Distributed and Parallel Computing

Lecture 11

Alan P. Sexton

University of Birmingham

Spring 2019

Snapshots

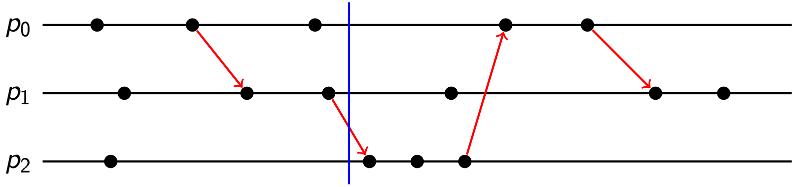
In a sequential program it is not unusual to save snapshots of the program state:

- Allows exiting and restarting to continue later on
- Guards against losing all the work after a system crash
- Allows returning to a previous good state if something bad happens (e.g. a poor choice is made, a character in a game dies, etc.)

Since the program is sequential, it is easy to access the state of the program and save a consistent snapshot of that state.

Distributed Snapshots

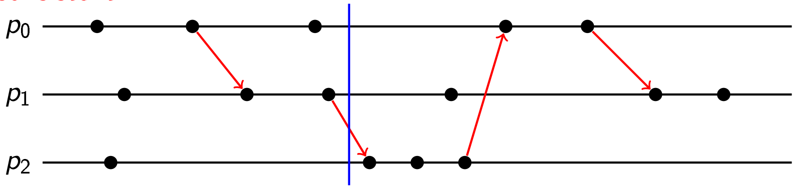
In a distributed program a snapshot must record the state of all the nodes and the state of all the channels (i.e. what messages are in transit on each channel) in such a way that the results are *consistent*:



Try: agree on a time in the future when all nodes and channels save their local states

Distributed Snapshots

In a distributed program a snapshot must record the state of all the nodes and the state of all the channels (i.e. what messages are in transit on each channel) in such a way that the results are *consistent*:

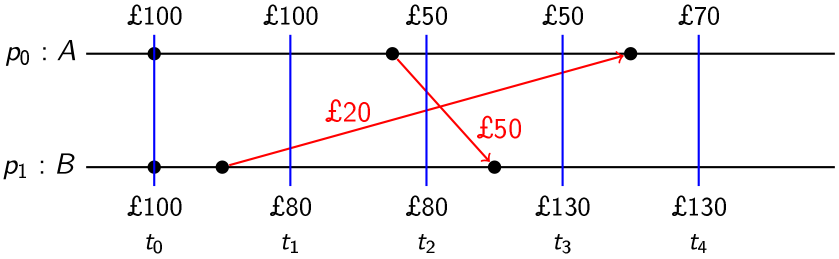


Try: agree on a time in the future when all nodes and channels save their local states

- **NOT POSSIBLE**: no global real time clock

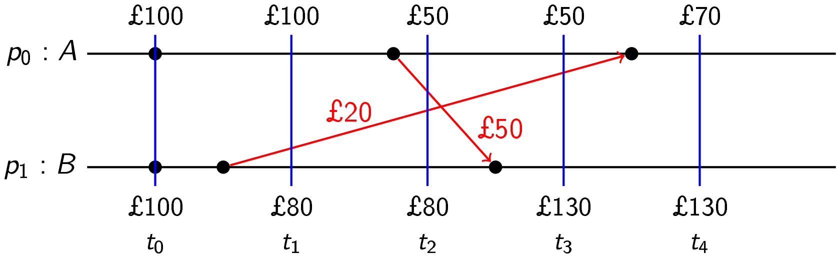
Consistency Example in Distributed Snapshots

A system has two bank nodes maintaining accounts A and B. Initially both accounts contain £100. Two transfers: £50 from A along channel c_{01} to B and £20 from B to A along channel c_{10}:



