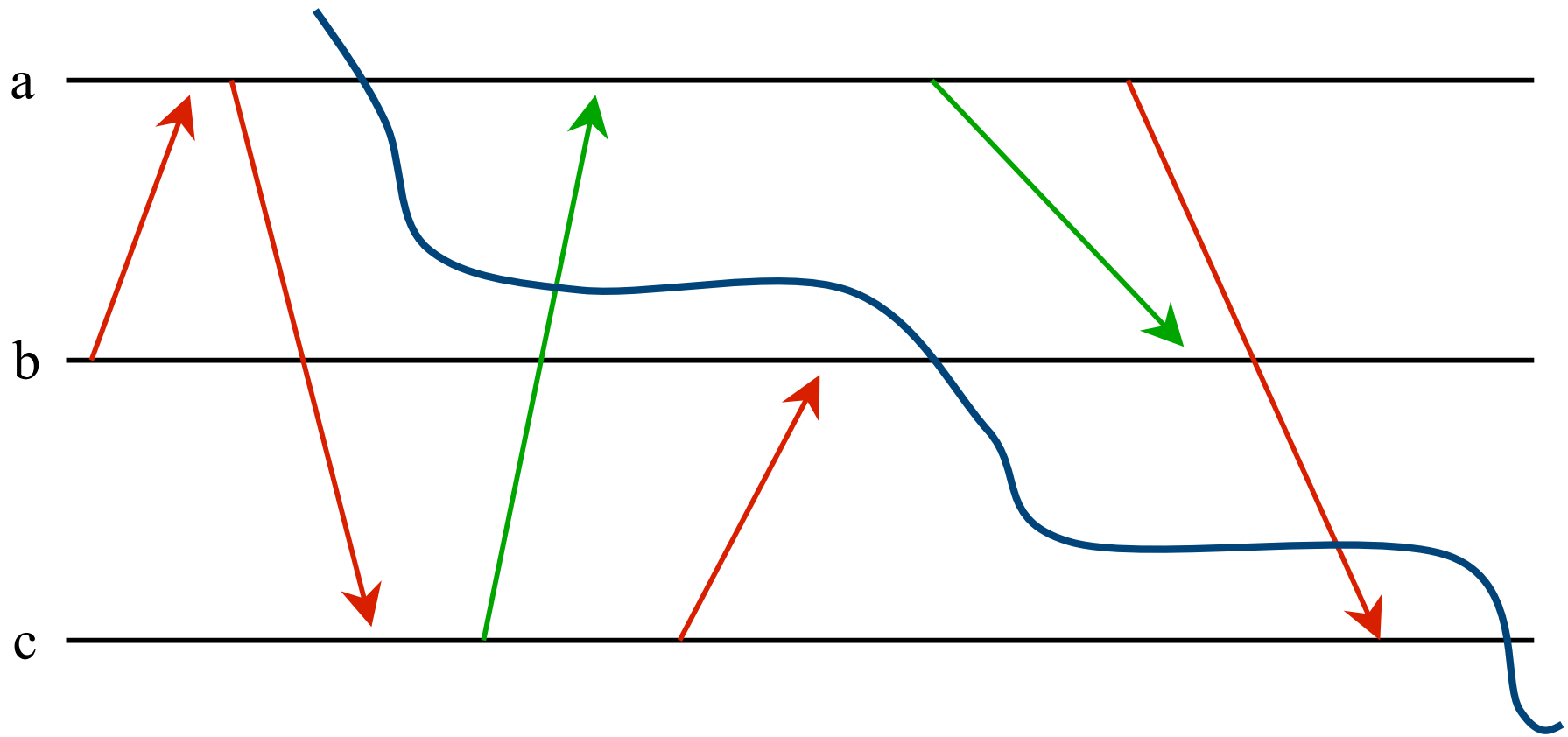
Today

- Completing the snapshot protocol
  - A few group proofs
  - Lamport clocks
  - Chandy/Lamport snapshot protocol
  - … a little bit about distributed deadlock detection

