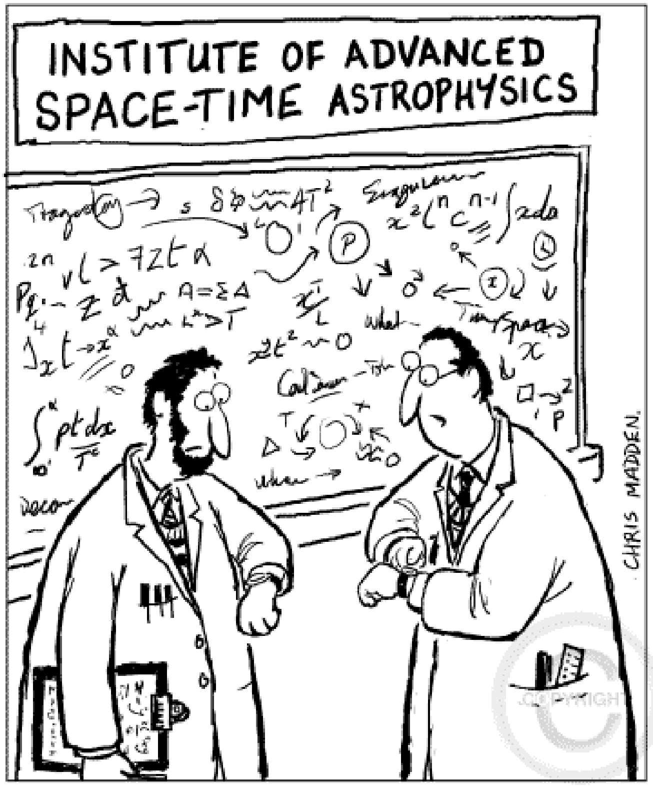
Time and Global State



I can never remember either.
Is it 'Spring back, fall forward'?

Today

- Clock synchronization
- Logical clocks
- Global state

Measuring time out in the world

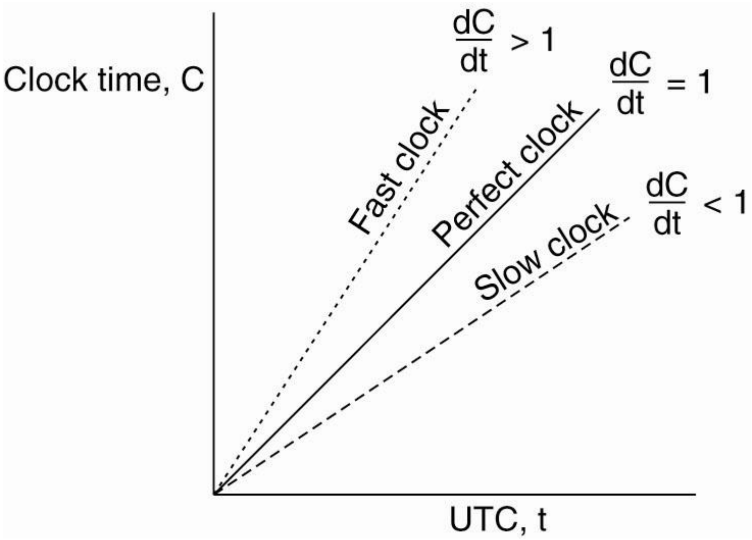
- Time has historically been measured astronomically
- A solar day
  - Time between two consecutive transits of the Sun
  - Transit of the Sun – when the Sun reaches its highest apparent point in the sky
- Solar second
  - 1/(24*3600) of a solar day
- But the period of earth rotation is not constant!
  - Slow down due to tidal friction and atmospheric drag
  - ~300 million years there were about 400 days per year

Atomic clocks

- Avoid problems with astronomical-based measurement
- Count transitions of Cesium 133 atom
  - A second – time to make 9,192,631,770 transitions (same as mean solar second when introduced)
  - TAI (International Atomic Time) the avg of several atomic clocks
- Universal Coordinated Time (UTC)
  - Problem – 86,400 TAI sec is 3msec < mean solar day today
  - Solution – add leap sec if TAI & solar time differ by 800 msec
- UTC seconds broadcasted on WWV shortwave radio (error > +/-10msec)