Consistency Example in Distributed Snapshots

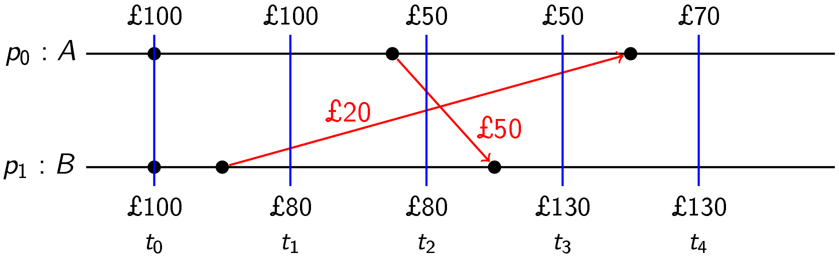
A system has two bank nodes maintaining accounts A and B. Initially both accounts contain £100. Two transfers: £50 from A along channel c_{01} to B and £20 from B to A along channel c_{10}:



- Snapshot taken on p_0 at time t_1, and on c_{01}, c_{10} and p_1 at time t_3 giving values £100, £0, £20 and £130 respectively $\Rightarrow$ £250 in the system — **NOT CONSISTENT**

Consistency Example in Distributed Snapshots

A system has two bank nodes maintaining accounts A and B. Initially both accounts contain £100. Two transfers: £50 from A along channel c_{01} to B and £20 from B to A along channel c_{10}:



- Snapshot taken on p_0 at time t_1, and on c_{01}, c_{10} and p_1 at time t_3 giving values £100, £0, £20 and £130 respectively $\Rightarrow$ £250 in the system — **NOT CONSISTENT**
- Snapshot taken on p_0 at time t_1, and on c_{01}, c_{10} and p_1 at time t_2 giving values £100, £50, £20 and £80 respectively $\Rightarrow$ £250 in the system — **NOT CONSISTENT**

Inconsistency in Distributed Snapshots

An inconsistent snapshot does not correspond to any possible configuration. That is:

- There is no possible execution of the system such that an inconsistent snapshot captures the true global state at any point in that execution

Recording of local snapshots must be coordinated correctly to ensure a consistent global snapshot.

If each process takes a local snapshot, then

- An event is *pre-snapshot* if it occurs in a process *before* the *local* snapshot in that process is taken
- Otherwise it is *post-snapshot*

Since the global snapshot includes all the local snapshots, all pre-snapshot events will be included in the global snapshot and all post-snapshot events will not be.

Consistency in Distributed Snapshots

A global distributed snapshot is *consistent* if:

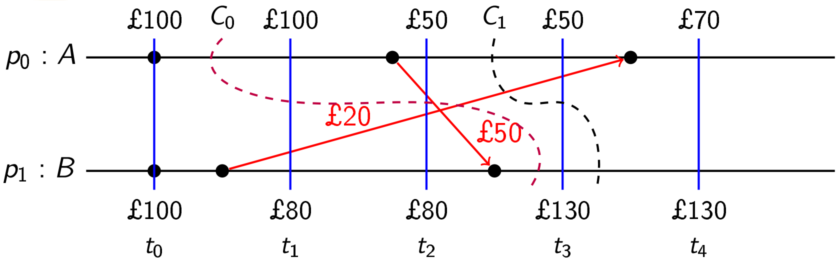
1. When a is pre-snapshot, $x \prec a \Rightarrow x$ is pre-snapshot
   - This ensures that the snapshot captures a point in a *computation*: i.e. a viable permutation of the concurrent events of the execution.
   - Note that x could be in a different process to a
2. A message is included in the channel state if and only if its sending is pre-snapshot and its reception is post-snapshot
   - This ensures that no message can be recorded as received in the snapshot if it is not also recorded as sent in the snapshot, and no message recorded as sent can be lost.

Thus, for message m with send and receive events s and r:

- if s is pre-snapshot, then
  - r is post-snapshot $\Rightarrow$ m is recorded in the channel state
  - r is pre-snapshot $\Rightarrow$ the effects of the message are recorded in the local node snapshots
- if s is post-snapshot $\Rightarrow$ r is also post-snapshot and m is not part of the global snapshot

Cuts

Visualise pre- and post-snapshots with *cuts*: a continuous line that cuts through each process line once:

