Global Snapshots

Banking System

Description of the problem

- Count the total amount of money in a banking system if:
  - There are no external deposits or withdrawals
  - Money is transferred between processes via messages

Banking System

- Each process p_i has a local variable, $money_i$, that contains the amount of money currently residing at that location.
- Each process sends arbitrary amounts of its locally located money to other processes.

Problem

- Each process should decide to a local balance in such a way that the total of the balances is the correct amount of money in the system.

The CountMoney Algorithm

Assumption

- There is a predetermined logical time t, assumed to be known to all processes.
- Each process sends infinitely many messages to each of its neighbors (these could be dummy transfers of 0 euros)

General Strategy

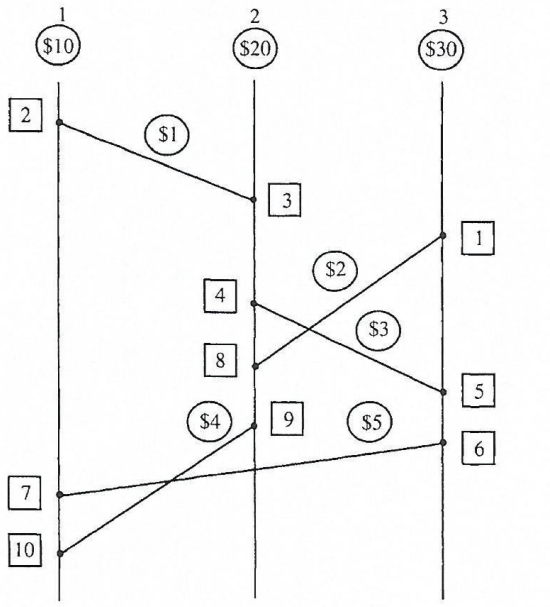
- Each process p_i determines the value of $money_i$ after all events with logical times less than or equal to t have been processed and before processing any event with logical time greater than t.
- For each channel incident to p_i, determine the amount of money in all the messages sent to p_i at logical times less than or equal to t but received by p_i at logical times strictly greater than t.

CountMoney – Using Logical Clocks

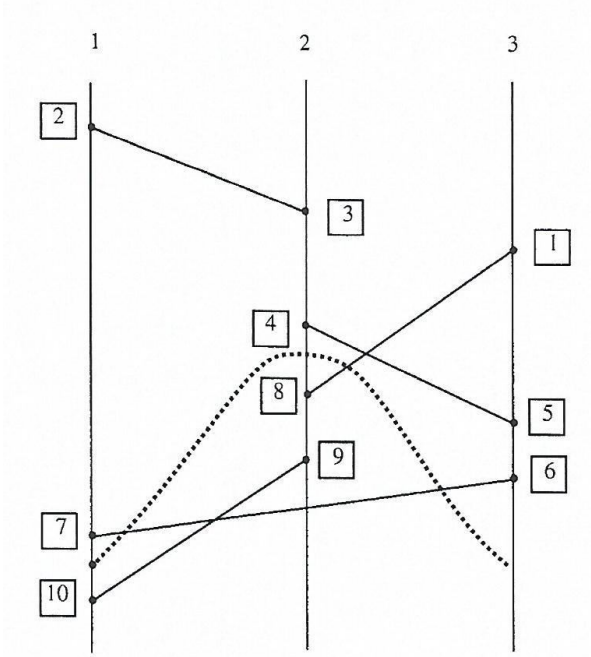
Each process p_i:

- Attaches the logical time of each send event to the message being sent, as a timestamp.
- Keeps track of the value of $money_i$ before and after the event of p_i most recently simulated.
- When it simulates the first event having a logical time strictly greater than t, returns the recorded value of $money_i$ before the execution of this event.
- To determine amount of money in incoming channel from p_j to p_i:
  - Starting with the first event with logical time exceeding t, p_i records messages coming in on the channel until it receives the first message for which the attached timestamp is less than or equal to t.
  - When a message arrives on the channel with timestamp strictly greater than t, p_i returns the sum of the amounts of money in the recorded messages.
  - The balance computed by p_i is the chosen value of $money_i$ and the sum of the values it determines for all its incoming channels.

