Today

- Some topics and notation on specifying concurrent programs.
- Snapshots: detecting stable properties
  - Causality
  - Consistent and inconsistent cuts
  - Lamport clocks
  - Chandy/Lamport snapshot protocol
  - … a little bit about distributed deadlock detection

A Little Bit of Concurrency

- Distributed systems are examples of concurrent systems.
- How does one describe what a concurrent system should do?
  - For non-concurrent systems, we often use input-output relations, eg $\{ P^{x}_{x+1} \}$ x := x + 1 $\{ P \}$
  - Concurrent systems often don’t terminate…

Behaviors

- We need a way to describe behaviors: sequences of interdigitated process histories.
  - We can do this either by giving the events the processes execute, or by giving the states the process goes through (*event based* vs. *state based*).
  - Since in either case we need to include the initial state, we'll use the simpler state based approach for purposes of specification.
    - Other times, we will use event sequences when more convenient…

Example concurrent program

```
{ $x = 0$ } cobegin    while (true) $x := x + 1$
                  ||    while (true) $x := x - 1$
            coend
```

0; 1; 0; 1; 0; 1; 0; 1; 0; 1; 0; 1; 0; 1; …
0; 1; 2; 3; 4; 5; 6; 7; 8; 9; 10; 11; 12; …
0; 1; 2; 1; 2; 3; 2; 1; 0; −1; −2; −3 ; −2 ; −1 ; 0; …
…

Some things about the program (maybe)

- Initially $x = 0$
- Always $|\, x - x' \,| = 1$ (where x' is x in the next state)
- Eventually $x - x' = 1$
- Infinitely often $x - x' = 1$
- It is possible for x to eventually be 100

Safety and Liveness

- *Properties* are *predicates evaluated on behaviors*.
- *Safety*: nothing bad happens (each state is good).
- *Liveness*: something good happens (the good state is eventually reached.

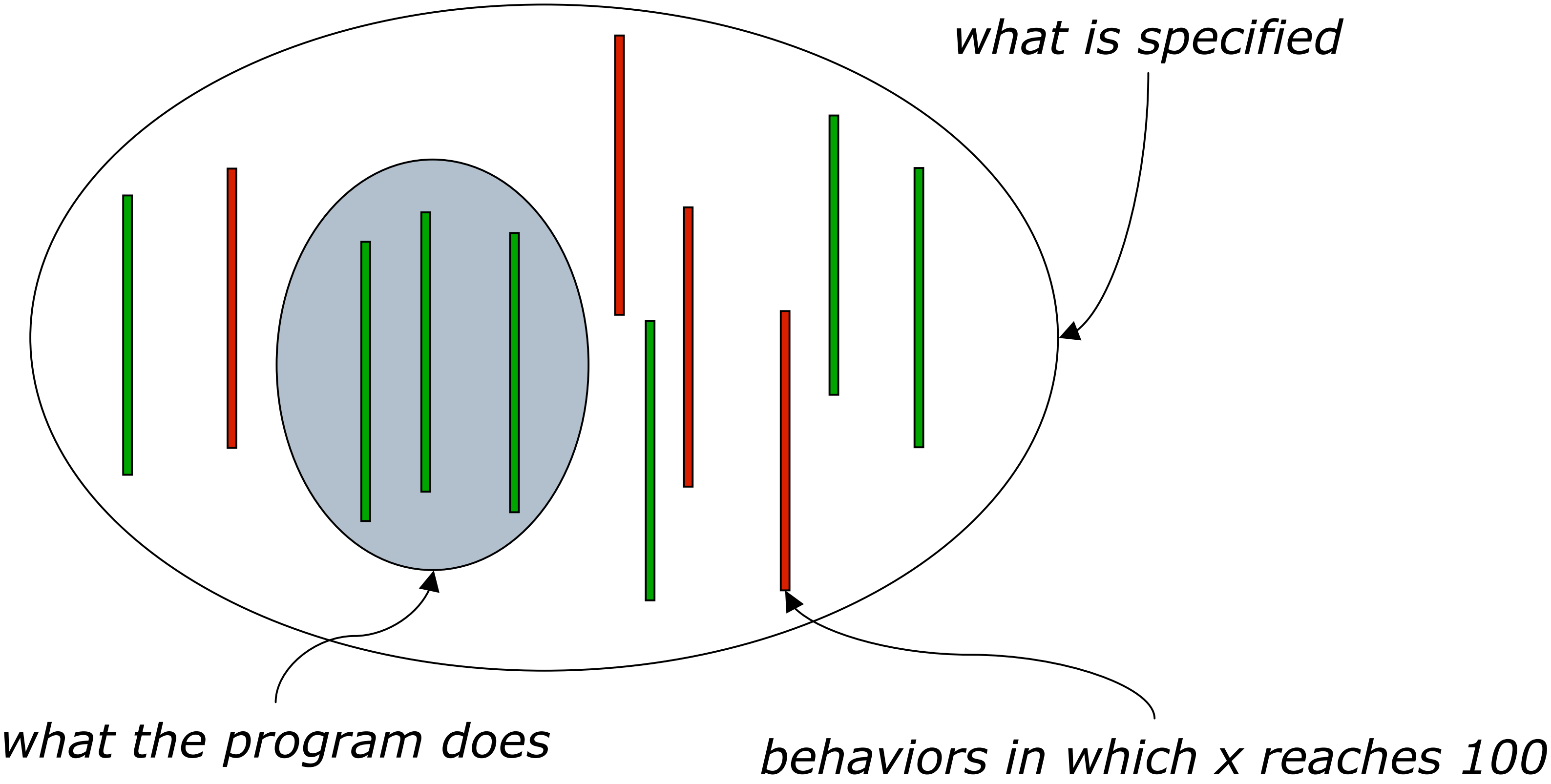
- Initially $x = 0$
- Always $| x - x' | = 1$
- Eventually $x - x' = 1$
- Infinitely often $x - x' = 1$
- It is possible for x to eventually be 100

Safety and Liveness

- *Properties* are *predicates evaluated on behaviors*.
- *Safety*: nothing bad happens (each state is good).
- *Liveness*: something good happens (the good state is eventually reached.

- Initially $x = 0$
- Always $| x - x' | = 1$
- Eventually $x - x' = 1$
- Infinitely often $x - x' = 1$
- It is possible for x to eventually be 100

Implementation as Implication



A little bit of notation

Linear time temporal logic is a way to write properties.

P: P holds in the current (initial) state.

$\Box P$: P always holds.

$\Diamond P$: P eventually holds.

$\neg \Diamond P \equiv \Box \neg P$

$\neg \Box P \equiv \Diamond \neg P$

$\Box \Box P \equiv \Box P$

$\Diamond \Diamond P \equiv \Diamond P$

$\Diamond \Box P$: Eventually, P always holds.