Physical clocks

- Hardware clock based on count of oscillations in a crystal
- Let's call this $C_p(t)$, the value of the clock on machine p when UTC is t
  - Ideally $C_p(t) = t$ for all p and all t – $C'_p(t) = dC/dt = 1$, but
  - Clocks drift (i.e. count time at different rates), so bound drift



Clock synchronization

- Two modes of synchronization
  - External – synchronize with a authoritative, external source of time; for a synchronization bound $D > 0$, and for a source of time S, $|S(t) - C_i(t)| < D$ *for* $i = 1..N$
  - Internal – synchronize the clock among them; $|C_i(t) - C_j(t)| < D$ *for* $i,j = 1..N$
- Synchronization in a synchronous system
  - Bounds are known for drift rates and maximum message transmission delays
  - Process sends time to another; if variation on transmission delay is u = max - min(max + min)/2 then t + (max + min)/2 gives a skew of at most u/2
  - But most distributed systems are asynchronous – no bounds on delays!

Clock synchronization

- External - Cristian's algorithm, ~NTP
  - Every machine asks a time server for the accurate time, gets *t* in a message
  - Set time to $t + T_{round}/2$, assuming equal split of transmission time
- Internal - Berkeley
  - Let a time server poll all machines periodically, calculate an average, and inform each host of to adjust its time
- NTP service
  - Provided by network of servers with primary servers connected to time source, secondary servers to …
  - NTP servers synchronize with others via multicast, procedure call or symmetric mode

Abstract model of a distributed system

- A distributed system – a collection P of N processes p_i
- Processes communicate (only) by sending messages
- Each process p_i has a state s_i which, in general, transform when executes
- Processes execute a series of actions – send/receive, or transform its state – an event is the occurrence of a single action
- Events within a process can be place in single, total ordering, a relationship between events denoted by $\rightarrow_i$
- History of a process – the series of events that take place within it

What happened before

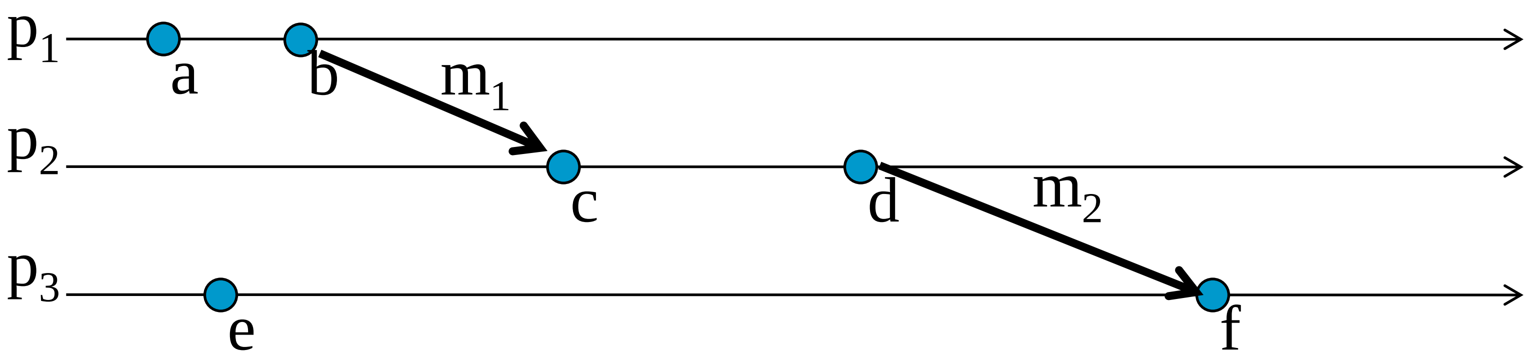
- Without perfectly synchronized clocks, how can we order events in a distributed system?
  - Events in a single process occur in the order the process observes them
  - When a message is sent between two processes, the sending occurs before the receiving
- The partial ordering that results from this – *happened-before* relation
  - *If e and e' are two events in the same process, and $e \rightarrow_i e'$ (e comes before e'), then $e \rightarrow e'$*
  - *If e is the sending of a message, and e' is the receipt of that message, then $e \rightarrow e'$*
  - *If $e \rightarrow e'$ and $e' \rightarrow e''$, then $e \rightarrow e''$*

