AARHUS UNIVERSITY

DEPARTMENT OF ENGINEERING

Test of Distributed Systems
Lecture 10

Distributed Snapshot Algorithms

Example: Chandy-Lamport

Today's lecture

- **Distributed snapshots**
- Chandy-Lamport's snapshot algorithm
- Properties of snapshots
- Next time, exercises

Distributed snapshots

- The ability to create discrete snapshots of global state is quite useful
- Apart from GPE, also useful in other situations
  - Take snapshots at suitable points in time, roll back and restart in case of system failure(s)
  - Create “system dump” in case of detected failure (for debugging purposes)
  - Etc.

Distributed snapshots

- Desirable properties of a snapshot algorithm
  - Superimposed on (or running alongside) underlying computation
  - May send messages and require dedicated computations
  - May not alter the underlying computation
  - Should leave a footprint as small as possible (memory, cpu, network bandwidth)

Today's lecture

- Distributed snapshots
- **Chandy-Lamport's snapshot algorithm**
- Properties of snapshots
- Next time, exercises

Chandy-Lamport's distributed snapshot algorithm

- Rather than "develop" it in a sequence of analytical steps, it will just be presented
- Afterwards we will prove that it actually works
- The algorithm has two parts, an easy one and a more tricky one:
  - **Easy:** Record local state when a special message is received
  - **Tricky:** Record channel state in a way that assures that we create a consistent snapshot

Chandy-Lamport's distributed snapshot algorithm

- Works by sending special "Marker messages"
- Apart from registering local state, each process p_i records the state of each incoming *channel* (unidirectional communications connection) $\chi_{j,i}$ from other processes, where

  $\forall j \neq i : \chi_{j,i}$ *are the set of messages sent from* p_j *but not yet received by* p_i
- I.e. we not only record local process states but also messages that were in transit when the snapshot was taken

Chandy-Lamport's distributed snapshot algorithm – details 1/4

- **Marker Receiving Rule (MRR) for process p_i on receiving marker from process p_f on an incoming channel :**

1. **if** (first marker seen)
   Record local process state σ_i
   Execute Marker Sending Rule (MSR)
   else
   Stop recording messages on channel from p_f
   Declare channel state $\chi_{f,i}$ to be those messages recorded
2. Carry on normal computation

Chandy-Lamport's distributed snapshot algorithm – details 2/4

- **Marker Sending Rule (MSR) for process p_i :**
  1. Relay marker message on all outgoing channels
  2. Set the state of each incoming channel $\chi_{j,i}$ to be an empty set of messages
  3. Start recording non-marker messages $\chi_{x,i}$ on all incoming channels from processes p_x, $x \neq f$, where p_f was the one from which first marker message came
  4. Carry on normal computation
- ***Important:*** No event processing of the underlying computation takes places between step 1 & 2

Chandy-Lamport's distributed snapshot algorithm – details 3/4

- **Recording Completion Rule (RCR) for process p_i :**
  1. When a marker has been received on all incoming channels (and recording thus terminated on all channels), p_i has completed its role in capturing snapshot
  2. Report recorded local state σ_l , and the set of all recorded channel states $\chi_{f,l}$, to observer
  3. Carry on normal computation

Chandy-Lamport's distributed snapshot algorithm – details 4/4

- **Recording Initiation Rule (RIR) for process p_i :**
  - Any process p_i that wants to take a snapshot simply executes the Marker Sending Rule