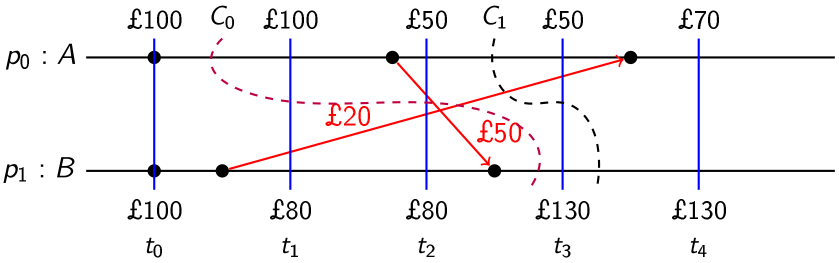


A consistent global state corresponds to a cut where every message received in the logical past of that cut was also sent in the logical past of that cut:

- C_0:

Cuts

Visualise pre- and post-snapshots with *cuts*: a continuous line that cuts through each process line once:

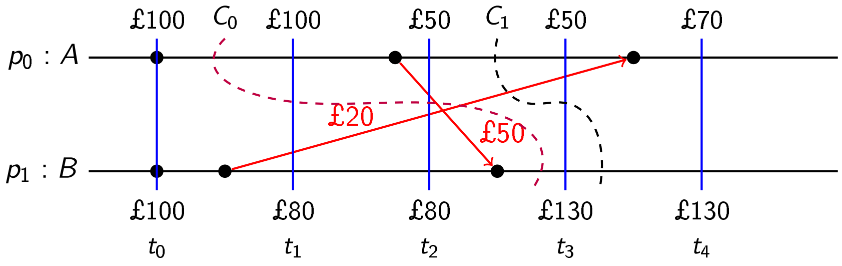


A consistent global state corresponds to a cut where every message received in the logical past of that cut was also sent in the logical past of that cut:

- C_0: The message from A to B was received in the logical past of the cut but sent in the logical future: **NOT CONSISTENT**
- C_1:

Cuts

Visualise pre- and post-snapshots with *cuts*: a continuous line that cuts through each process line once:



A consistent global state corresponds to a cut where every message received in the logical past of that cut was also sent in the logical past of that cut:

- C_0: The message from A to B was received in the logical past of the cut but sent in the logical future: **NOT CONSISTENT**
- C_1: All messages received in the logical past of the cut were sent in the logical past of the cut: **CONSISTENT**

Distributed Snapshot Algorithms

A simple distributed snapshot algorithm could come to a quiescent state, then, when all processes are quiescent, take all the local snapshots.

- This is inefficient: the whole system has to come to a stop to take a snapshot
- The system has to be designed so that quiescent states are possible — not always feasible

A better solution is to allow snapshots to be taken without interfering with the underlying processing.

- *Basic Algorithm*: the underlying distributed algorithm that we are taking a snapshot of. This algorithm uses *Basic Messages*
- *Control Algorithm*: the higher level distributed algorithm, in this case the global snapshot algorithm, implemented using *Control Messages*

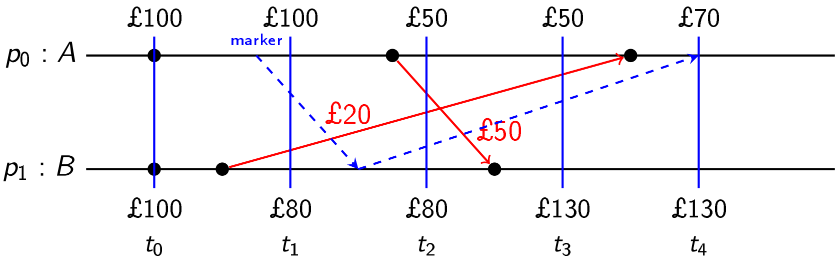
Chandy-Lamport Algorithm

NOTE: Applies to FIFO channel systems only

Idea: send control messages called *markers* along channels to separate pre- and post-snapshot events and trigger local snapshots.