Example



- P1: $10 - $1 + $5 = $14
- P2: $20 + $1 - $3 = $18
- P3: $30 - $2 + $3 - $5 = $26

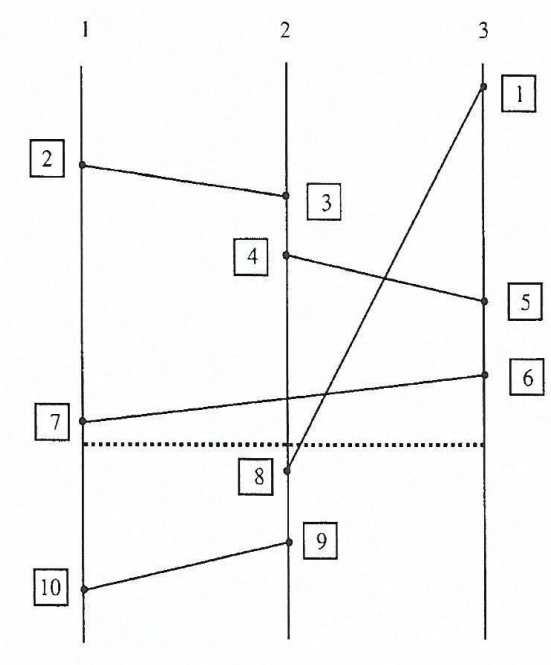


All channels are empty, except for the channel from P3 to P2 which contains $2.

Total = $14 + $18 + $26 + $2 = $60, as needed.

Correctness

- Consider any fixed execution a of *CountMoney.*
- There is another execution a′ (with the same events as a) which is similar with a to all processes, and in which all events occur in the order of their logical times.
- What the general strategy does is to "cut" execution a′ immediately after any events that have logical time = t and to record the money that is at all the processes and in all the channels, at this instant.



How to Determine Logical Time t

Problem

- Choosing an arbitrary t does not work, because that logical time might have already passed at some process before it begins executing the subroutine.

Possible Solution

- The processes might use a predetermined sequence $t_1, t_2, \ldots,$ of increasing logical times such that every t is $\leq t_i$ for some i, and attempt to complete the subroutines for all of them (in parallel).
- By broadcasting their results, the processes can determine the first t_i whose subroutine succeeds everywhere and use the results of that subroutine.

Global Snapshots

- Underlying graph: Arbitrary connected undirected graph G
- A global snapshot returns a global state of the system: a collection of states for all processes and channels that looks to the processes as if it was taken at the same instant everywhere in the system.

Global Snapshots – The LogicalTimeSnapshot Algorithm

How can we design an global snapshot algorithm that works using logical time?

Strategy

- Determine the state of each process after all events with logical times less than or equal to t and before all events with logical times greater than t.
- For each channel, determine the sequence of messages sent at logical times less than or equal to t but received at logical times strictly greater than t.

Global Snapshots: ChandyLamport Algorithm

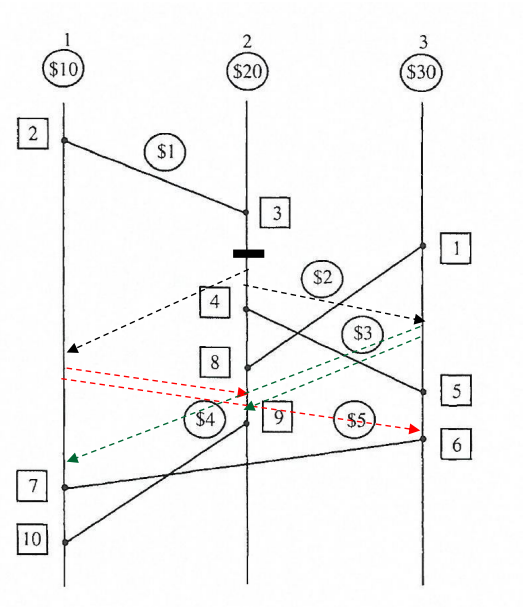
- A signal $snap_i$ may be received by each process p_i to begin taking a snapshot of the underlying algorithm A.
- Eventually, every process p_i will report a $response_i$ event.

- The various states reported by all the processes must constitute a global state of A which satisfies the following consistency property:
  - Consider any execution a of A. There should be another execution a' of A such that all of the following conditions hold:
    - a' is indistinguishable from a to each process p_i
    - a' begins with the prefix a_1 of a occurring before the first *snap* event
    - A' ends with the suffix a_2 of a occurring after the last *report* event
    - The returned state is exactly the global state after a prefix of a' that includes all of a_1 and none of a_2

The ChandyLamport Algorithm

- When a process p_i receives a $snap_i$ input:
  1. it records its current state
  2. it sends a *marker* message on each of its outgoing channels
- Then, p_i begins recording the messages arriving on each incoming channel until it encounters a *marker*
  - When this happens, p_i has recorded all the messages sent on that channel before the neighbor at the other end recorded its local state.
- If a process p_i receives a *marker* message before it has recorded its state:
  - p_i records its current state
  - sends out *marker* messages,
  - begins recording incoming messages.
  - The channel upon which it has just received the *marker* is recorded as empty.

