Global Snapshots

Global State

- Changes when events occur (local, messages)
- Capturing global state in distributed systems is challenging because of asynchronous nature of computation and communication
- Time based snapshots: Every process saves its own state at the “same time”
- Analogy: Composite picture of flying birds in the sky
  - Can’t capture entire field of view in single snapshot
  - Multiple snapshots necessary to get a global picture
  - But birds can move around etc. What if we want to count total number of birds?
- This class: event-based: using happened before relationships

Why Global Snapshots

- Checkpointing: If application fails, resume from earlier state’s snapshot
- Debugging
- Garbage collection: delete unreferenced objects
- Useful to detect “Stable properties”
  - A stable property persists throughout application execution
  - Such as termination, deadlock
- If a stable property holds before snapshot begins, it holds in the recorded global snapshot

Single Process Checkpointing

- Local process state saved to stable storage (disk)
- Offline: stop process execution and save all local state
- Online/Live: process continues executing when snapshot is taken

Snapshot Requirements

- “Live”: applications shouldn’t stop sending messages / making forward progress
- Each process can take snapshot of local state
- Any process can initiate snapshot

Cuts

- Snapshots also referred to as “cuts”
- Line joining arbitrary point in time on each process that slices the space-time diagram into a past and future
- A cut is a set of local states of processes, and state of all communication channels (messages in transit)
- Consistent cut: for any received message in the cut, the send event must also be in the cut
- Snapshots must comprise of concurrent events
- Consistent cut C is subset of events s.t.: for all e in C: If d $\rightarrow$e, then d is in C

Distributed Global Snapshot Challenges

- Recorded global states are mutually concurrent
- State of communication channels is captured somehow
  - Let processes record sent messages
- Basic Idea: Processes send a “marker” message to initiate/propagate a snapshot
- 2 states: White: no marker rcvd. Red: marker rcvd
- Once a process turns red, it must send marker along all outgoing channels before sending any message
- Processes in red state start recording all incoming messages

Message Types

- ww: Sent and received by white processes, before global snapshot
- rr: Sent and received by red processes after snapshot
- rw: Sent by red, but received by white. These cross the cut in backward direction and make the cut inconsistent.
- wr: Cross the cut in the forward direction and are part of the global state.
- FIFO assumption: If you receive a marker from a process, then all subsequent messages from it will be rr and need not be recorded.

Chandy-Lamport

- Process save local state and state of all incoming communication channels
- Initiates snapshot by turning red, and sending special “marker” message to all others
- Start recording all incoming messages
- Termination: When each process has received a marker on all its incoming channels

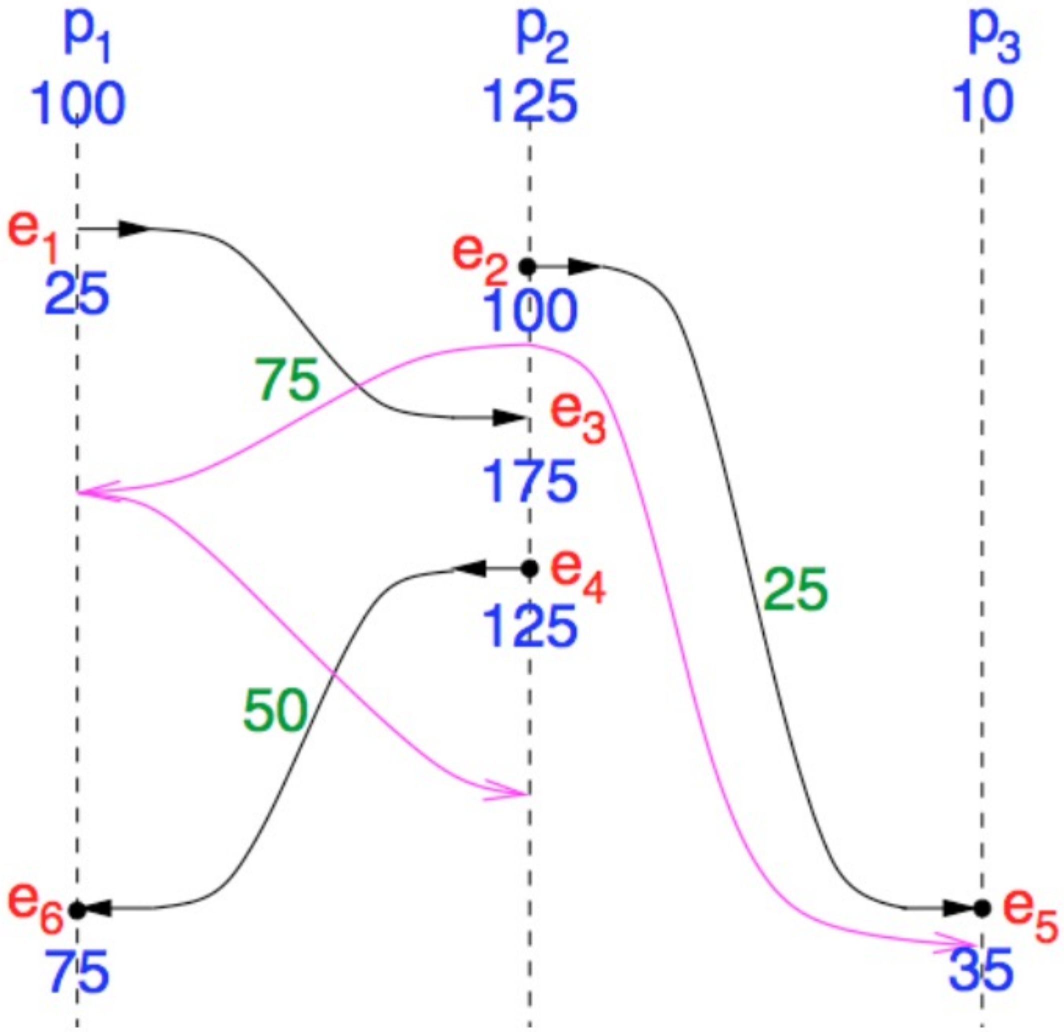
Chandy-Lamport Algorithm

```
def turn_red() enabled if (color==white):
      save_local_state;
      color = red ;
      send(marker) to all neighbors

def receive(marker) on incoming channel j:
      if(color==white):
            turn_red();
      closed[j] = true; #Initialized to false

def receive(message) on incoming channel j:
      if(color==red and not closed[j]):
            chan[j].append(message)
```