$\Box \Diamond P$: P holds infinitely often. $\Diamond \Box P \Rightarrow \Box \Diamond P$

$\Box \Diamond \Box P \equiv \Diamond \Box P$

$\Diamond \Box \Diamond P \equiv \Box \Diamond P$

Examples

$x = 0$

$\Box\, | x - x' | = 1$

$\Diamond\, x - x' = 1$

$\Box\Diamond\, x - x' = 1$

State of a distributed system

- The preceeding was based on the idea of a interleaved model. But, with true concurrency, such an interleaving is clearly a fiction.
- What does it mean for a distributed system to have a particular global state?
- Look at this problem in the context of an important class of states: those that arise from *stable properties*:

$$P \text{ stable: } P \Rightarrow \Box P$$

RPC Deadlock

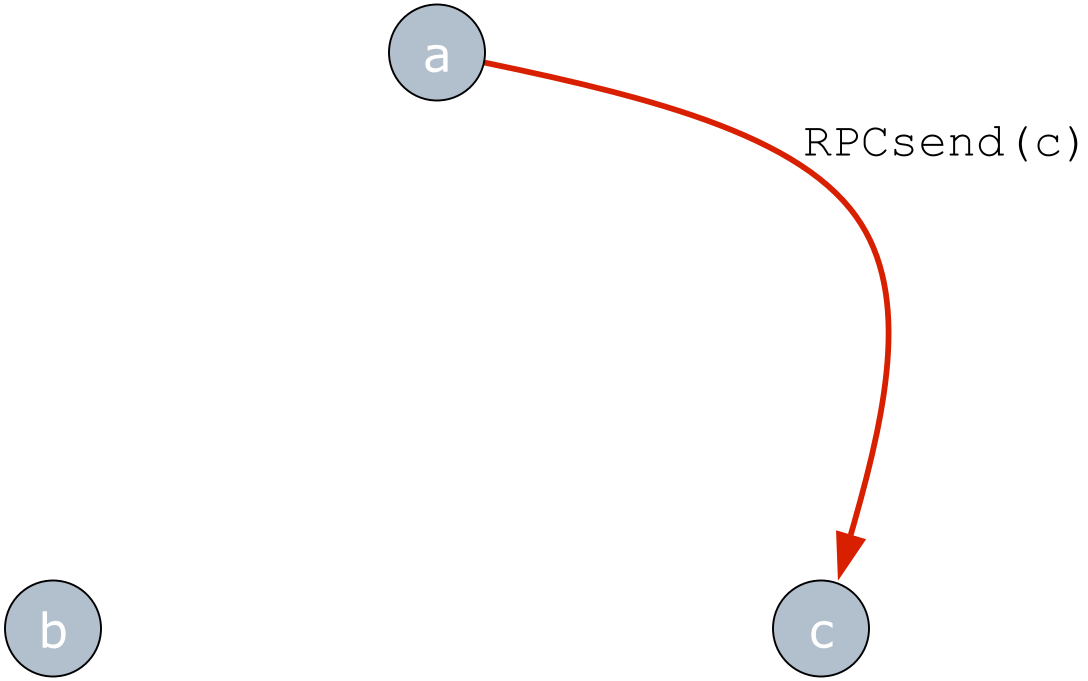
Design a protocol to detect stable property *RPC Deadlock*

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

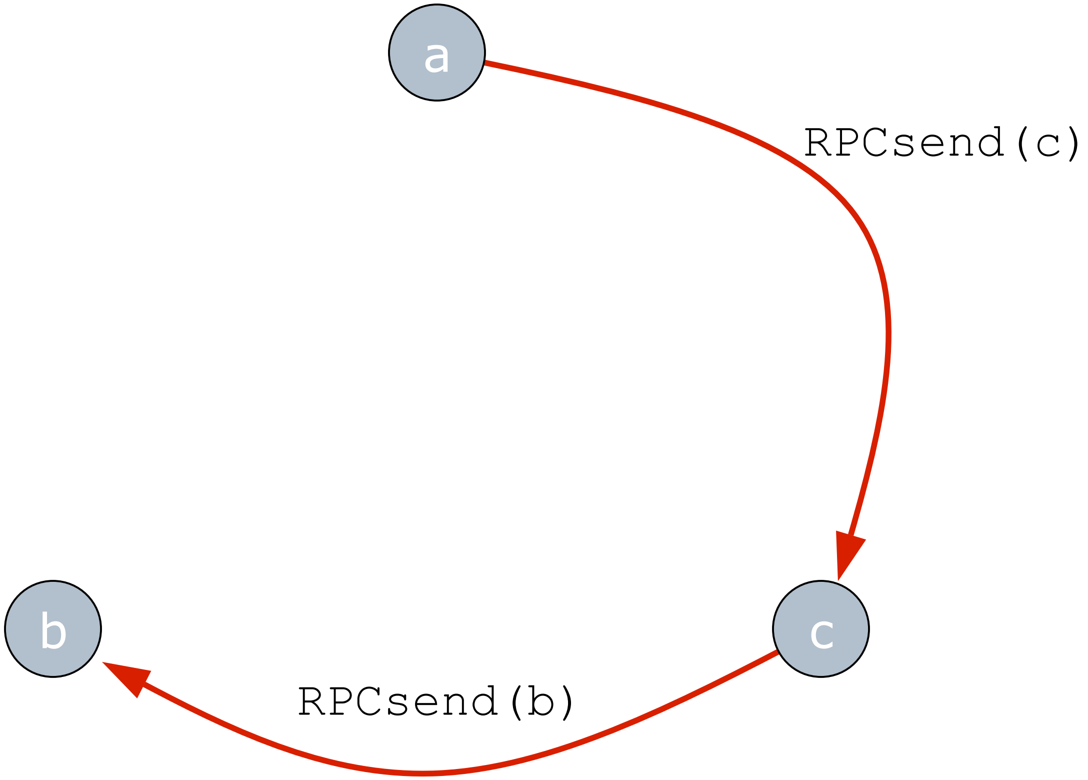
```
        r = RPCsend(p, m);
(m, p) = RPCreceive();
        RPCreply(r);
```

