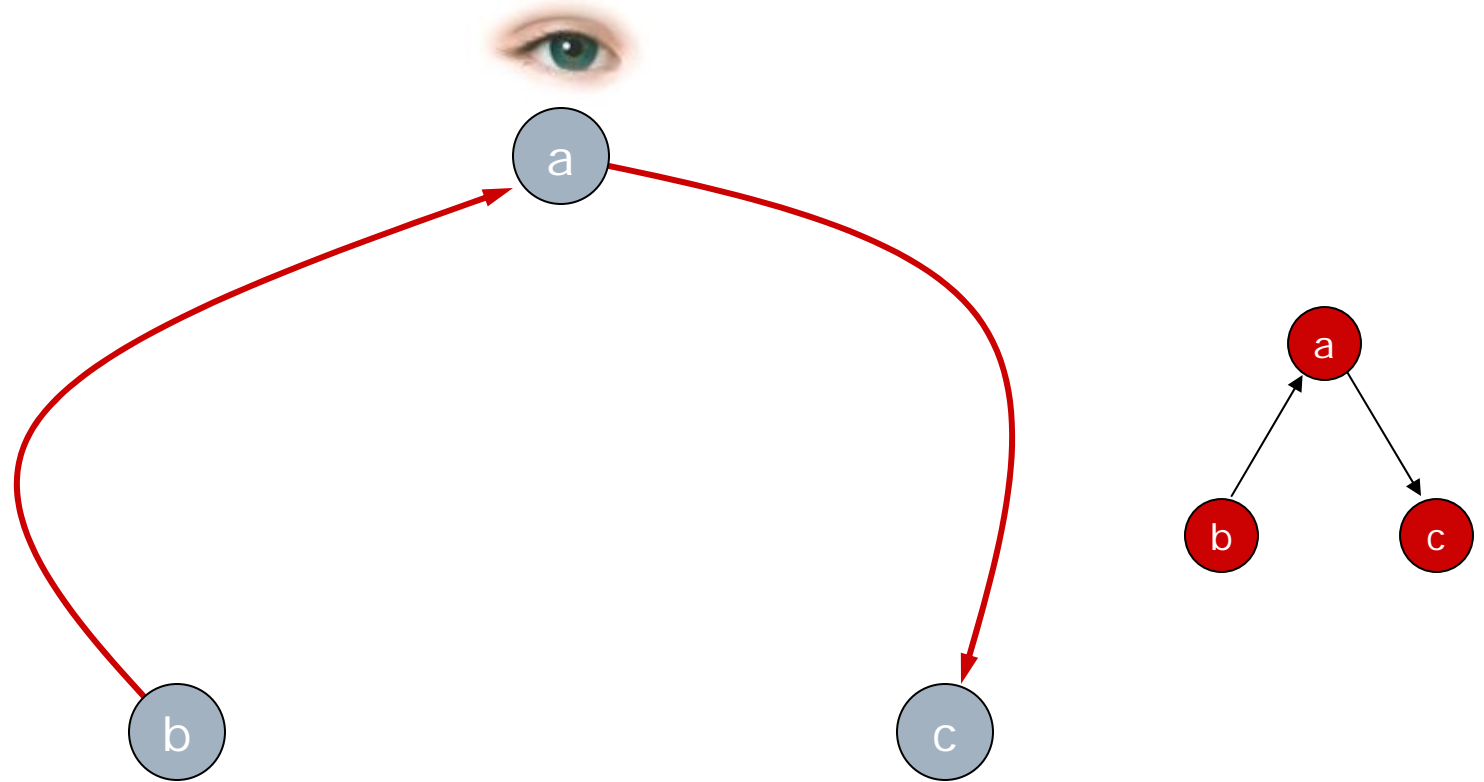
Missing deadlock



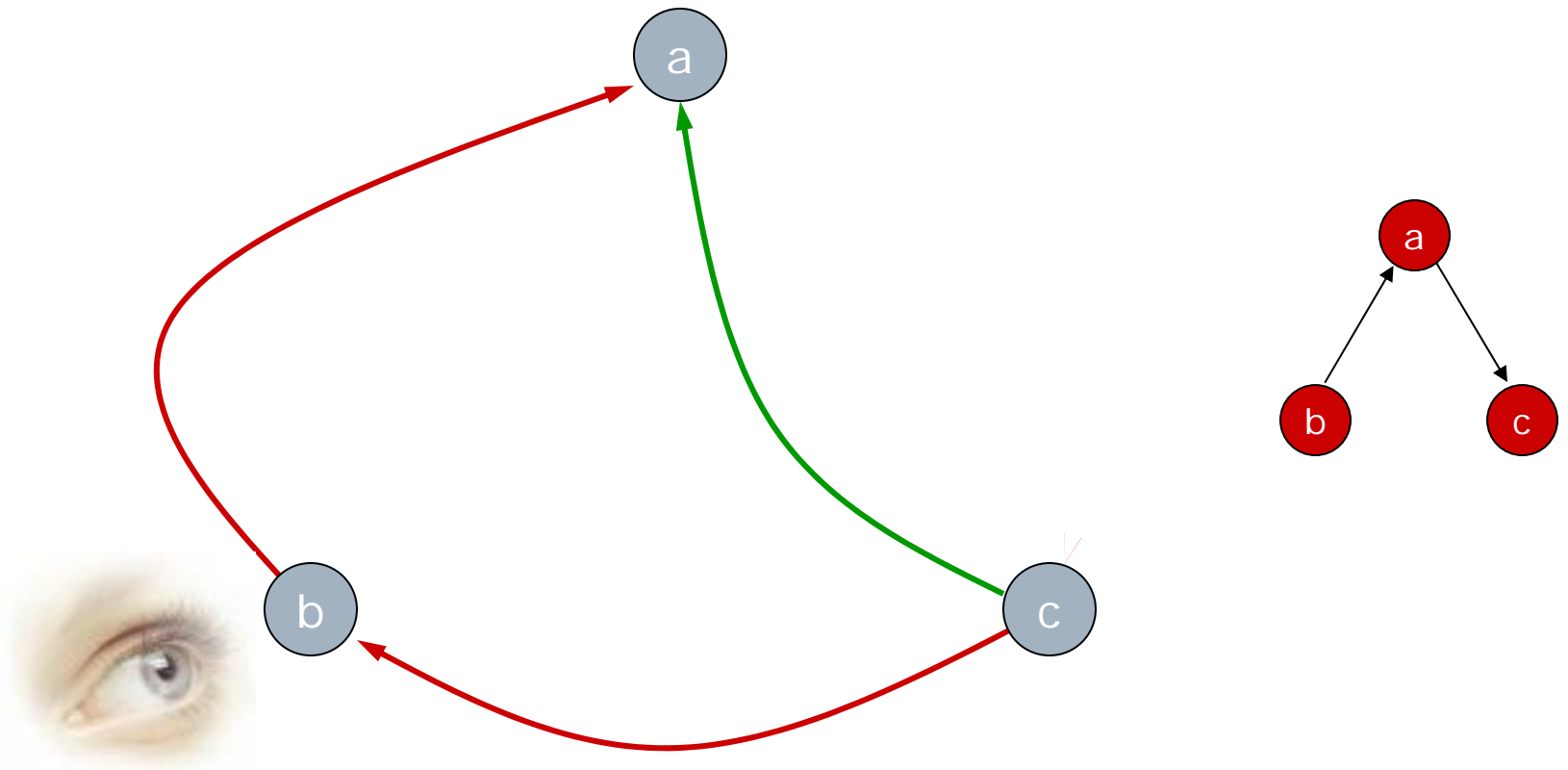
Missing deadlock



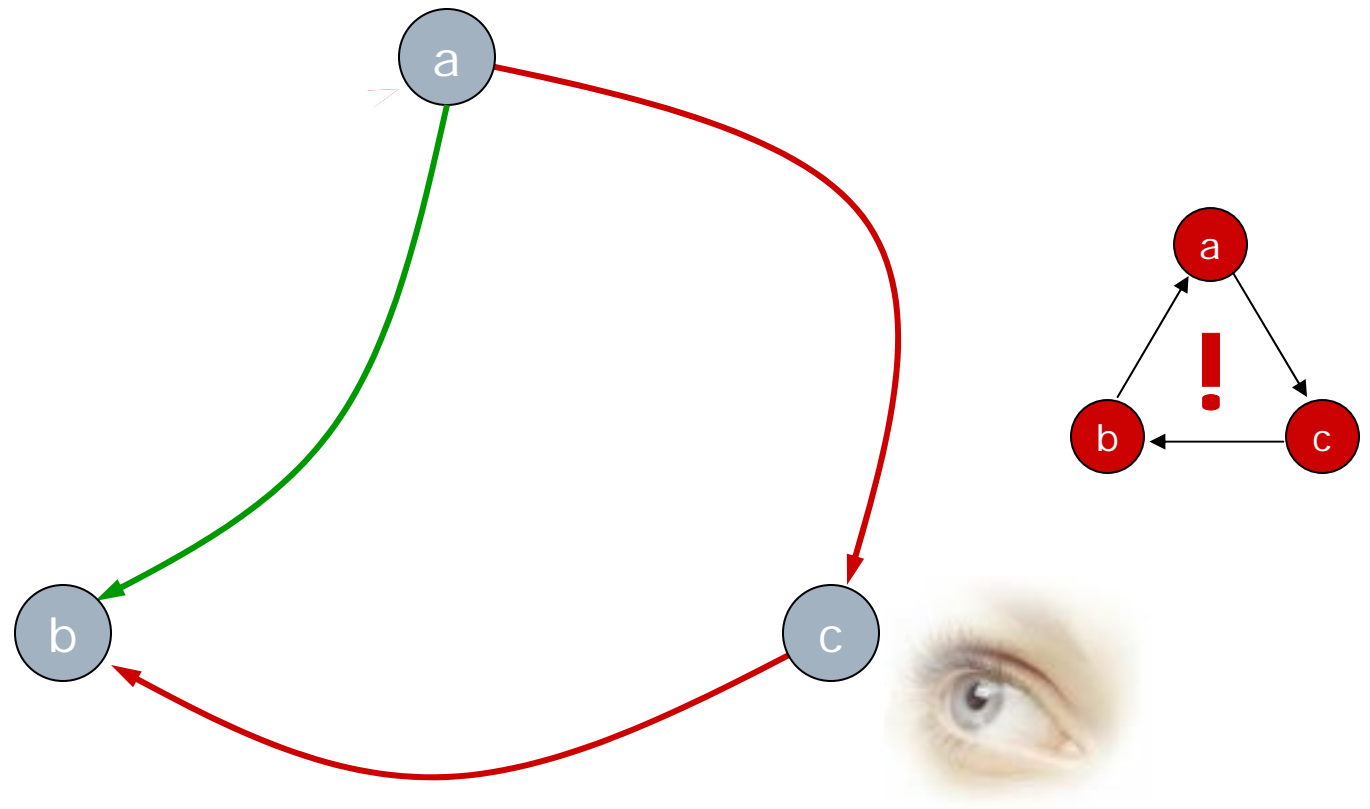