Lamport clocks

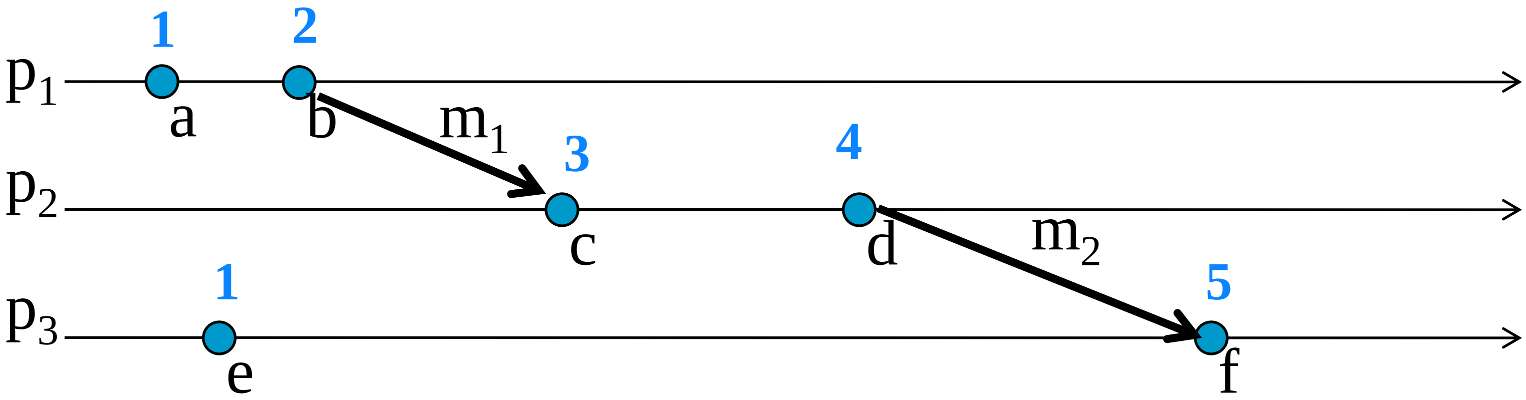
- To maintain a global view on the system's behavior that is consistent with the happened before relation
- Lamport clock
  - A monotonically increasing software counter
  - Each process p_i has its own Lamport clock L_i that uses to timestamp its events ($L_i(e)$ is the timestamp of e)
- To capture the happened-before relation
  - LC_1: If e and e' are events in the same process, and $e \rightarrow e'$, then $L_i(e) < L_i(e')$; i.e. L_i is incremented before event
  - LC_2: If a processes sends a message
    - It piggybacks with it the value $t = L_i$
    - On receiving a message, a process p_j computes $L_j := \max(L_j, t)$ and then applies LC_1
- To create a total order we can take into account the processes ids (practical but without physical meaning)