The ChandyLamport Algorithm

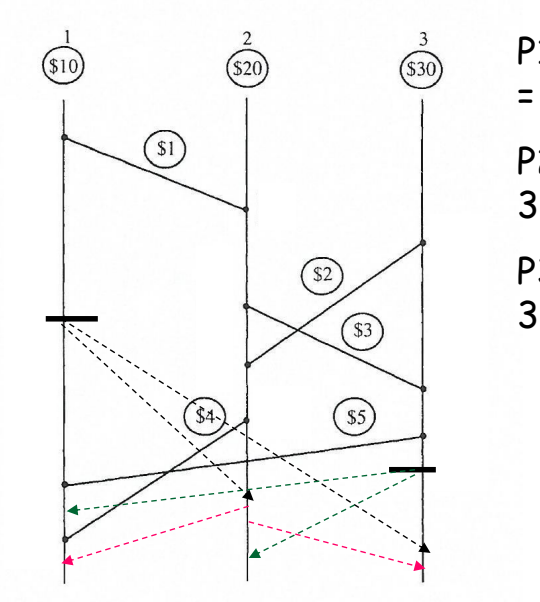


P1 reports 9

P2 reports 21 + 2

P3 reports 28

The ChandyLamport Algorithm

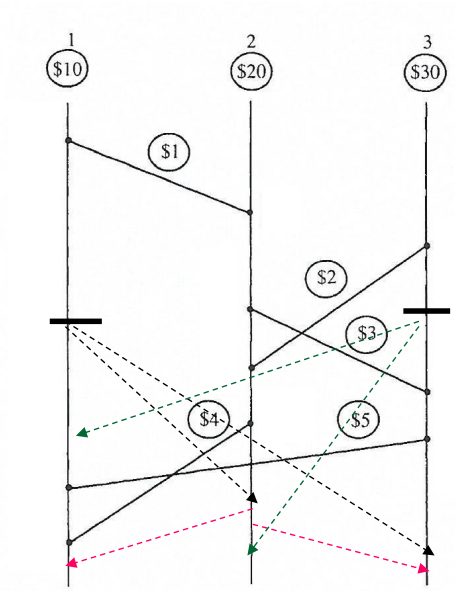


P1 reports 9 + 4 + 5 = 18

P2 reports 20 + 1 - 3 + 2 – 4 = 16

P3 reports 30 – 2 + 3 – 5 = 26

The ChandyLamport Algorithm



P1 reports 9 + 4 = 13

P2 reports 20 + 1 - 3 + 2 – 4 = 16

P3 reports 30 – 2 + 3 = 31

The ChandyLamport Algorithm – Why does every process report?

- As soon as a snap signal arrives at a process p_i, it records its state and sends out markers on all its channels.
- As soon as any other process p_j receives a marker on any channel, records its state and sends out markers on all of its channels (if haven't done so already)
- Thus, markers propagate to all processes and all processes record their state and finish collecting messages in all its incoming channels.

The ChandyLamport Algorithm – Why is the global state consistent?

- a_1: portion of a before the first snap event
- a_2: portion of a after last report event

Execution a'

- begins with a_1 and ends with a_2 and reorders the events of a between the first snap and the last report

The ChandyLamport Algorithm – Why is the global state consistent?

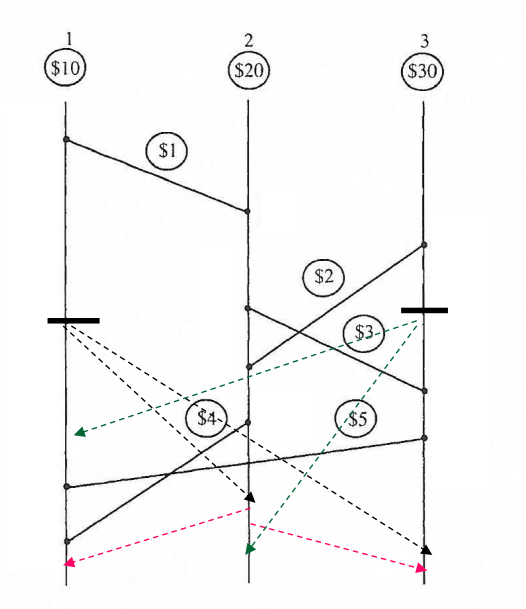
- For each process p_i, divide each event of a that takes place between a_1 and a_2 into two sets:
- S_1: those that precede the event $snap_i$ or $receive_{j,i}$(marker) of pi at which the state of p_i is recorded
- S_2: those that follow these event
- The reordering of a' places all S_1 events before all S_2 events while preserving the order of events of p_i and the order of each send with respect to the corresponding receive.
  - There is no send that follows the recording of p_i and whose receive precedes the recording of p_j
  - If a $send(m)_{i,j}$ follows the recording of the state of p_i, then m is placed in the $outbuffer_{i,j}$ after *marker*, so p_j receives marker first, so the state of p_j has already been recorded when m arrives.