Misdetecting deadlock



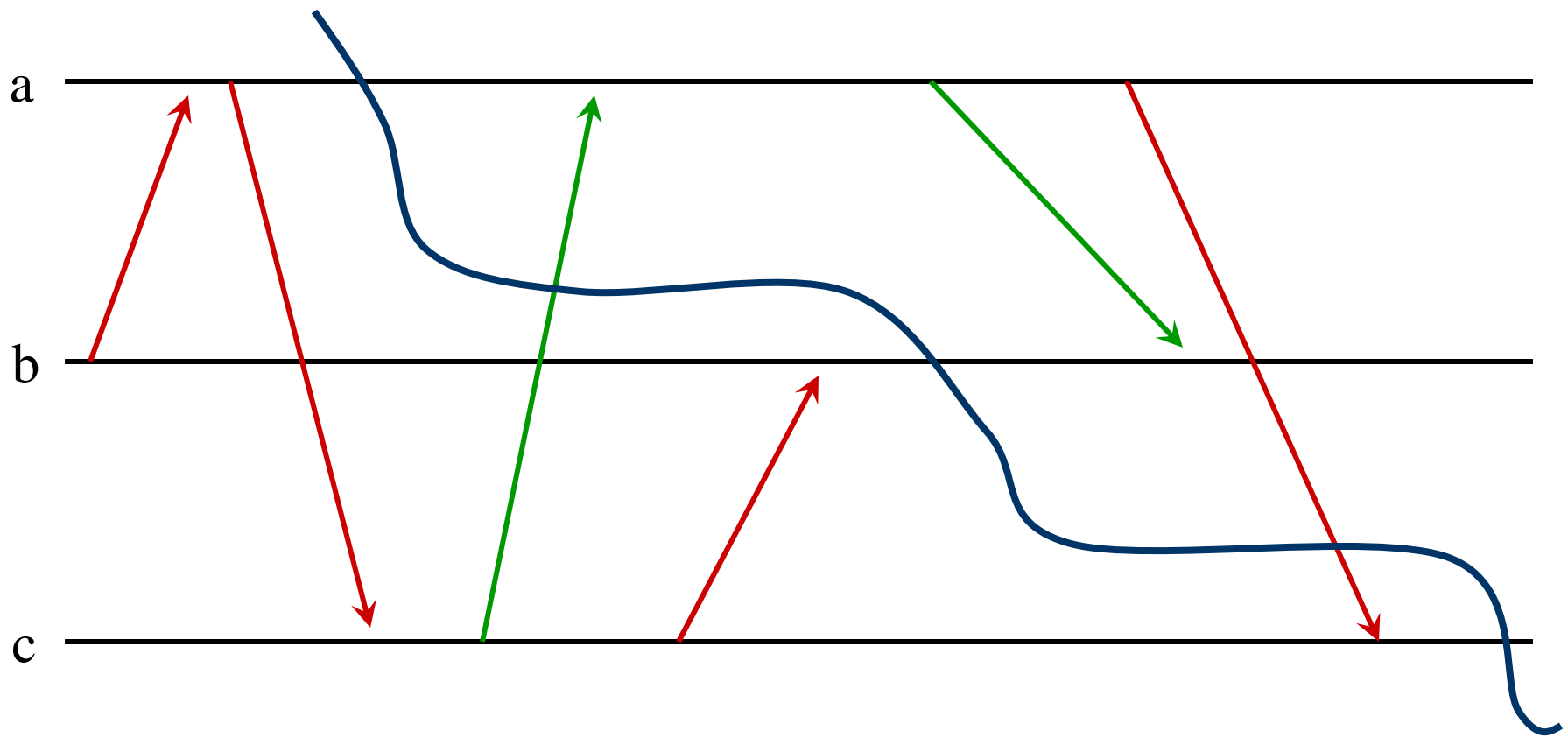
Misdetecting deadlock



Misdetecting deadlock



Misdetecting deadlock



Happens-before relation

A process executes *send* events, *receive* events, and *internal* events.

- e' happens before e ($e' \rightarrow e$) is the transitive closure of
 - A process executed e' and then executed e .
 - e' is a *send* event and e is the corresponding *receive* event.
- e' and e are *concurrent* if neither $e' \rightarrow e$ nor $e \rightarrow e'$.