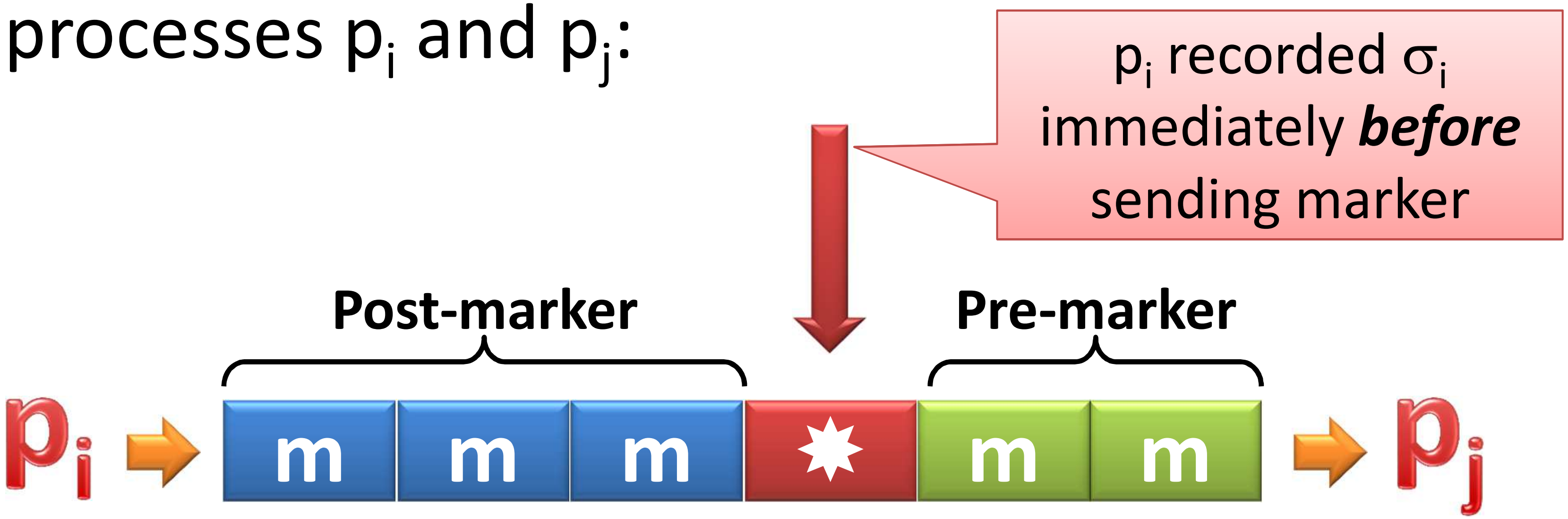
Chandy-Lamport's distributed snapshot algorithm – does it work?

- We need to assure ourselves that the algorithm produces consistent observations
- Remember: an observation/cut C is ***consistent*** if $\forall$ **events e, e': (e $\in$ C) $\wedge$ (e' $\rightarrow$ e) $\Rightarrow$ e' $\in$ C**
- That translates into: *every message that is recorded as received must also be recorded as sent*

  (i.e we cannot include the effect of receiving a message if we have no record of it being sent, that would put effect before cause)

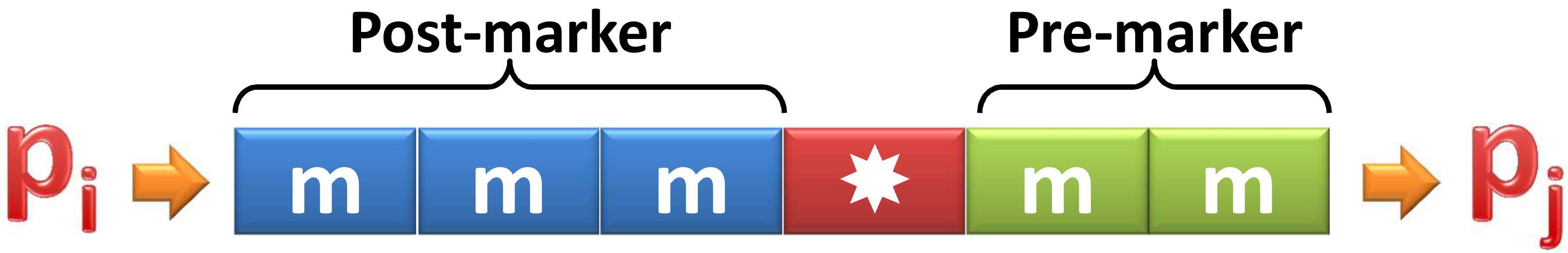
Chandy-Lamport's distributed snapshot algorithm – does it work?

- Let's look at the channels between any two processes p_i and p_j:



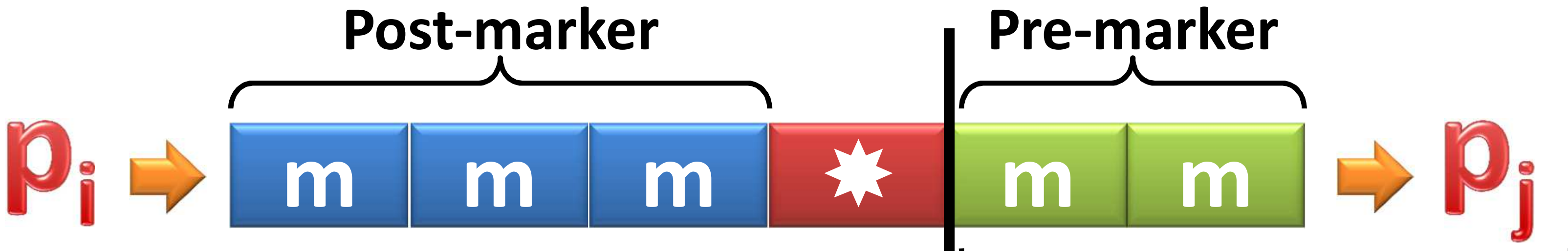
- What do we know? When was pre- and post-marker messages sent in relation to the time where p_i recorded its local state σ_i ?