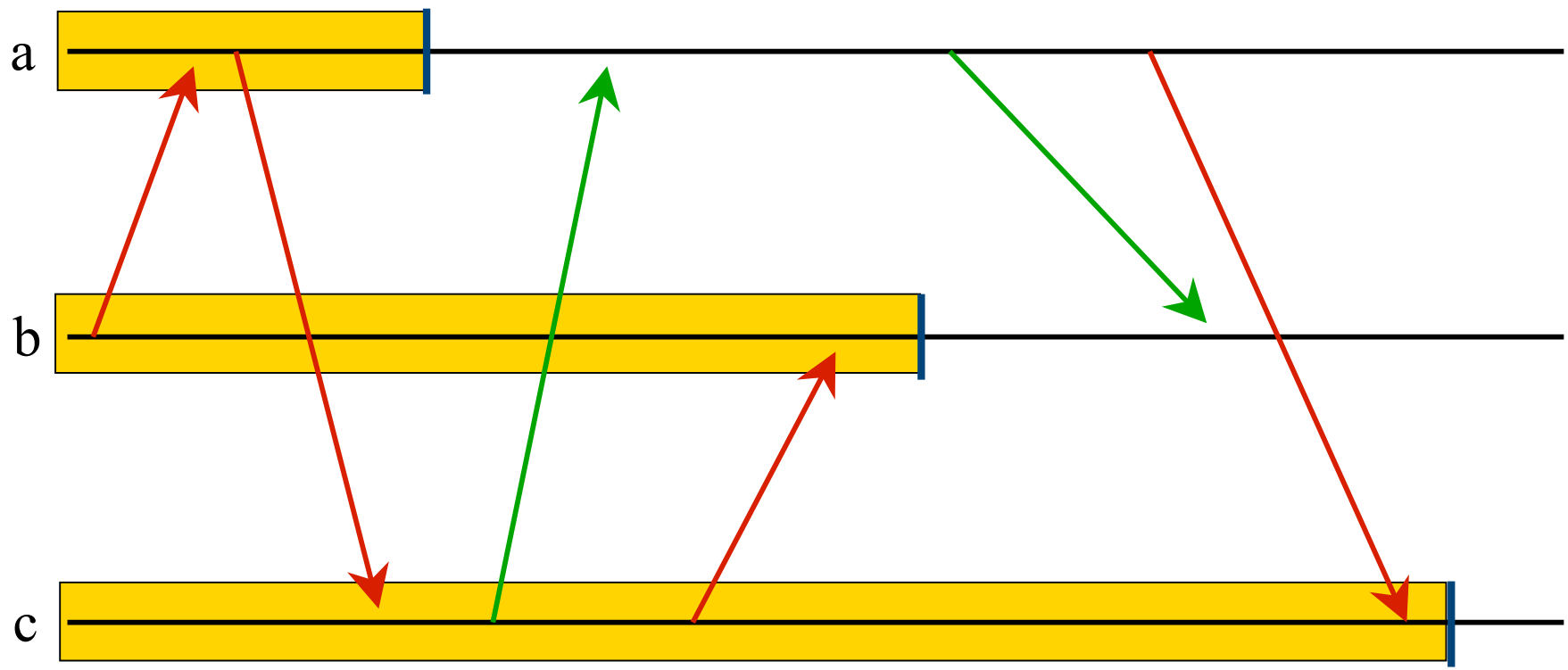
Misdetecting deadlock



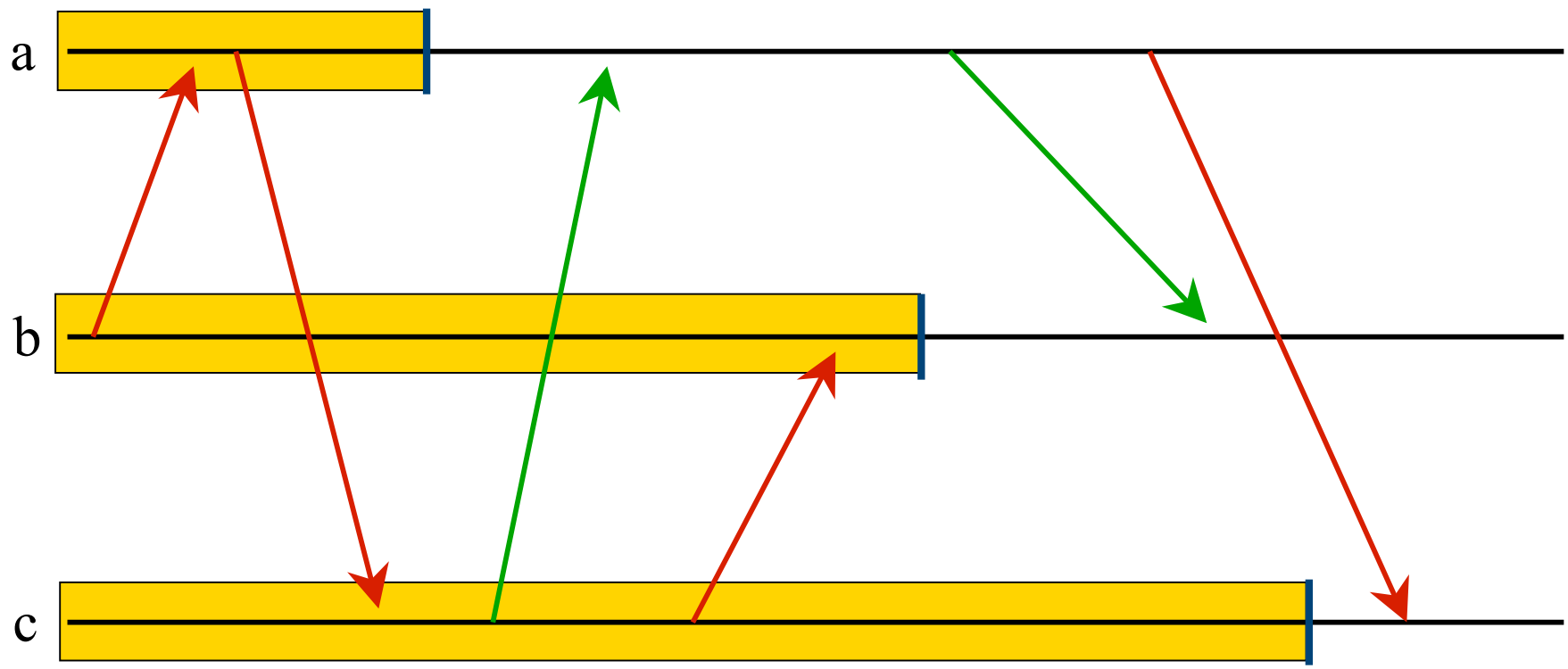
Consistent cut

- A *global state* C is a set of event sequences $\{\sigma_a, \sigma_b, \sigma_c, \ldots\}$, 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.

Inconsistent cut and global state



Consistent cut and global state



Prove:

- A cut is consistent iff all of states immediately following the cut are concurrent.
  - assume not concurrent
    - $s_i \rightarrow s_j$ implies a send event e_i after s_i and a receive event e_j before s_j: $e_i \rightarrow e_j$.
    - since e_j before s_j, e_j in cut.
    - since e_i after s_i, e_i not in cut.
    - thus, not a consistent cut.
  - assume not consistent cut
    - there is a send event e_j of i and a receive event e_j of j: $e_i \rightarrow e_j$ where e_j in cut and e_j is not in cut.