Happens-before relation (2)

The happens-before relation can also be defined in terms of states.

s' *happens before* s ($s' \rightarrow s$) is the transitive closure of

- A process executed an event that changed its state from s' to s .
- s' is a state immediately before a *send* event and s is the state immediately following the corresponding *receive* event.
- s' and s are *concurrent* if neither $s' \rightarrow s$ nor $s \rightarrow s'$.

Consistent cut

- e' and e are *concurrent* if neither $e' \rightarrow e$ nor $e \rightarrow e'$.
- A *global state* C is a set of sequences $\{s_a, s_b, s_c, \dots\}$, one for each process.
 - The *cut* is the last event in each sequence.
- C is *consistent* if, for all events e in C , all events e' : $e' \rightarrow e$ are in C .
 - A cut is consistent iff all of states immediately following the cut are concurrent.

Snapshot

A *snapshot* is a representation of a global state of a system.

- The local state S_i of each process p_i .
- For each pair p_i, p_j of processes, the state $Q_{i,j}$ and $Q_{j,i}$ of the unidirectional and FIFO channels between p_i and p_j .

Some process p_x will initiate a snapshot, and will wait to receive the snapshot from all processes (including itself).

Snapshot protocol

Stepwise development of *Chandy/Lamport Snapshot protocol*.

Based on development by Colin Fidge

1. Give one that is obviously correct but uses perfectly synchronized clocks and bounded message delivery.
2. Change to an asynchronous protocol by using a property about clocks.
3. Simplify to the actual Snapshot protocol.

Assumes point-to-point FIFO reliable channels, and a connected (but not necessarily fully connected) network.

Step 1: Use clocks

1. A process p_x chooses a time T_s to take a snapshot.
 - T_s must be far enough in the future that p_x can flood the value to everyone.
2. Process p_x floods T_s to everyone.
 - p_x sends to itself.
 - When some process p_i receives T_s for the first time (say from p_j), p_i sends it to all of its neighbors except p_j .

Step 1 (continued)

3. When the clock C_i of p_i reaches T_s it:
 1. Records its local state S_i .
 2. For each neighbor p_j , records the messages $H_{j,i}$ sent by p_j before T_s and not yet received by p_j by T_s .
 - This requires each message m to carry a timestamp $m.T$ which is set by p_j to C_i when it sent m .
 - How do we ensure liveness?
4. Each process p_i sends S_i and its channel states to p_x .

Step 1: Pseudocode

p_x : send(p_x, T_S);

p_i : when (receive(p_x, T_S) for the first time, from p_j)
 for (each neighbor $p_k \neq p_j$) send(p_k, T_S);
 when ($C_i == T_S$) {
 record local state S_i ;
 for (each neighbor p_k) {
 send(p_k, \perp);
 record messages $H_{k,i}$ received from p_k
 sent before T_S ;
 }
 send($p_x, S_i, H_{*,i}$);
 }

Step 1: Proof

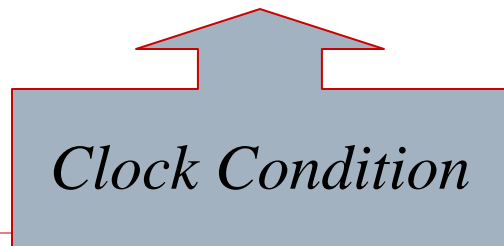
Consider an event e that is in the consistent global state X that the protocol constructs.

Let $T(e)$ be the time that e was executed.

For all events e in X , $T(e) \leq T_s$.

Consider another event e' : $e' \rightarrow e$.

Since $e' \rightarrow e \Rightarrow T(e') < T(e)$, e' is also in X .