Chandy-Lamport's distributed snapshot algorithm – does it work?

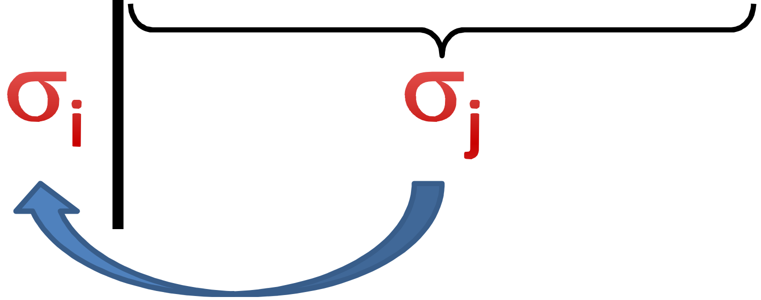


- So pre-marker messages are always included in σ_i and always included in σ_j (directly or as part of recorded channel state $\chi_{i,j}$)
- Post-marker messages are never included in σ_i and never included in σ_j (because p_j will always form σ_j first or stop recording on channel)
- So pre-marker messages are always, and post-marker messages never, part of the snapshot in p_j and p_i

Chandy-Lamport's distributed snapshot algorithm – does it work?



All pre-marker messages that somehow may influence σ_j are recorded in the state σ_i that p_i recorded when it inserted the marker



1. For each recorded recv-message in p_j (either directly or as part of channel state), p_i has recorded a send-message
2. Hence the combined snapshot is consistent