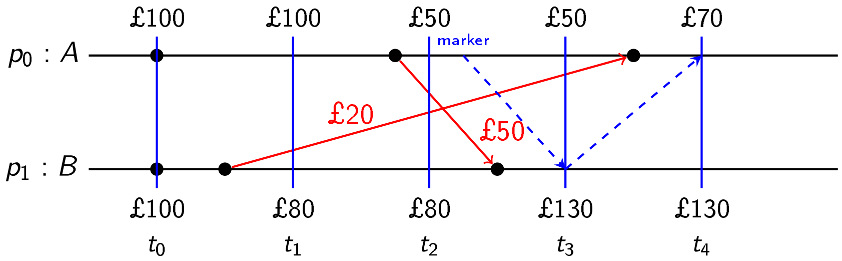
- Initiator takes a local snapshot and sends a marker through all outgoing channels
- On process p_j receiving a marker along channel c_{ij}
  - if p_j has not yet saved its local state
    - p_j saves its local state
    - p_j sets channel state c_{ij} to $\{\}$
    - p_j sends a marker through all outgoing channels
  - else
    - p_j records the state of channel c_{ij} as the set of all basic messages received on c_{ij} after it saved its local state and before it received the marker message from p_i
- Algorithm terminates at each process when it has received a marker along every incoming channel
- Note: a marker message is sent once along every channel in every allowed direction

Chandy-Lamport Example 1



- p_0 is the initiator, records local state $A = £100$, and sends a marker to p_1
- FIFO channels $\Rightarrow$ p_1 receives marker before £50 message
- p_1 records local state $B = £80$, sets channel state $c_{01} = \{\}$ and sends a marker to p_0
- FIFO channels $\Rightarrow$ p_0 receives marker after £20 message
- p_0 records channel state $c_{10} = \{£20\}$

Chandy-Lamport Example 2



- p_0 is the initiator, records local state $A = £50$ and sends a marker to p_1
- FIFO channels $\Rightarrow$ p_1 receives marker after £50 message
- p_1 records local state $B = £130$, sets channel state $c_{01} = \{\}$ and sends a marker to p_0
- FIFO channels $\Rightarrow$ p_0 receives marker after £20 message
- p_0 records channel state $c_{10} = \{£20\}$

Correctness of Chandy-Lamport, part 1

To prove: If $a \prec b$ and b is pre-snapshot, then a is pre-snapshot

- If a and b are in the same process, then trivially true
- Suppose a is a send and b is the corresponding receive in processes p and q respectively.
  - Since b is pre-snapshot, q has not yet received a marker when b occurs.
  - Since channels are FIFO, this means p has not yet sent a marker to q when a occurs.
  - Hence a is pre-snapshot.
- This argument extends transitively to causal chains of internal, send and receive events

Correctness of Chandy-Lamport, part 2

To prove: Basic message m from p to q is in the channel state of c_{pq} if and only if the send at p is pre-snapshot and the receive at q is post-snapshot

$\Rightarrow$ direction: Assume m is in the channel state for c_{pq}:

- Since m is in the channel state of c_{pq}, the receive of m must occur after saving the local state of q, hence it is post-snapshot
- q must receive the m first and then the marker through c_{pq}
- Since channels are FIFO, p must send m first and then the marker through c_{pq}
- Since sending a marker is always immediately preceded by saving local state, the send of m must be pre-snapshot

Correctness of Chandy-Lamport, part 3

To prove: Basic message m from p to q is in the channel state of c_{pq} if and only if the send at p is pre-snapshot and the receive at q is post-snapshot

$\Leftarrow$ direction: Assume the send of m is pre-snapshot and the receive of m is post-snapshot

- The send of m is pre-snapshot $\Rightarrow$ the local state of p is saved after the send of m
- Hence a marker is sent from p to q after the send of m
- Channels are FIFO $\Rightarrow$ the receive of the marker occurs after the receive of m
- The receive of m is post-snapshot $\Rightarrow$ it occurs after the local state of q is saved
- The receive of m is between the saving of the local state of q and the receive of the marker $\Rightarrow$ m is in the channel state of c_{pq}

Lai-Yang-Mattern Algorithm

NOTE: works on NON-FIFO channels

- Here, a marker message sent before/after a basic message may arrive after/before that same basic message respectively.
- Idea: rather than having separate marker messages, piggyback the markers on basic messages, and keep track of state by marking processes and messages with boolean flags (traditionally described as the colours white and red).
- The original Lai-Yang algorithm had no control messages at all, but required keeping a full history of all messages sent and received at each node
- The Mattern algorithm uses logical clocks directly and forces processes to wait until an agreed logical time.
- The Mattern paper describes a modification of Lai-Yang that replaces the need for a full history of messages with a single control message. This algorithm is what we will study.

Lai-Yang-Mattern Algorithm

- Every process is initialised to white.