    - let e'_i be the last event of i and e'_j be the last event of j in the cut. we know $e'_i \rightarrow e_i$, $e_j \rightarrow e'_j$ or $e_j = e'_j$, and $e'_i \rightarrow e'_j$.
    - thus, (state following e'_i) $\rightarrow$ (state following e'_j) and so are not 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).

Step 1 (take at T_s): Pseudocode

```
px: send(px, Ts);

pi: when (receive(Ts) for the first time, from pj)
      for (each neighbor pk ≠ pj) send(pk, Ts);
    when (Ci == Ts) {
      record local state Si;
      for (each neighbor pk) {
        send(pk, ⊥);
        record messages Qk,i received from pk
          sent before Ts;
        }
      send(px, Si, 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

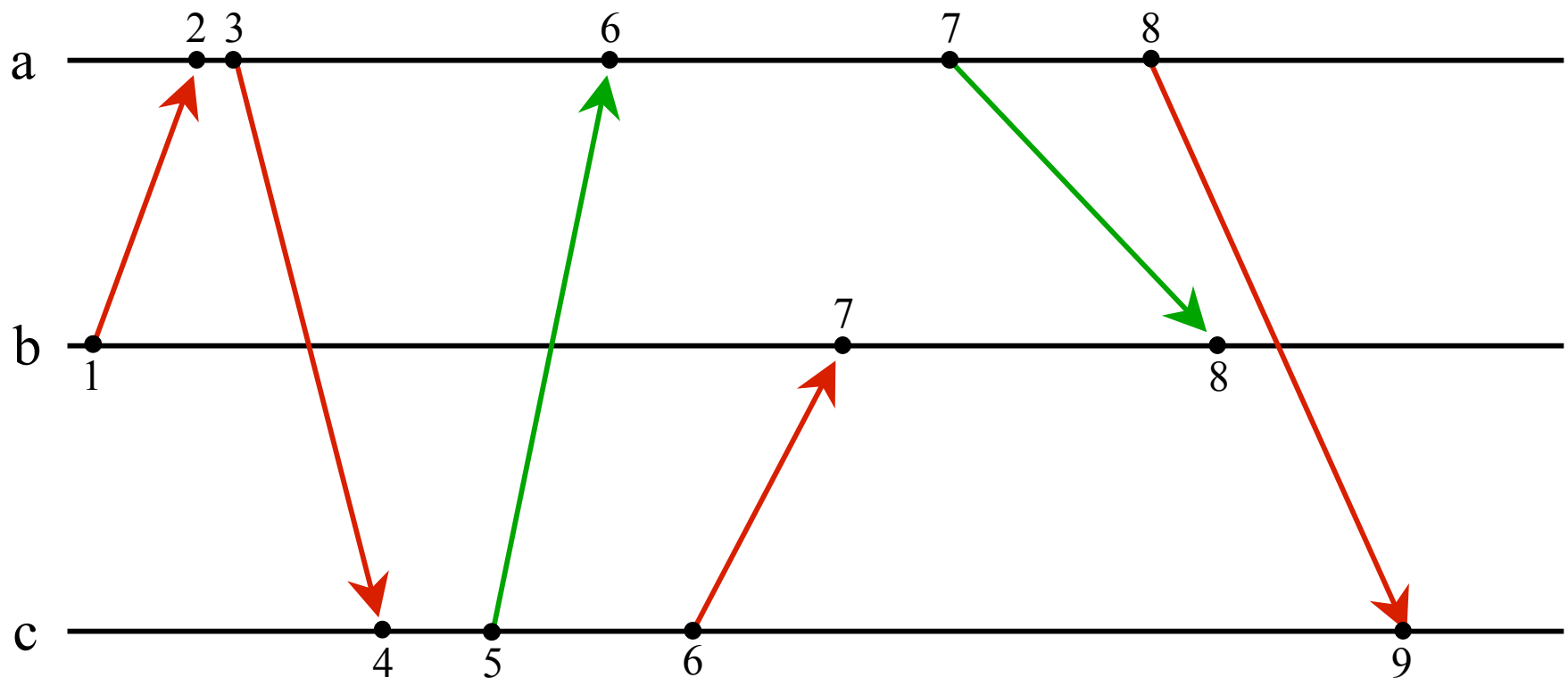
Logical Clocks

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 $\perp$.

Step 2: Pseudocode