p_2 initiates the algorithm



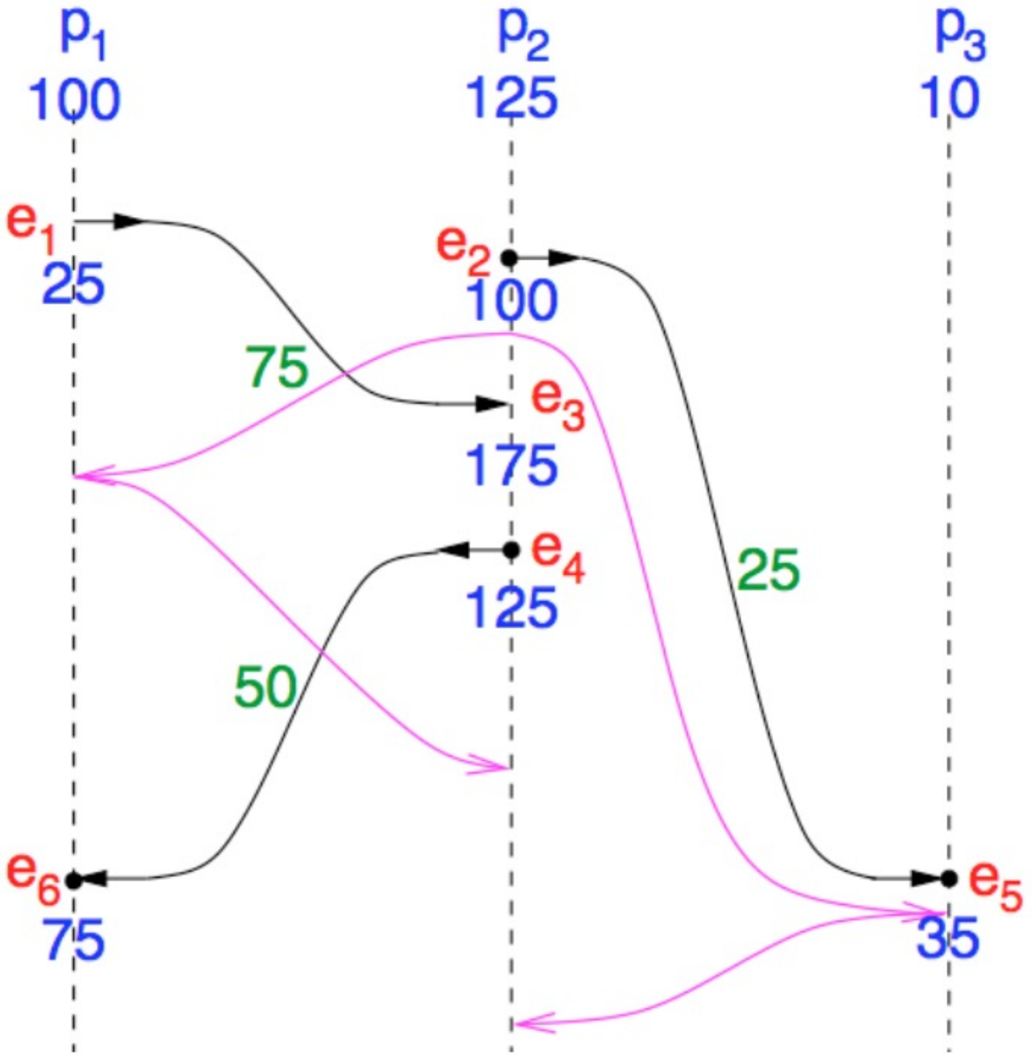
$p2: \quad 100$

$p1: \quad 25 \quad c_{21} = <>$