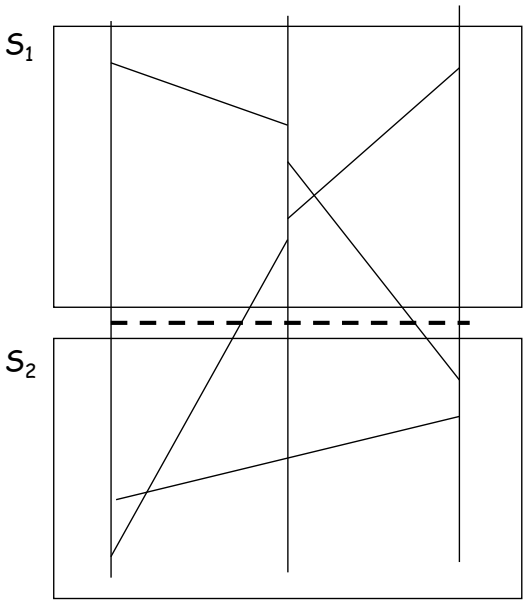
ChandyLamport Consistency

P1 reports 9 + 4 = 13

P2 reports 20 + 1 - 3 + 2 – 4 = 16

P3 reports 30 – 2 + 3 = 31





The ChandyLamport Algorithm – Why is the global state consistent?

- a_3: prefix of a' ending just after all the events in S_1. Then:
- The returned state of each p_i is exactly the state of p_i after a_3, because a_3 is defined to include exactly the events of p_i preceding the recording of the state of p_i.
- The channel recordings give exactly the messages that are in transit in the channels after a_3.
  - The messages in transit from p_i to p_j after a_3 are the messages whose $send(m)_{i,j}$ occur before the recording of the state of p_i and whose $receive(m)_{i,j}$ occur after the recording of the state of A_j.
  - These are the messages that arrive from p_i to p_j ahead of the marker and after p_j records its state.

Applications of ChandyLamport Algorithm

Distributed Debugging

- The designer of a DistAlg A can describe key properties of A by invariants about the global state of A.
- A debugger can allow A to run, obtaining consistent global snapshots from time to time and checking that the invariants are true for each snapshot.
  - The global state information can be transmitted to a single process, which can check the invariants locally.
  - Or, a distributed algorithm can be used, using the information returned by the snapshot algorithm as input data.
    - verify that a set of given parent pointers comprise a spanning tree rooted at a given node p_r

Applications of ChandyLamport Algorithm

Stable Property Detection

- If P is a stable property then: if P ever becomes true in an execution of A, then P remains true from that point onward
- Strategy for Stable Property Detection
  - obtain a consistent global state using a global snapshot algorithm and then to determine whether P is true or false of the returned global state
- The correctness conditions for a consistent global snapshot algorithm imply the following:
  1. If P is true based on the snapshot state, then P is also true at the global configuration of A just after the last *report* of the snapshot algorithm.
  2. If P is false based on the snapshot state, then P is also false for the global configuration of A just before the first *snap* of the snapshot algorithm.

Applications of ChandyLamport Algorithm

Deadlock Detection Problem

- Consider a send/receive network algorithm A in which each process p_i has local states that indicate that it is "waiting for" some subset of its neighboring processes (say, to release resources).
- When p_i is waiting for a nonempty set of neighbors, it is in a quiescent state and it cannot perform any local steps until it has received a message from each of the neighbors for which it is waiting.
- After p_i receives a message from any of the processes for which it is waiting, it continues to wait for the remaining processes.
- *A deadlock* at a configuration consists of a cycle of two or more processes, each waiting for the next in the cycle, with no messages en route from any process to its predecessor in the cycle.
- Deadlock is a stable property.