```
p_x: send(p_x, ⊤_s ω);
     C_i = ω

p_i: when (receive(⊤_s ω) for the first time, from p_j)
       for (each neighbor p_k ≠ p_j) send(p_k, ⊤_s ω);
     when (C_i ——— ⊤_s 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 ⊤_S ω;
         }
       send(p_x, S_i, Q_*,i);
       }
```

Step 2: Pseudocode

```
px: send(px, ω);
    Ci = ω;

pi: when (receive(ω) for the first time, from pj)
      for (each neighbor pk ≠ pj) send(pk, ω);
    when (Ci passes through ω) {
      record local state Si;
      for (each neighbor pk) {
        send(pk, ⊥);
        record messages Qj,i received from pk
          sent before ω;
        }
      send(px, Si, Q*,i);
      }
```

Step 3

```
px: for (each neighbor pj) send(pi, ω);
    Ci = ω;
```

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

Step 3 (continued)

```
pi:  when (receive(ω) for the first time, from pj)
         for (each neighbor pk ≠ pj) send(pk, ω);
     when (Ci passes through ω) {
         record local state Si;
         for (each neighbor pk) {
                 send(pk, ⊥);
                 record messages Qj,i received from pk
                         sent before ω;
                 }
         send(px, Si, 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)

```
px: send(px, "Take ss");

pi: when (receive("Take ss") for the first time,
        from pj)
      record local state Si;
      for (each neighbor pk) {
        send(pk, "Take ss");
        if (pk ≠ pj)
          record messages Qk,i received from pk
          until receive(pk, "Take ss");
        else Qj,i = ∅
        }
      send(px, Si, Q*,i);
      }
```

The state that is captured

Let

- S_i be the global state in which the snapshot started
- S_f be the global state in which the snapshot finished
- S^* be the global state the snapshot captured

Consider the actual behavior that went through S_i and S_f. There is an equivalent behavior that goes through S_i, S^*, and S_f.

Proof

Consider the system behavior as a sequence of events.

Label each event as *pre* or *post* depending on whether it happened before snapshot at that process or not.

All events before S_i are *pre*, and all events after S_f are *post*, but in between there may be *post* events before *pre* events.

Consider two consecutive events e_1; e_2 in the behavior such that e_1 is *post* and e_2 is *pre*.

- Can e_1 and e_2 be of the same process?
- Can e_1 be a send with e_2 the corresponding receive?

Proof

- By swapping events like e_1 and e_2 events the behavior can be reordered to an equivalent one in which all *pre* events precede all *post* events.
  - The prefixes of *pre* events comprise the consistent cut.
  - The reordering of events does not reorder any *pre* send events nor *post* receive events

  … so the channel states are accurate.

Two questions

What use is $S_i \rightsquigarrow S^* \rightsquigarrow S_f$?

What happens if two processes start snapshots concurrently?

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* iff waits-for* graph has cycle.
- $\Box$(deadlock* $\Rightarrow$ deadlock) and $\Box$(deadlock $\Rightarrow \Diamond$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*.