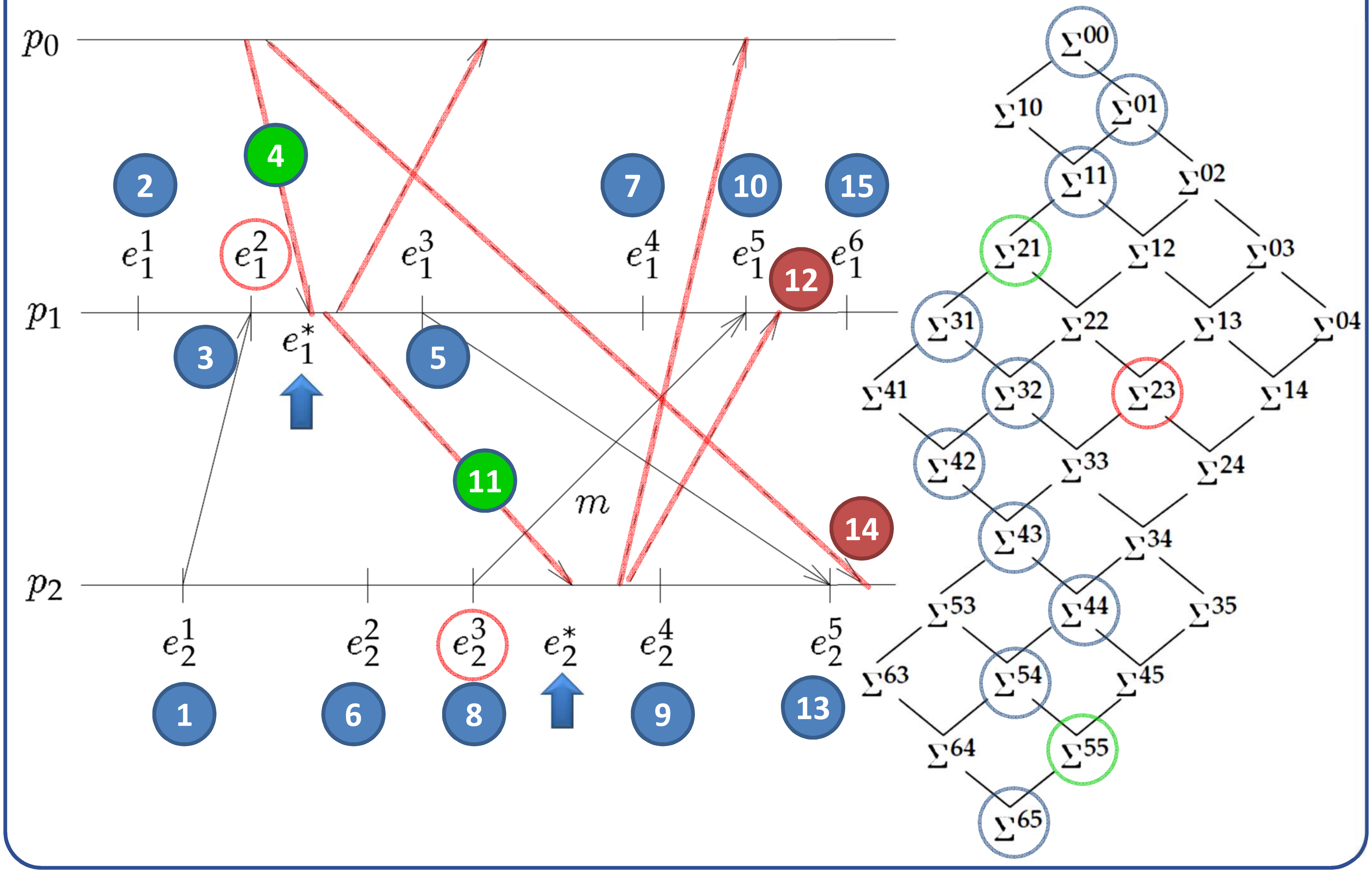
Today’s lecture

- Distributed snapshots
- Chandy-Lamport’s snapshot algorithm
- **Properties of snapshots**
- Next time, exercises

Properties of (CL) snapshots

- We know a global state Σ^s constructed via CL is consistent. But the actual system run may not even pass through Σ^s .
- However, Σ^s is not some arbitrary state, it has useful properties closely related to the run that created it.
- Let’s look at an example …

Properties of (CL) snapshots -example



Properties of (CL) snapshots - example

- Run is at state Σ^{21} when snapshot starts and in state Σ^{55} when snapshot is complete
- The actual run does not pass through the constructed global state Σ^{23}
- However $\Sigma^{21} \rightarrow_R \Sigma^{23} \rightarrow_R \Sigma^{55}$ in this example
- I.e. the constructed state lies somewhere between the start and end state
- We'll show that this holds in general

Properties of (CL) snapshots – proof of $\Sigma^{first} \rightharpoonup_R \Sigma^{snapshot} \rightharpoonup_R \Sigma^{last}$

- Let r = actual run, Σ^f, Σ^l be first/last global state involved during protocol run, Σ^s global state constructed

$\Sigma^f = \Sigma^{21}$ Σ^s $\Sigma^l = \Sigma^{55}$

$$r = e_2^1\ e_1^1\ \mathbf{e_1^2}\ \overbrace{e_1^3\ e_2^2\ e_1^4\ e_2^3\ e_2^4\ e_1^5}^{\Sigma^s}\ \mathbf{e_2^5}\ e_1^6$$

Σ^s is sandwiched between Σ^s and Σ^l

- We want to show that another consistent run R always exists so that $\Sigma^f \rightharpoonup_R \Sigma^s \rightharpoonup_R \Sigma^l$

Properties of (CL) snapshots – proof of $\Sigma^{\text{first}} \rightarrow_R \Sigma^{\text{snapshot}} \rightarrow_R \Sigma^{\text{last}}$

- Step 1: Swap pre- and post-recording events
  - For each process pi that receives 1st marker event e_i*, any other event e_i is either pre-recording ($e_i \rightarrow e_i$*) otherwise it is post-recording
  - $\forall$ e,e' in run: (e post-rec) $\wedge$ (e' pre-rec) $\Rightarrow$ $\neg$(e$\rightarrow$e') I.e. no pre-rec event is causally dependent on a post-rec event
    - Trivially true for two events e, e' in same process
    - If e is send-event of p_i and e' is corresponding receive event of p_j, p_i will have sent marker-msg before e, so e' will also be post-recording

Properties of (CL) snapshots – proof of $\Sigma^{first} \rightarrow_R \Sigma^{snapshot} \rightarrow_R \Sigma^{last}$

- Step 1: Swap pre- and post-rec events (contd.)
  - Since no pre-rec event is causally dependent on a post-rec event, we can form a new run R from r by sorting events so that all pre-rec events comes before post-rec events