$p3: \quad 35 \quad c_{23} = <>$

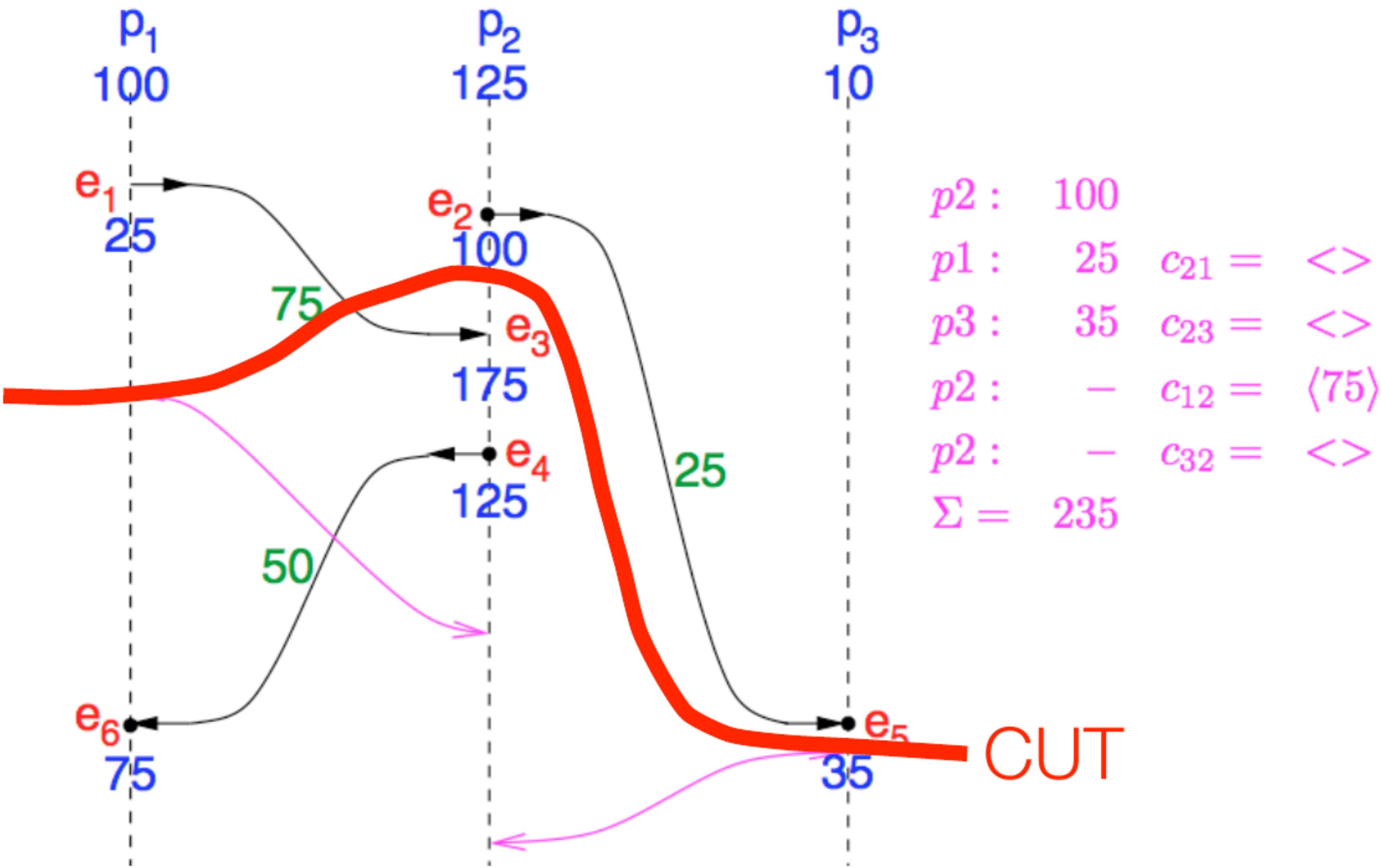
$p2: \quad - \quad c_{12} = \langle 75 \rangle$