Lamport clocks

Sequence of events

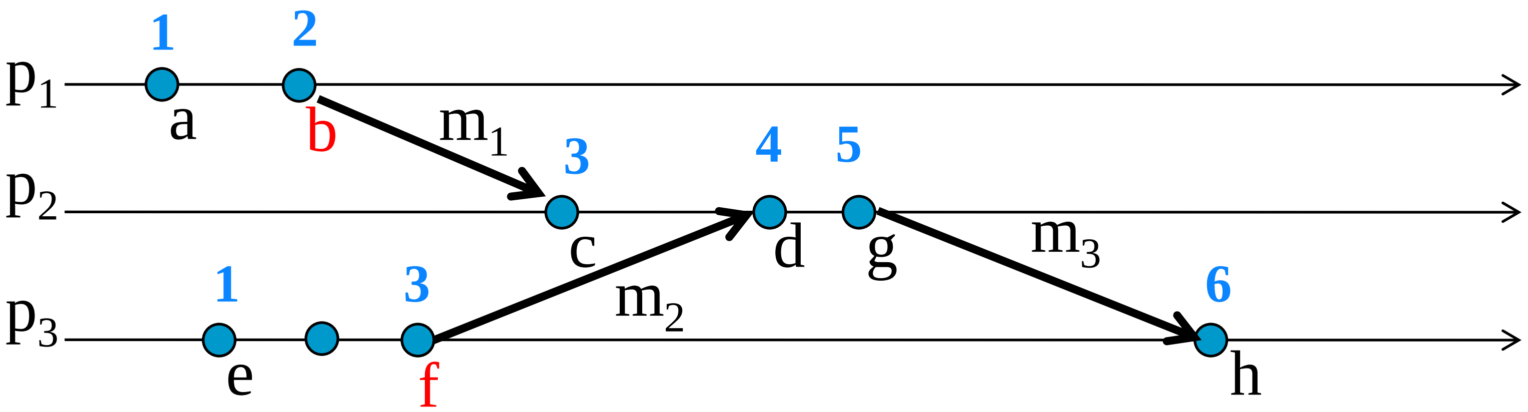


Lamport clocks



Problem with Lamport clocks

- Observation: Lamport clocks do not guarantee that if *L(e) < L(e'), e causally preceded e':*
  - E.g. L(b) < L(f), but !(b → f)

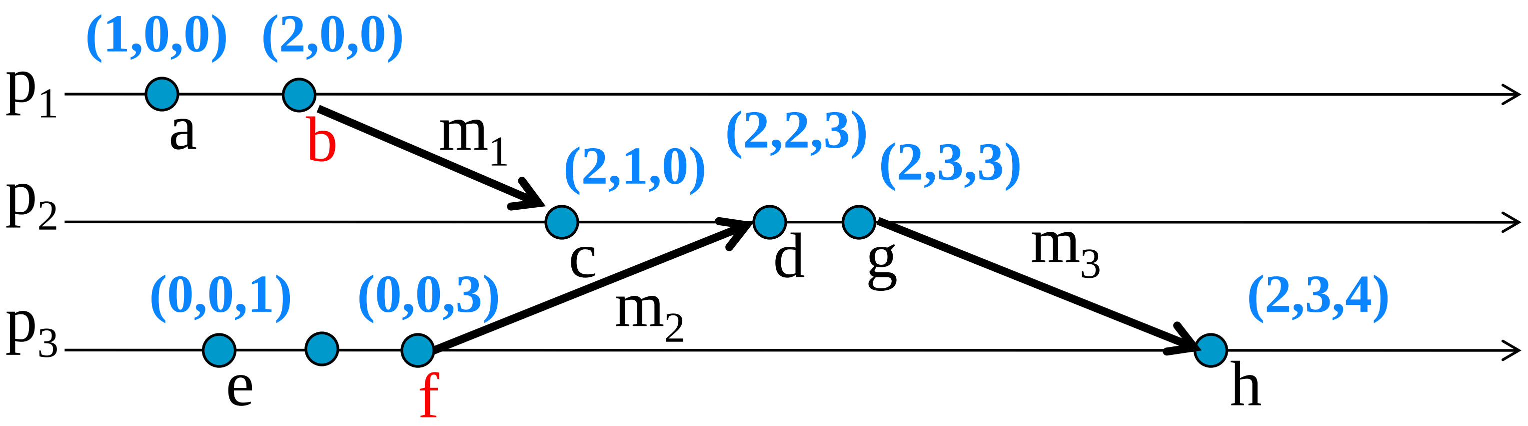


Vector clocks

- Vector clock – an array of N integers for a system of N processes
- Each process P_i keeps its own vector V_i; initially $V_i[j] = 0$ for $i,j = 1..N$
- Before executing an event P_i - $V_i[\,i\,] := V_i[\,i\,] + 1$
- When P_i sends a message m to P_j,
  - It executes the previous step
  - It sets *m*'s (vector) timestamp *ts (m)* equal to V_i
- Upon receipt of a message m
  - P_j adjusts its own vector by setting $V_j[k] := \max\{V_j[k], ts(m)[k]\}$ for each *k* (it "merges" both vectors)
  - It executes first step

Comparing vector clocks

- V = V' iff V[j] = V'[j] for j = 1..N
- V ≤ V' iff V[j] ≤ V'[j] for j = 1..N
- V < V' iff V ≤ V' ∧ V != V'
- Two events e and e' are concurrent (e || e') if neiher V(e) ≤ V(e') nor V(e) ≥ V(e')
- Question: What does $V_i[j] = k$ mean?