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

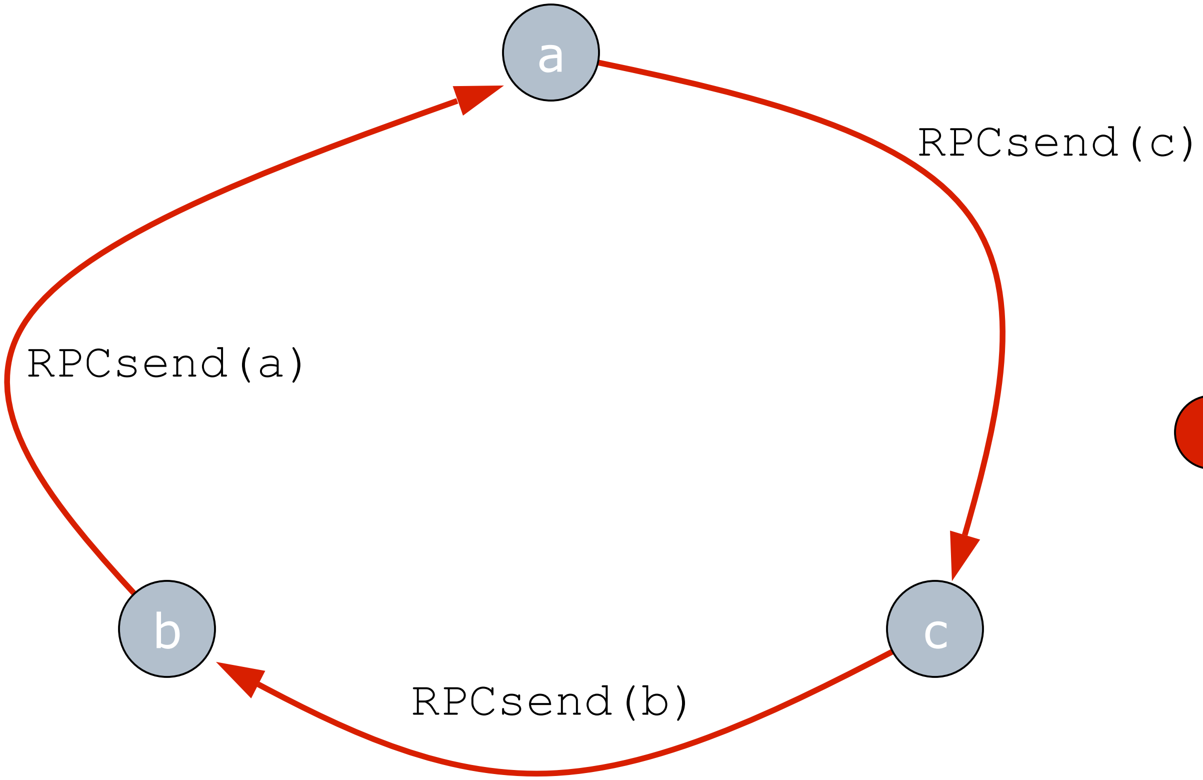
Example of deadlock

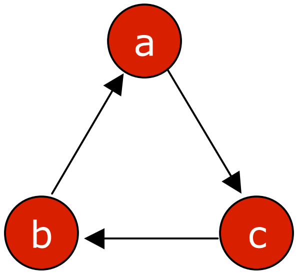


Example of deadlock



Example of deadlock





A simple protocol

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

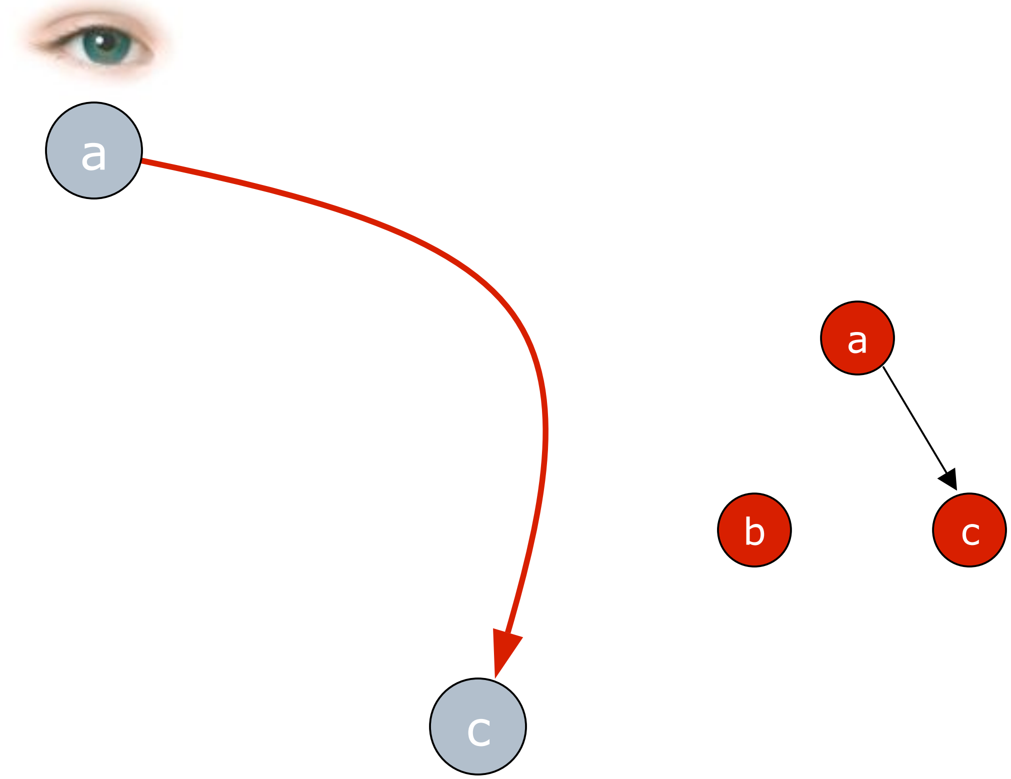
- Who *p* is waiting on;
- Which `RPCsend` requests have been received (even if not `RPCreceive`'d).

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

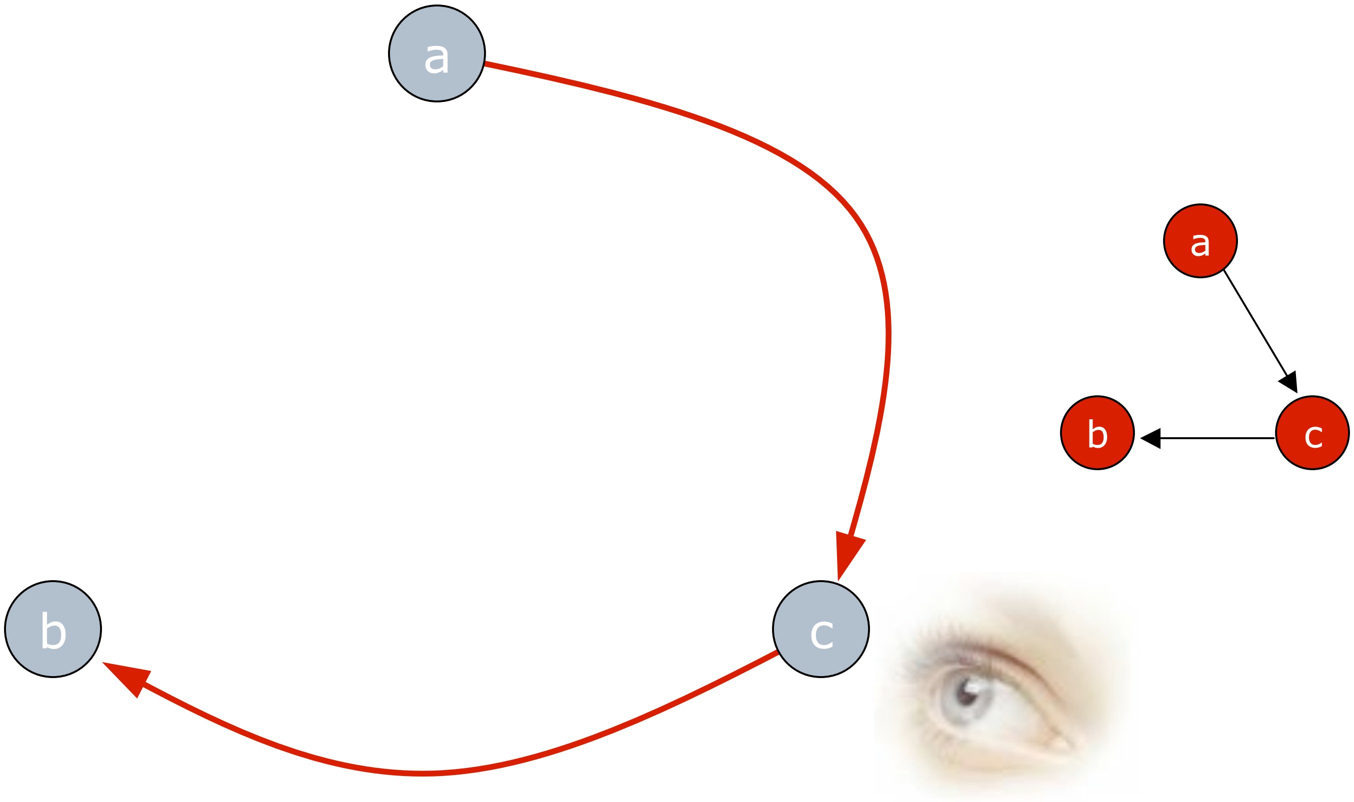
A simple protocol (2)

```
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;
```