Lai-Yang-Mattern Algorithm

- Every process is initialised to white.
- When a process saves its local state, it turns red and sends a *control* message on all outgoing channels to say how many white messages in total it has sent down that channel.

Lai-Yang-Mattern Algorithm

- Every process is initialised to white.
- When a process saves its local state, it turns red and sends a *control* message on all outgoing channels to say how many white messages in total it has sent down that channel.
- Every *basic* message is the same colour as the process that sends it (a white message is sent pre-snapshot, a red is sent post-snapshot)

Lai-Yang-Mattern Algorithm

- Every process is initialised to white.
- When a process saves its local state, it turns red and sends a *control* message on all outgoing channels to say how many white messages in total it has sent down that channel.
- Every *basic* message is the same colour as the process that sends it (a white message is sent pre-snapshot, a red is sent post-snapshot)
- A white process can save its local state at any time, but must save it no later than on receiving a red message and **BEFORE** processing that message

Lai-Yang-Mattern Algorithm

- Every process is initialised to white.
- When a process saves its local state, it turns red and sends a *control* message on all outgoing channels to say how many white messages in total it has sent down that channel.
- Every *basic* message is the same colour as the process that sends it (a white message is sent pre-snapshot, a red is sent post-snapshot)
- A white process can save its local state at any time, but must save it no later than on receiving a red message and **BEFORE** processing that message
- Each process, on receiving the control message, immediately saves its local state if it hasn't done so already, and, knowing how many white messages it has received to date on each input channel, knows how many white messages to wait for along the corresponding channels.

Lai-Yang-Mattern Algorithm

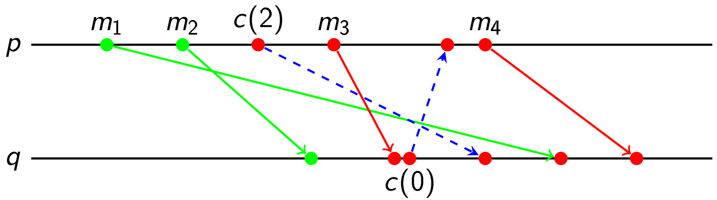
- Every process is initialised to white.
- When a process saves its local state, it turns red and sends a *control* message on all outgoing channels to say how many white messages in total it has sent down that channel.
- Every *basic* message is the same colour as the process that sends it (a white message is sent pre-snapshot, a red is sent post-snapshot)
- A white process can save its local state at any time, but must save it no later than on receiving a red message and **BEFORE** processing that message
- Each process, on receiving the control message, immediately saves its local state if it hasn't done so already, and, knowing how many white messages it has received to date on each input channel, knows how many white messages to wait for along the corresponding channels.
- After waiting for outstanding white messages, each process q computes channel state c_{pq} as the set of white messages it receives after it has saved its local state.

Lai-Yang-Mattern Algorithm details

The control message element to the algorithm is necessary for two reasons:

1. After saving its local state, a process may not have any basic messages to send for a long time (or ever!). The control message ensures that the snapshot algorithm proceeds
2. Without the control message, processes have no idea how long to wait for messages in transit, or how many such messages there might be.

Lai-Yang-Mattern Example



For clarity, white messages and nodes are drawn in green, control messages as dashed blue

1. p sends white m_1 and m_2, then saves local snapshot and sends control message to q with count of 2 white messages sent to q, then sends red m_3
2. q receives white m_2, then receives red m_3, saves local snapshot, sends control message to p with count of 0 white messages sent to p
3. q receives control message and waits to receive 1 white message (2 sent - 1 previously received), then sets channel c_{pq} state to $\{m_1\}$.
4. p receives control message, doesn't have to wait and sets channel c_{qp} state to $\{\}$

Correctness of Lai-Yang-Mattern, part 1

To prove: If $a \prec b$ and b is pre-snapshot, then a is pre-snapshot

- If a and b are in the same process, then trivially true
- Suppose a is a send and b is the corresponding receive in processes p and q respectively.
  - Since b is pre-snapshot, b is white.
  - Hence a must be white
  - Therefore a is pre-snapshot
- This argument extends transitively to causal chains of internal, send and receive events