Σ^s formed

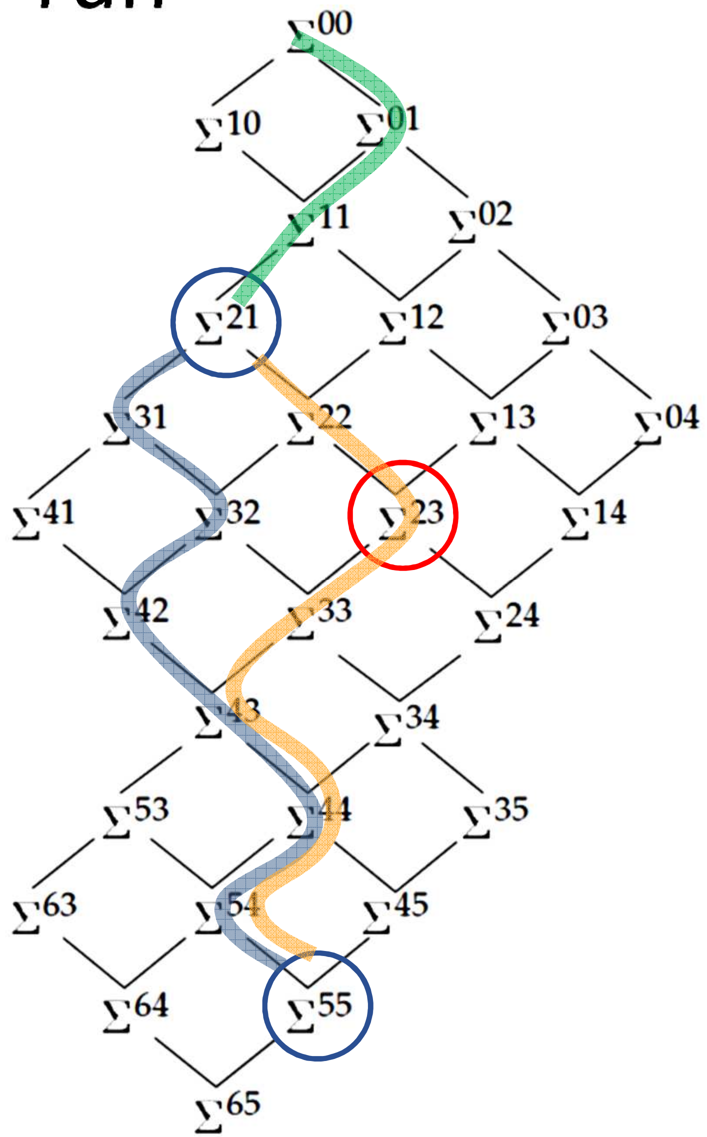
$$r = e_2^1\ e_1^1\ \mathbf{e_1^2}\ e_1^3\ e_2^2\ e_1^4\ \mathbf{e_2^3}\ |\ e_2^4\ \mathbf{e_1^5}\ \mathbf{e_2^5}\ e_1^6$$

$$R = e_2^1\ e_1^1\ \mathbf{e_1^2}\ e_2^2\ \mathbf{e_2^3}\ |\ e_1^3\ e_1^4\ e_2^4\ \mathbf{e_1^5}\ \mathbf{e_2^5}\ e_1^6$$

- Snapshot state Σ^s formed after last pre-rec event
- Σ^s reachable from Σ^f and Σ^l reachable from Σ^s

Comparing the original run with the “sorted” run

e_2^1	01	e_2^1	01
e_1^1	11	e_1^1	11
e_1^2	21	e_1^2	21
e_1^3	31	e_2^2	22
e_2^2	32	e_2^3	23
e_1^4	42	e_1^3	33
e_2^3	43	e_1^4	43
e_2^4	44	e_2^4	44
e_1^5	54	e_1^5	54
e_2^5	55	e_2^5	55
e_1^6	65	e_1^6	65



Properties of (CL) snapshots – proof of $\Sigma^{first} \rightarrow_R \Sigma^{snapshot} \rightarrow_R \Sigma^{last}$

- Hence we have shown that given any run r, there exists another run R, such that $\Sigma^f \rightarrow_R \Sigma^s \rightarrow_R \Sigma^l$
- Step 2: We also need to show that the state Σ^s formed after the last pre-rec event in R, is the same as the one formed by the protocol
  - Each process p_i records its local state σ_i in the same relative spot in R with respect to all events e_i
  - Each process p_i records the same channel state(s), because, as seen by process p_i, the sequence of pre-rec messages, marker and post-rec messages remain the same

Today’s lecture

- Distributed snapshots
- Chandy-Lamport’s snapshot algorithm
- Properties of snapshots
- **Next time, exercises**

Next time

- Exercise on monday
  - We'll hear some (informal) presentations of suggested solutions for improving the "broadcast token passing algorithm" for DS
- Reading material
  - Nothing