p_2 initiates the algorithm



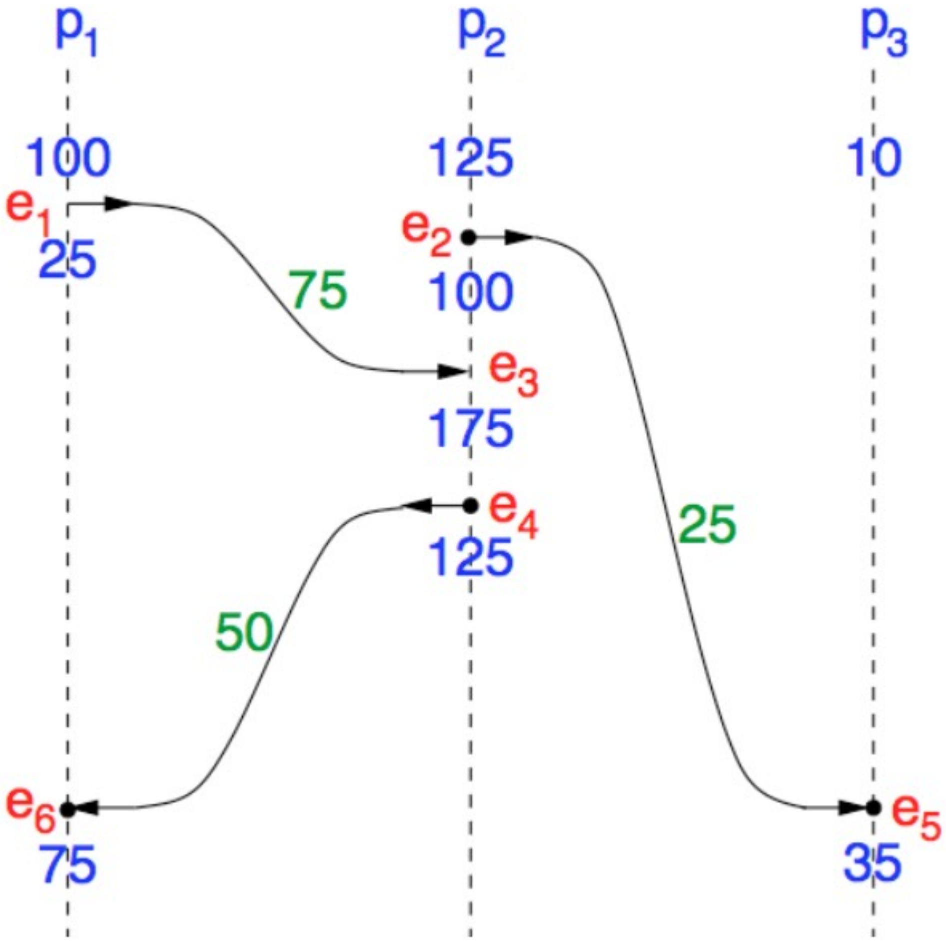
$$
\begin{array}{llll}
p2: & 100 & & \\
p1: & 25 & c_{21} = & <> \\
p3: & 35 & c_{23} = & <> \\
p2: & - & c_{12} = & \langle 75 \rangle \\
p2: & - & c_{32} = & <> \\
\Sigma = & 235 & &
\end{array}
$$

p_2 initiates the algorithm



Homework

- Snapshot initiated by p1 just after e1



Vector Clock View

- Cut: set of states from each process.
- Consistent: States should be pairwise concurrent
  - Easy to verify with vector clocks
- Intuition: All processes take snapshot at future time ‘K’
- Process i initiates checkpoint at K = V_i + increment local component
  - Broadcasts K ;
  - All processes take local snapshot when they reach time stamp K
  - Initiator process sends second `dummy’ broadcast

Vector Clock View

- Let Vi be the vector clock of process i exactly at i’s cut-point. Let K = max(V_1, V_2, … V_n).
- Thm: Cut is consistent iff for every i, K(i) = V_i(i)
- That is, the maximum information about process-i that is known by anyone at the cut is the same as what it knows about itself at its cut point
  - No one else knows more about I than I know myself know
- This rules out receiving message before its cutpoint that was sent after its cut-point, because otherwise the recipient would have more info about the sender than the sender had about itself.

Vector Clock View Continued

- Let Vi be the vector clock of process i exactly at i’s cut-point. Let K = max(V_1, V_2, … V_n).
- Thm: Cut is consistent iff for every i, K(i) = V_i(i)
- Restatement: for every i and j, $V_j(i) \leq V_i(i)$
- All events before the snapshot happen-before all events after the snapshot