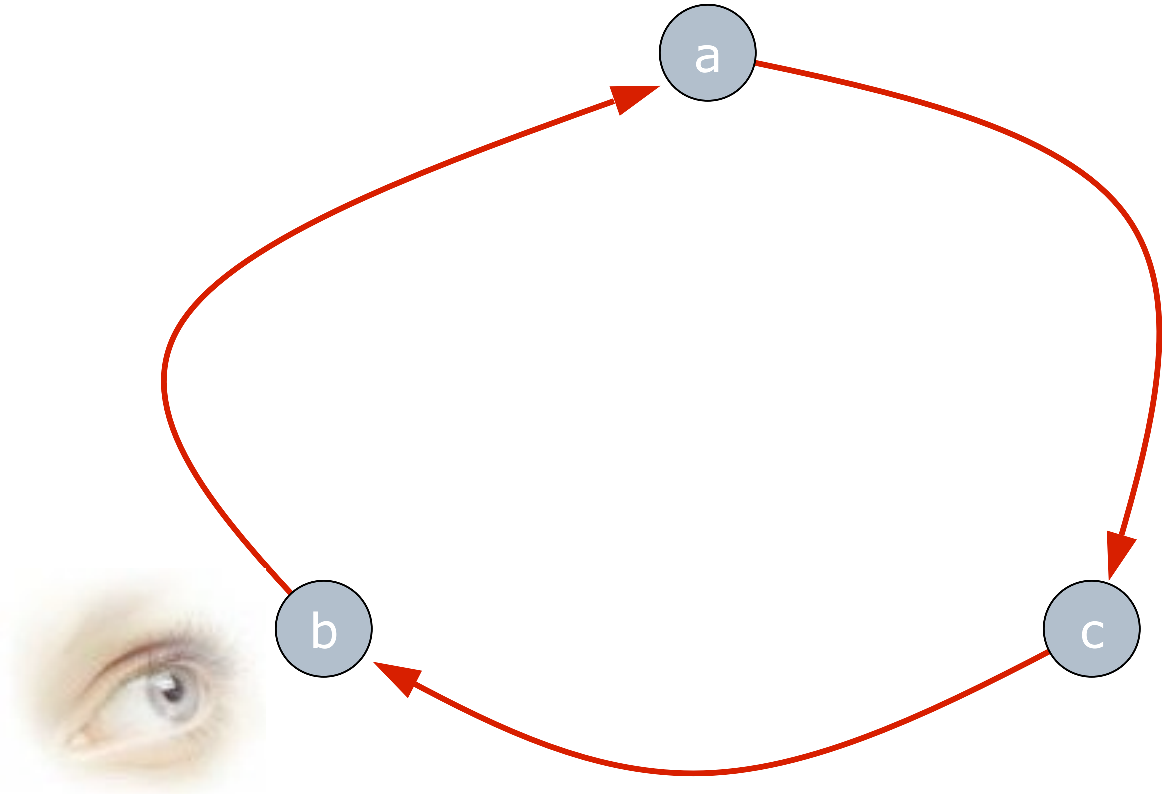
Detecting deadlock

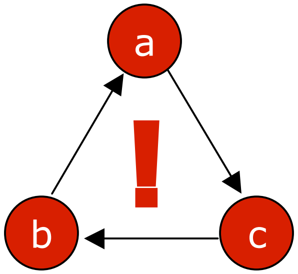


Detecting deadlock

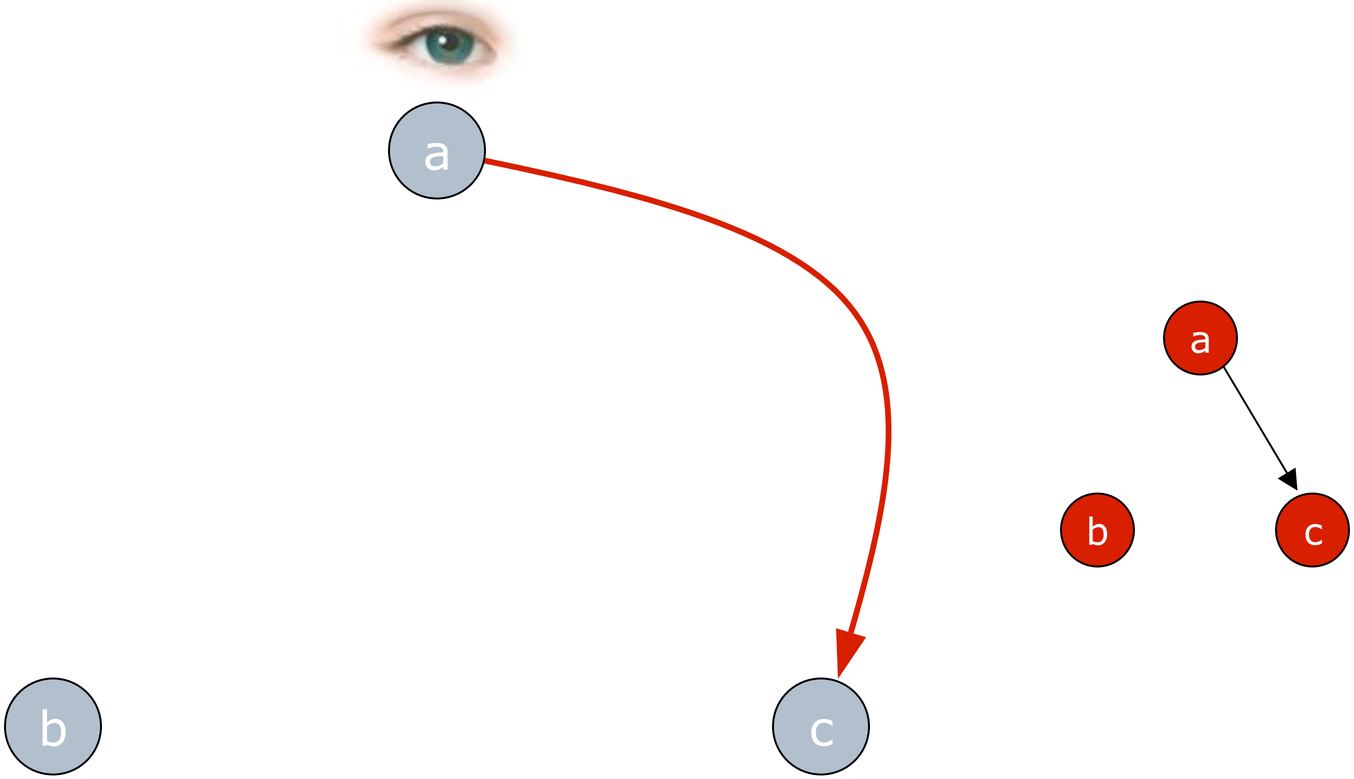


Detecting deadlock

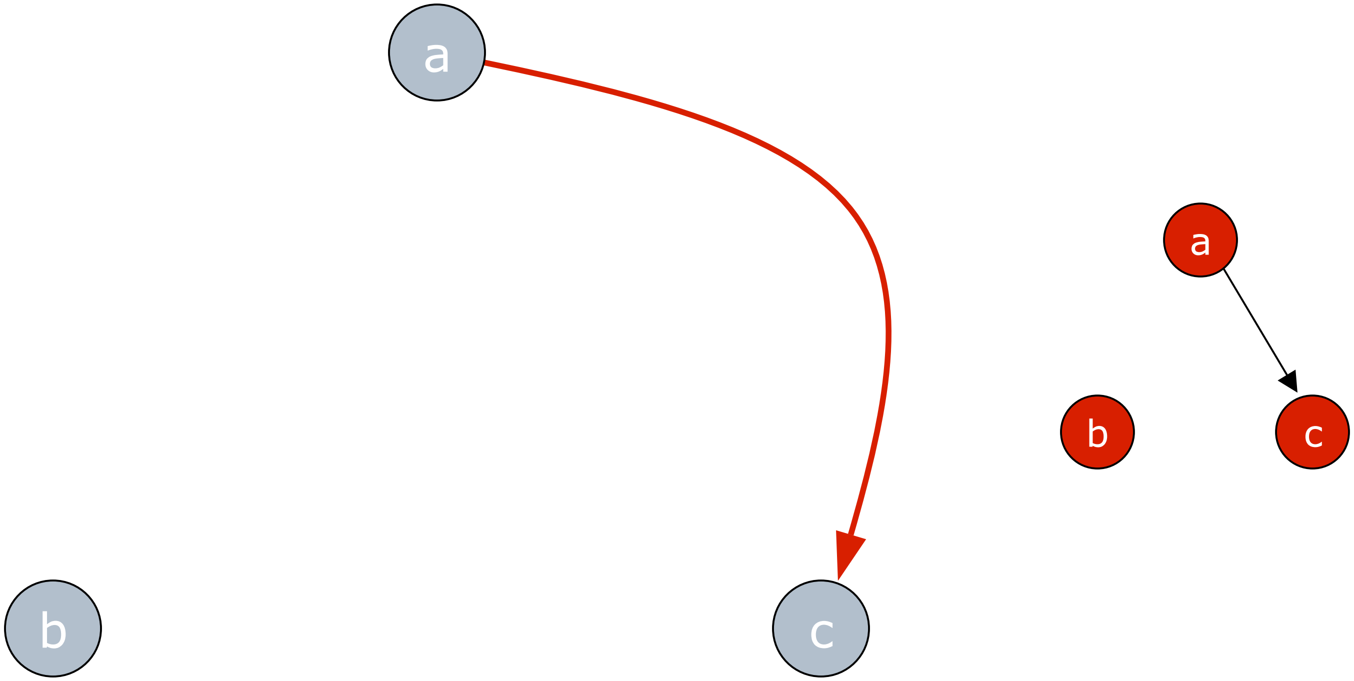




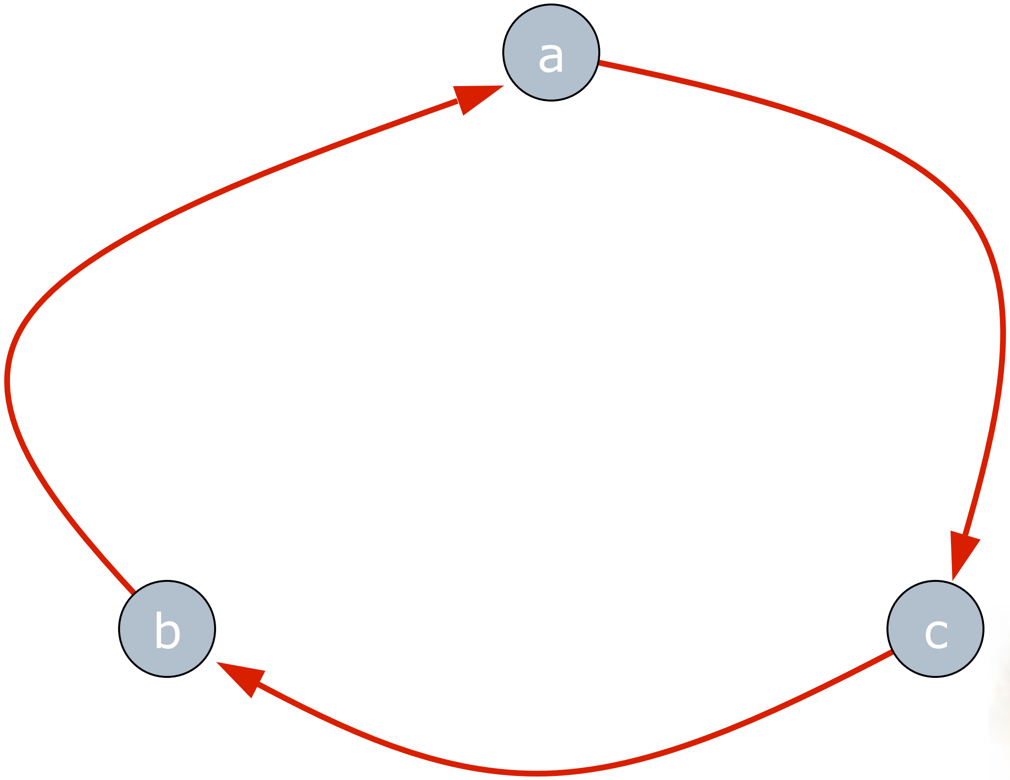
Missing deadlock

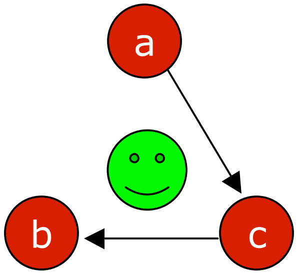


Missing deadlock

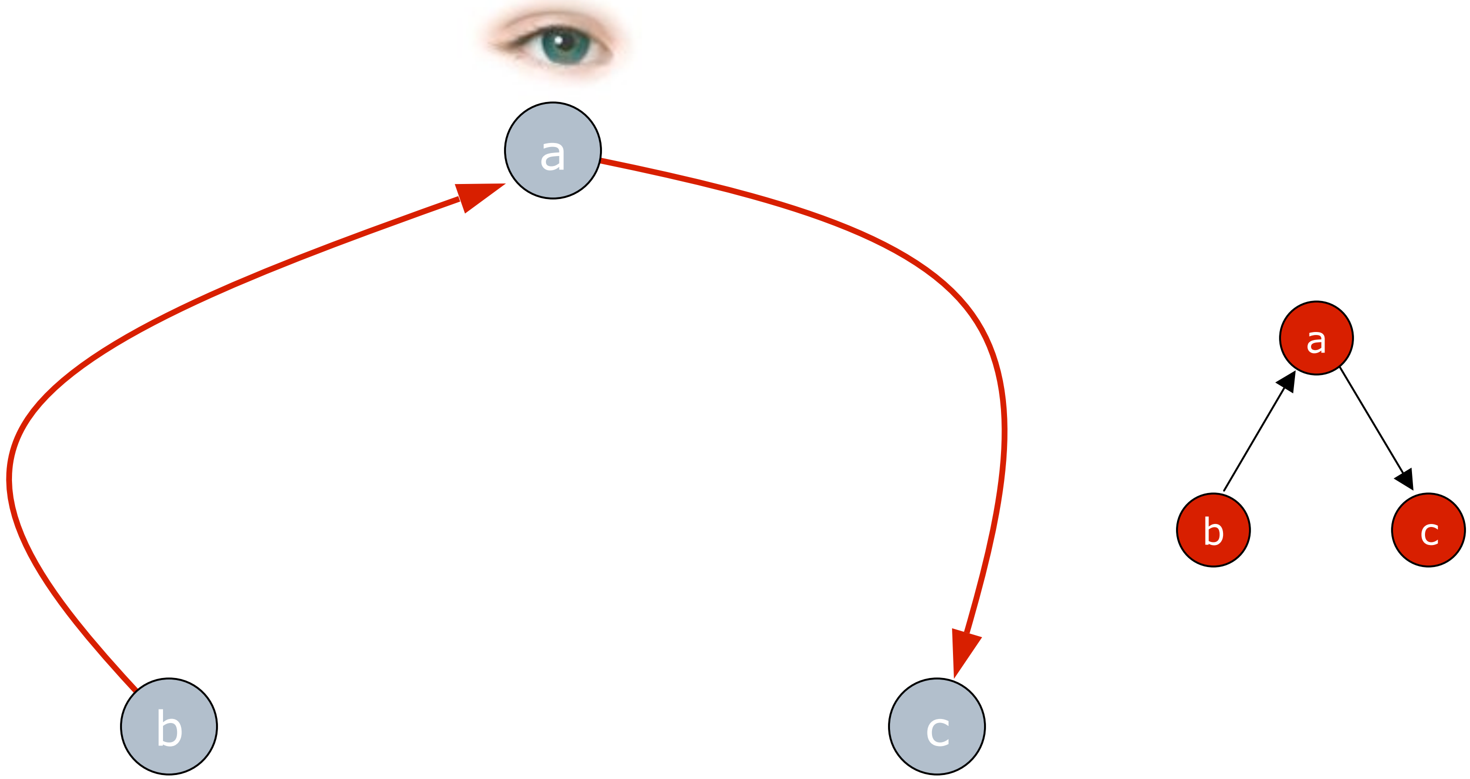


Missing deadlock

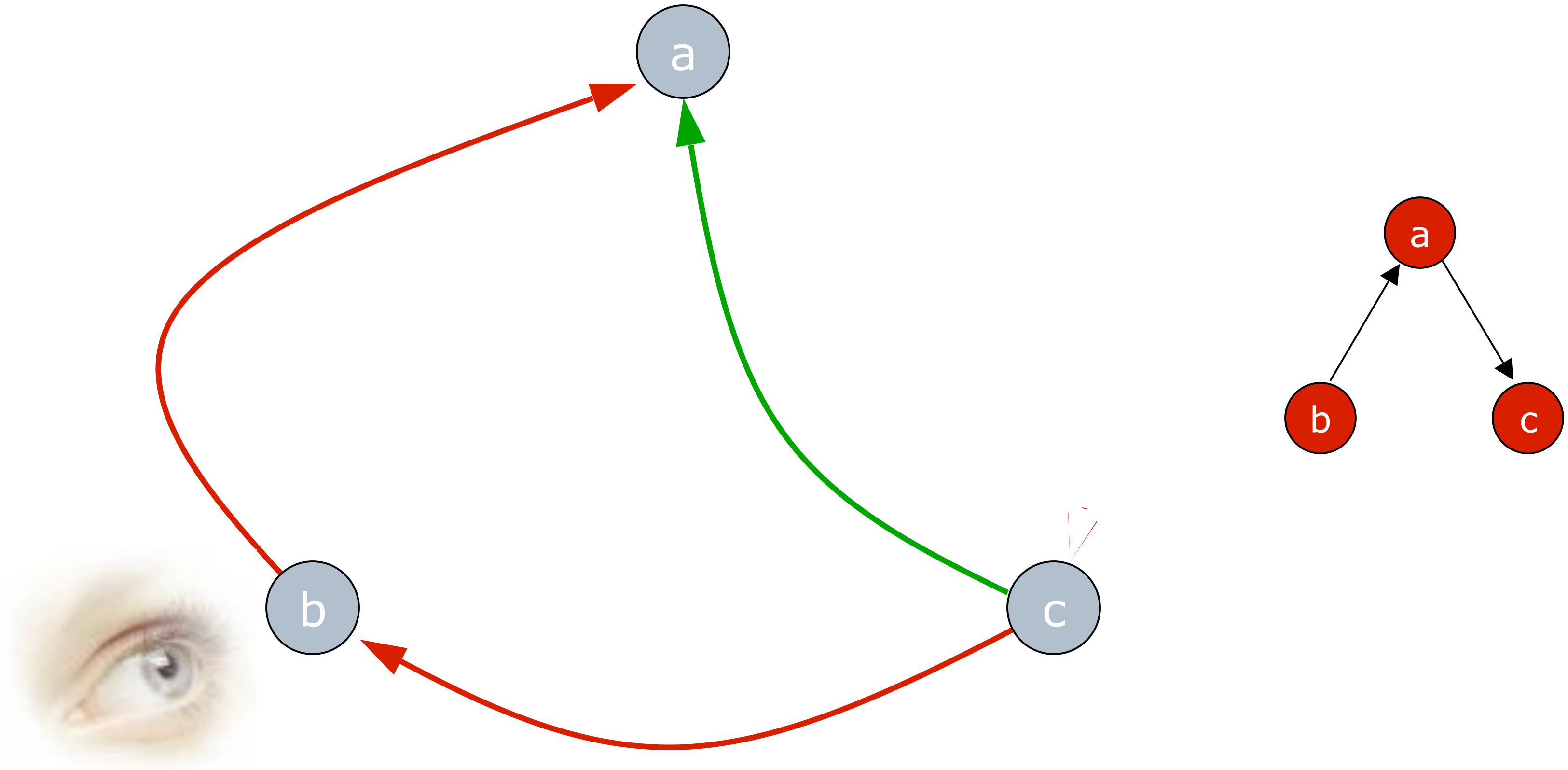




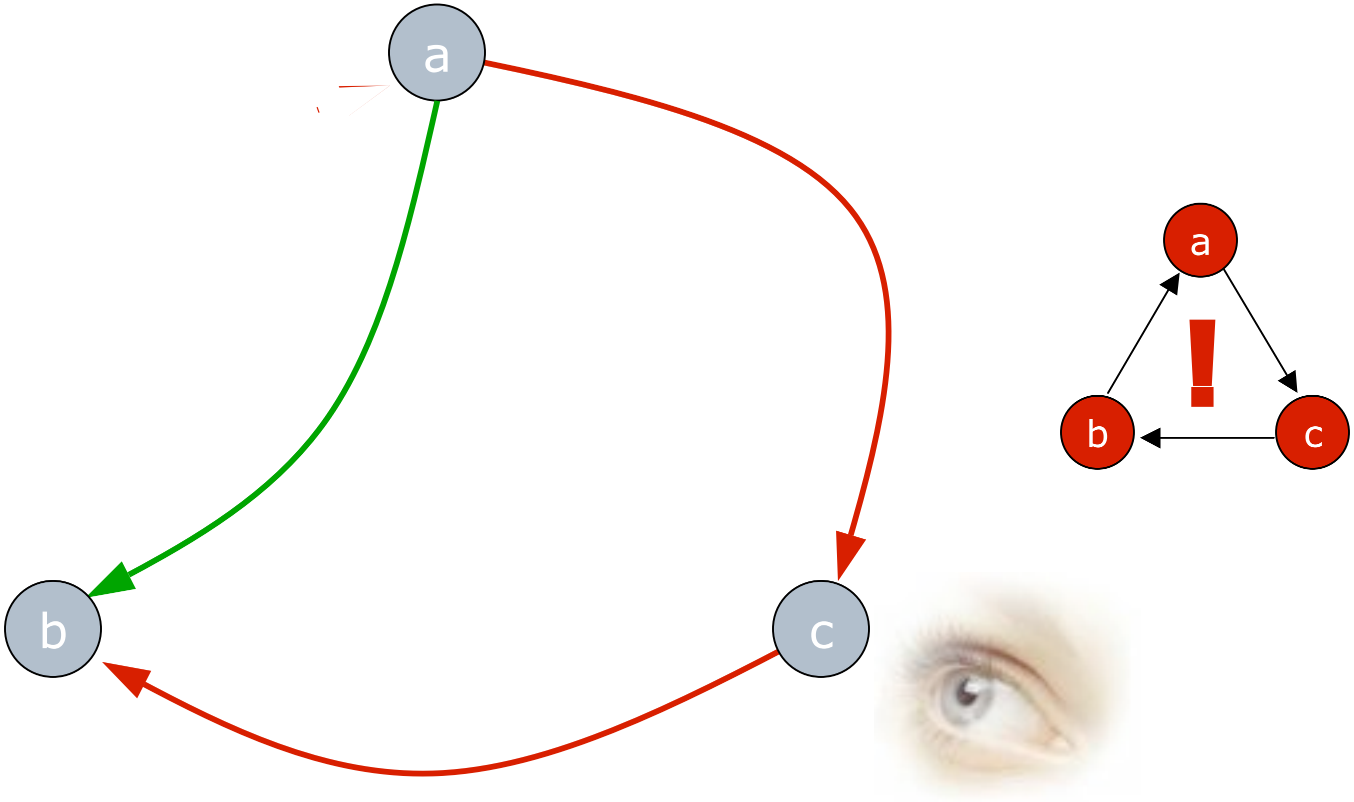
Misdetecting deadlock



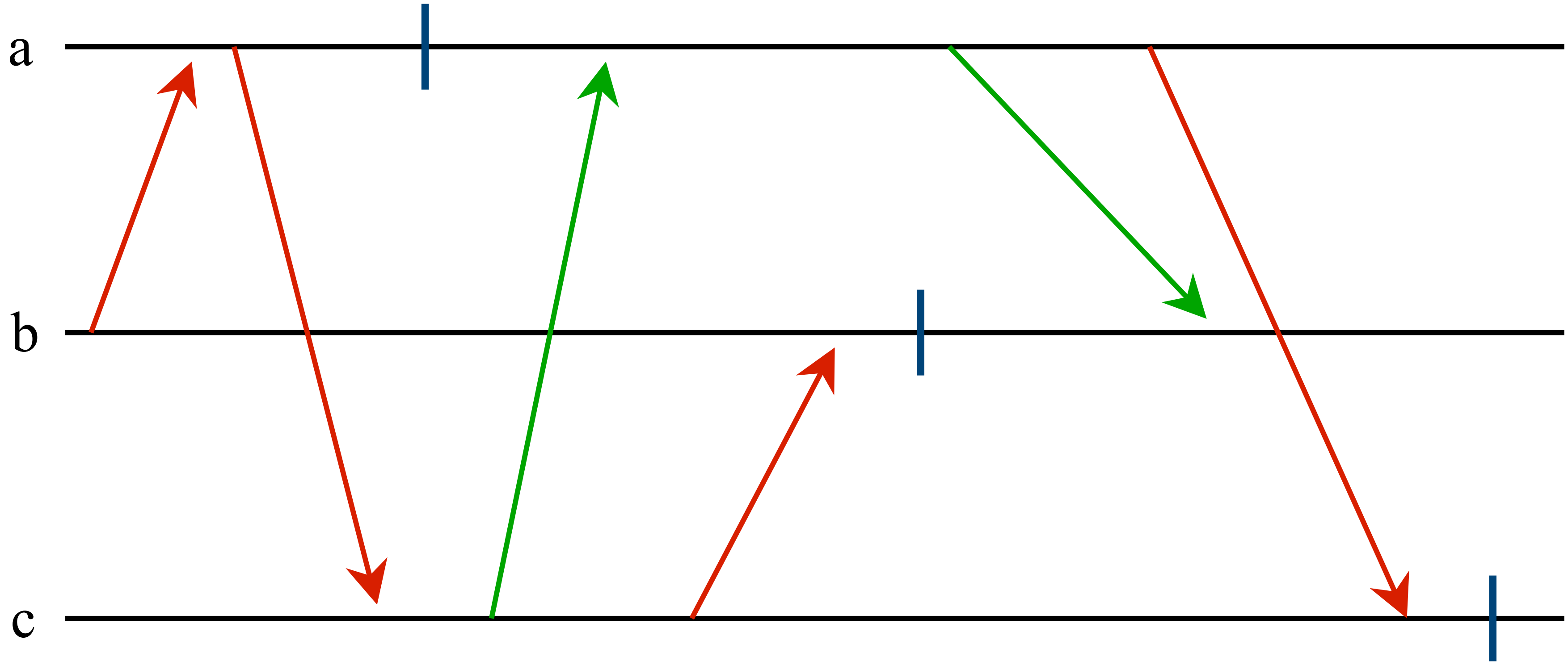
Misdetecting deadlock



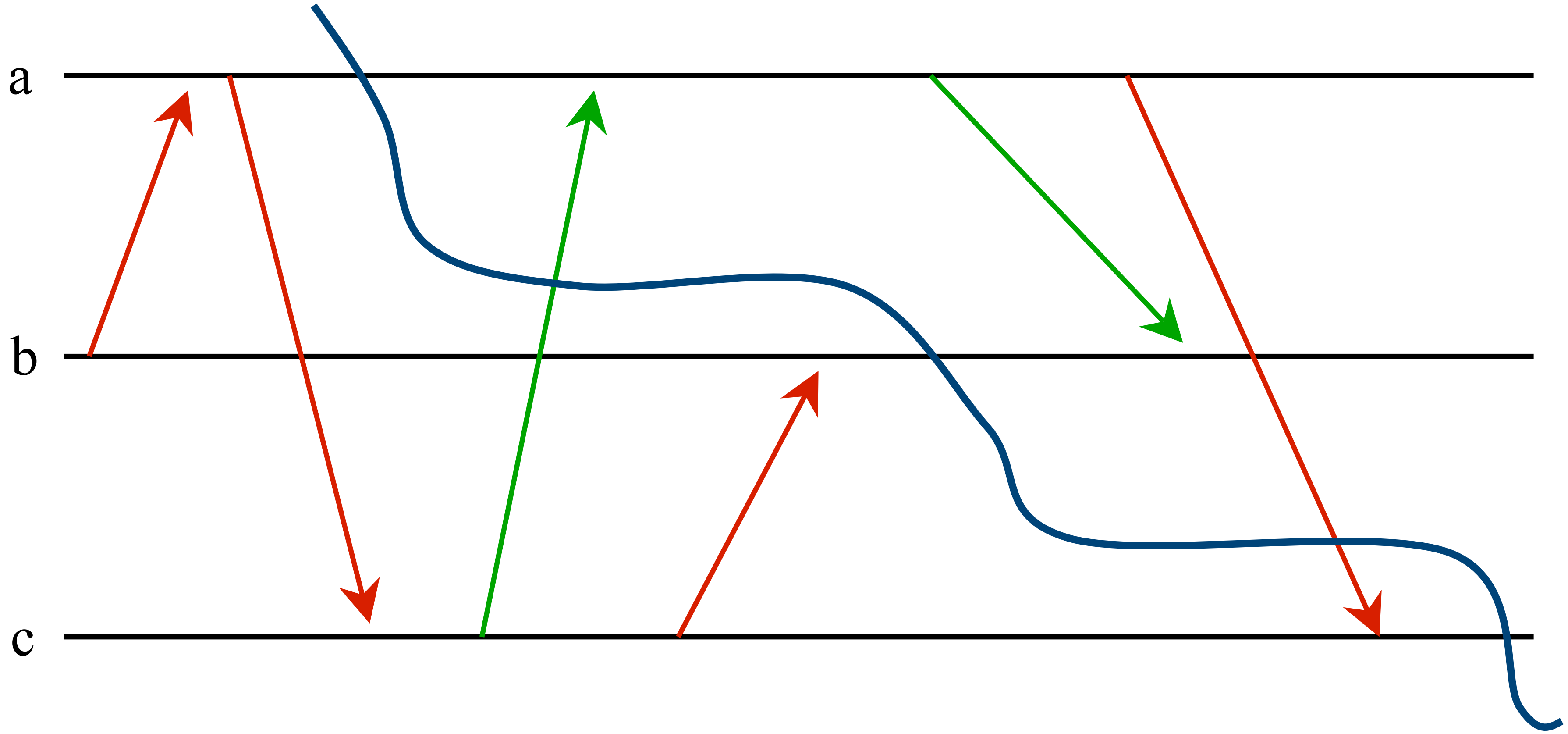
Misdetecting deadlock