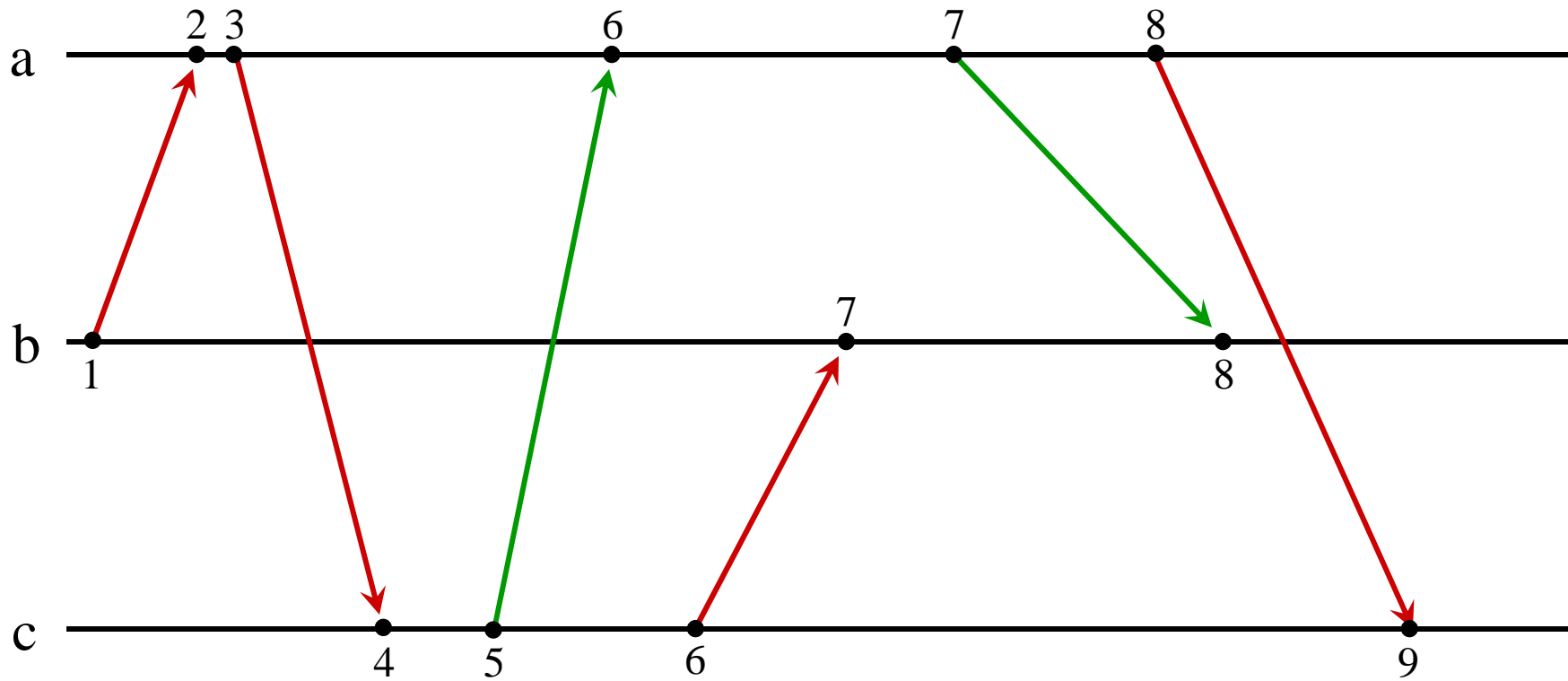
Logical Clocks

A clock that implements $e' \rightarrow e \Rightarrow T(e') < T(e)$ is called a *logical clock*.

A simple logical clock is a *Lamport clock*, which is an integer.

- C_i is initially zero.
- When p_i executes an event e :
 - If e is an internal event, then C_i is increased.
 - If e is a send event of message m , then C_i is increased and piggybacked on the message $m.C$.
 - If e is a receive event of message m , then C_i is set to be larger than both its current value and the value of $m.C$.