Correctness of Lai-Yang-Mattern, part 2

To prove: Basic message m from p to q is in the channel state of c_{pq} if and only if the send at p is pre-snapshot and the receive at q is post-snapshot

$\Rightarrow$ direction: Assume m is in the channel state for c_{pq}:

- Since m is in the channel state of c_{pq}, m must be white, hence the send of m is pre-snapshot
- Since m is in the channel state of c_{pq}, the receive of m must occur after the save of the local snapshot at q, hence the receive of m is post-snapshot

Correctness of Lai-Yang-Mattern, part 3

To prove: Basic message m from p to q is in the channel state of c_{pq} if and only if the send at p is pre-snapshot and the receive at q is post-snapshot

$\Leftarrow$ direction: Assume the send of m is pre-snapshot and the receive of m is post-snapshot

- The send of m is pre-snapshot $\Rightarrow$ the local state of p is saved after the send of m, hence m is white.
- The receive of m is post-snapshot $\Rightarrow$ the local state of q is saved before the receive of m
- Hence m is a white message received after the local state of q is saved hence will be in the channel state of c_{pq}

Lai-Yang-Mattern Multiple Snapshots

As described previously, after the first global snapshot, all messages are red. How can we handle a sequence of global snapshots?

Lai-Yang-Mattern Multiple Snapshots

As described previously, after the first global snapshot, all messages are red. How can we handle a sequence of global snapshots?

- On saving local snapshot in a process:
  - All white messages received before then will be captured in the local snapshot and not relevant to the next global snapshot
  - All white messages received after then will be captured in the channel state and not relevant to the next global snapshot
  - All messages sent after then will be red, but should be considered to be white to the next global snapshot

Lai-Yang-Mattern Multiple Snapshots

As described previously, after the first global snapshot, all messages are red. How can we handle a sequence of global snapshots?

- On saving local snapshot in a process:
  - All white messages received before then will be captured in the local snapshot and not relevant to the next global snapshot
  - All white messages received after then will be captured in the channel state and not relevant to the next global snapshot
  - All messages sent after then will be red, but should be considered to be white to the next global snapshot
- Solution: instead of red/white flags, use counter k:
  - Initially processes and messages have $k = 0$
  - Instead of turning processes and messages red on saving local snapshot, increment k
  - Now, for first global snapshot, $k = 0 \Rightarrow$ white, $k = 1 \Rightarrow$ red
  - For second global snapshot, $k = 1 \Rightarrow$ white, $k = 2 \Rightarrow$ red

Lai-Yang-Mattern Multiple Snapshots

As described previously, after the first global snapshot, all messages are red. How can we handle a sequence of global snapshots?

- On saving local snapshot in a process:
  - All white messages received before then will be captured in the local snapshot and not relevant to the next global snapshot
  - All white messages received after then will be captured in the channel state and not relevant to the next global snapshot
  - All messages sent after then will be red, but should be considered to be white to the next global snapshot
- Solution: instead of red/white flags, use counter k:
  - Initially processes and messages have $k = 0$
  - Instead of turning processes and messages red on saving local snapshot, increment k
  - Now, for first global snapshot, $k = 0 \Rightarrow$ white, $k = 1 \Rightarrow$ red
  - For second global snapshot, $k = 1 \Rightarrow$ white, $k = 2 \Rightarrow$ red
- What if two different processes start a global snapshot concurrently?

Lai-Yang-Mattern Multiple Snapshots

As described previously, after the first global snapshot, all messages are red. How can we handle a sequence of global snapshots?

- On saving local snapshot in a process:
  - All white messages received before then will be captured in the local snapshot and not relevant to the next global snapshot
  - All white messages received after then will be captured in the channel state and not relevant to the next global snapshot
  - All messages sent after then will be red, but should be considered to be white to the next global snapshot
- Solution: instead of red/white flags, use counter k:
  - Initially processes and messages have $k = 0$
  - Instead of turning processes and messages red on saving local snapshot, increment k
  - Now, for first global snapshot, $k = 0 \Rightarrow$ white, $k = 1 \Rightarrow$ red
  - For second global snapshot, $k = 1 \Rightarrow$ white, $k = 2 \Rightarrow$ red
- What if two different processes start a global snapshot concurrently?
  - Both processes increment k to the same value so both global snapshots will be the same single global snapshot