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_1 and e_2 are *concurrent* if neither $e_1 \rightarrow e_2$ nor $e_2 \rightarrow e_1$.

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_1 and S_2 are *concurrent* if neither $S_1 \rightarrow S_2$ nor $S_2 \rightarrow S_1$.

Consistent cut

- e' and e are *concurrent* if neither $e' \rightarrow e$ nor $e \rightarrow e'$.
- A *global state* C is a set of event sequences $\{s_a, s_b, s_c, \ldots\}$, 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 (who already knows it!)

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 $Q_{j.i}$ 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 channel states to p_x.

Step 1: Pseudocode

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

$p_i$: when (receive($T_s$) 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 $Q_{k,i}$ received from $p_k$
             sent before $T_{S;}$
           }
         send($p_x$, $S_i$, $Q_{*,i}$);
         }
```

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.

Clock Condition

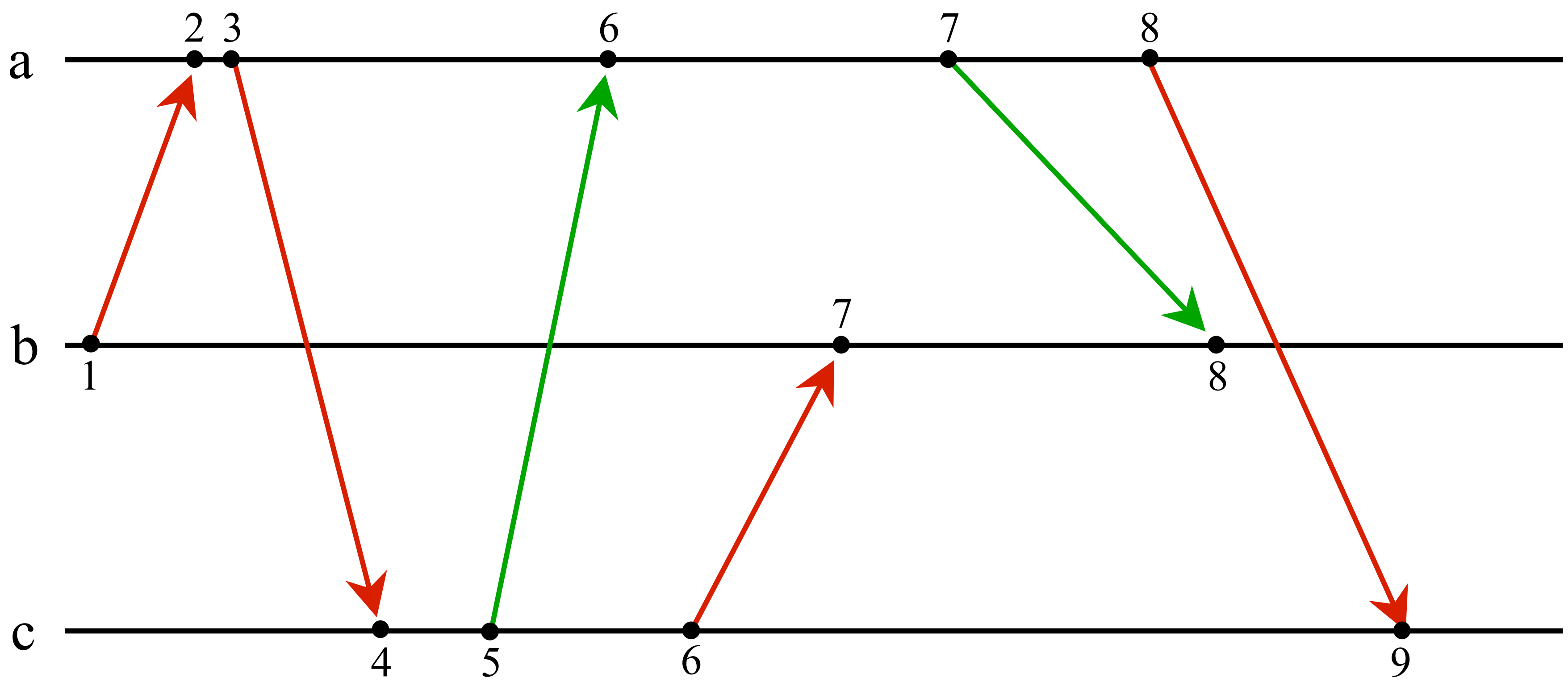
Logical Clocks

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 ⊥.

Step 2: Pseudocode

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

$p_i$: 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 $Q_{j,i}$ received from $p_k$
          sent before ~~$T_S$~~ $\omega$;
      }
      send($p_x$, $S_i$, $Q_{*,i}$);
      }
```

Step 2: Pseudocode

```
$p_x$: send($p_x$, $\omega$);
    $C_i$ = $\omega$;