Lamport clocks



Step 2

If all we need from time is the clock condition, then we should be able to use the previous protocol with logical clocks rather than real clocks.

Problems:

1. We need a time T_s that is far enough in the future.

Use some integer value ω that is so large that it can't be reached by normal execution.

Step 2 (continued)

2. Lamport clocks don't take on consecutive values.

Instead of a process p waiting for clock to have a value t to execute some action a , have p execute a when its clock is about to take on a value greater than or equal to t (as a result of executing an event e).

At this point, have p execute a before e with a clock equal to t .

Step 2 (continued)

3. How can we ensure liveness?

Having started the flood of ω , p_x can set C_x to ω and then send a message to all of its neighbors.

Since channels are FIFO, each neighbor will need to advance its clock to a value greater than ω and so will start their snapshot.

The message that will do this is \perp .

Step 2: Pseudocode

```
 $p_x$ : send( $p_x, T_s \omega$ );

 $p_i$ : when (receive( $T_s \omega$ ) for the first time, from  $p_j$ )
    for (each neighbor  $p_k \neq p_j$ ) send( $p_k, T_s \omega$ );
when ( $C_i == T_s$  passes through  $\omega$ ) {
    record local state  $S_i$ ;
    for (each neighbor  $p_k$ ) {
        send( $p_k, \perp$ );
        record messages  $H_{j,i}$  received from  $p_k$  sent before  $T_s \omega$ ;
    }
    send( $p_x, S_i, H_{*,i}$ );
}
```

... missing is p_x setting C_x to ω and flooding a message.

Step 2: Pseudocode

p_x : send(p_x, ω);

p_i : when (receive(ω) for the first time, from p_j)

 for (each neighbor $p_k \neq p_j$) send(p_k, ω);

when (C_i passes through ω) {

 record local state S_i ;

 for (each neighbor p_k) {

 send(p_k, \perp);

 record messages $H_{j,i}$ received from p_k sent before ω ;

 }

 send($p_x, S_i, H_{*,i}$);

 }

... missing is p_x setting C_x to ω and flooding a message.

Step 3

p_x : send(p_x, ω);
 $C_x = \omega$
 send(p_x, ∇);

p_i : when (receive(ω) for the first time, from p_j) for (each neighbor $p_k \neq p_j$) send(p_k, ω);
 when (C_i passes through ω) {
 record local state S_i ;
 for (each neighbor p_k) {
 send(p_k, \perp);
 record messages $H_{j,i}$ received from p_k sent before ω .
 }
 send($p_x, S_i, H_{*,i}$);
 }
 when (receive(∇) for the first time, from p_j) for (each neighbor $p_k \neq p_j$) send(p_k, ∇);

... note interplay of ω , ∇ and \perp messages.

Step 3: Pseudocode (Chandy/Lamport)

p_x : send(p_x , *take ss*);

p_i : when (receive(T_s *take ss*) for the first time, from p_j)
if ($p_j \neq p_i$) $H_{j,i} = \emptyset$
for (each neighbor $p_k \neq p_j$) {
 send(p_k , *take ss*);
 record local state S_i ;
 record messages $H_{j,i}$ received from p_k
 until receive *take ss*.
 }
send(p_x , S_i , $H_{*,i}$);
}

Detecting RPC deadlock

Define p waits-for* q if p has executed $\text{RPCsend}(q, m)$, q has received this message, and q has not yet executed $\text{RPCreply}(r)$.

- $(\text{deadlock}^* \Rightarrow \text{deadlock})$ and $(\text{deadlock} \Rightarrow \diamond \text{deadlock}^*)$.

Detecting RPC deadlock (continued)

- Periodically have some process p_x start a snapshot, where S_i is the process (if any) from which p_i has received an RPCsend message and to which p_i has not yet executed RPCreply.
- Process p_x uses these states to construct a waits-for* graph. If it contains a cycle, then the system is RPCdeadlocked*.