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

Step 3

```
$p_x$: for (each neighbor $p_j$) send($p_i$, $\omega$);
    $C_i$ = $\omega$;
```

This is a local action and can be combined into one.

Step 3 (continued)

```
$p_i$:  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($p_k$, ⊥);
                    record messages $Q_{j,i}$ received from $p_k$
                            sent before ω;
                    }
            send($p_x$, $S_i$, $Q_{*,i}$);
            }
```

The two floods (of ω and of ⊥) can be combined into one (of "Take SS"). Need to have p_x send "Take SS" to itself as well.

Step 3: Pseudocode (Chandy/Lamport)

```
$p_x$: send($p_x$, "Take ss");

$p_i$: when (receive("Take ss") for the first time,
        from $p_j$)
      record local state $S_i$;
      for (each neighbor $p_k$) {
        send($p_k$, "Take ss");
        if ($p_k \neq p_j$)
          record messages $Q_{k,i}$ received from $p_k$
          until receive($p_k$, "Take ss");
        else $Q_{j,i} = \emptyset$
        }
      send($p_x$, $S_i$, $Q_{*,i}$);
      }
```

Detecting RPC deadlock

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

- (deadlock* ⇒ deadlock) and (deadlock ⇒ ◊deadlock*).

Detecting RPC deadlock (continued)

- Periodically have some process p_x start a snapshot, where the reported state 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 constructs a waits-for* graph. If it contains a cycle, then the system is RPCdeadlocked*.