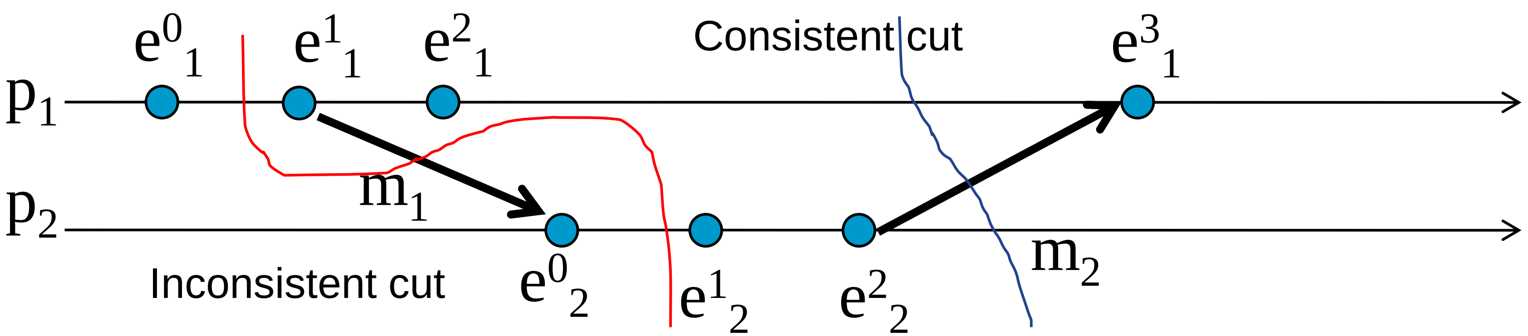


Global states

- Checking if a property of a distributed system is true
- Examples
  - Distributed garbage collection – is an object garbage? Is there a reference to it somewhere?
  - Distributed deadlock situation
  - Distributed debugging
- Detecting a condition like any of these is the same as evaluating a global state predicate
- Global state, mathematically any set of local states can be put together to form it $S = (s_1, s_2, \ldots, s_N)$
  - Which of those is meaningful?

Global states

- Cut – subset of its global history
  - $C = h^{c1}_1 \cup h^{c2}_2 \cup \ldots h^{cN}_N$
  - Set of events $\{e^{ci}_i : I = 1..N\}$ is the frontier of the cut
- A cut is consistent if, for each event it contains, it also contains all the events that happened- before it
  - A *consistent global state* corresponds to a consistent cut
  - A *linearization* or *consistent run* – an ordering of events in a global history that is consistent with happened-before
  - A state S' is *reachable* from S if there is a linearization that passes through S and then S'



Chandy and Lamport's snapshots

- Chandy & Lamport's algorithm
  - Useful to determine the global state of a distributed system
  - Records states locally to a process, gathering is extra
- It assumes that
  - Neither channels nor processes fail (comm. is reliable)
  - Channels are unidirectional and FIFO
  - Graph of processes and channels is strongly connected
  - Any process many initiate a global snapshot at any time
  - Processes can continue with what they were doing while the snapshot is being taken

Chandy and Lamport's algorithm idea

- Basic idea – each process records its state and, for every channel, the set of messages sent to it
- Use a special message – marker – with a dual role
  - Prompt receiver to save its own state
  - Help determine which message to include in the channel state
- Defined by two rules
  - Marker receiving rule – obligates a process to save its state and help defined the state of the channel
  - Marker sending rule – obligates a process to send a marker after having recorded their state and before sending anything else

Chandy and Lamport's algorithm

- Algorithm is defined by two rules
  - *Marker receiving rule for process p_i*
    - On p_i's receipt of a marker message over channel c:

      if (p_i has not yet recorded its state) it

      records its process state now

      records the state of c as the empty state

      turns on recording of messages arriving over other incoming channels

      else

      p_i records the state of c as the set of messages it has received over c since it saved its state

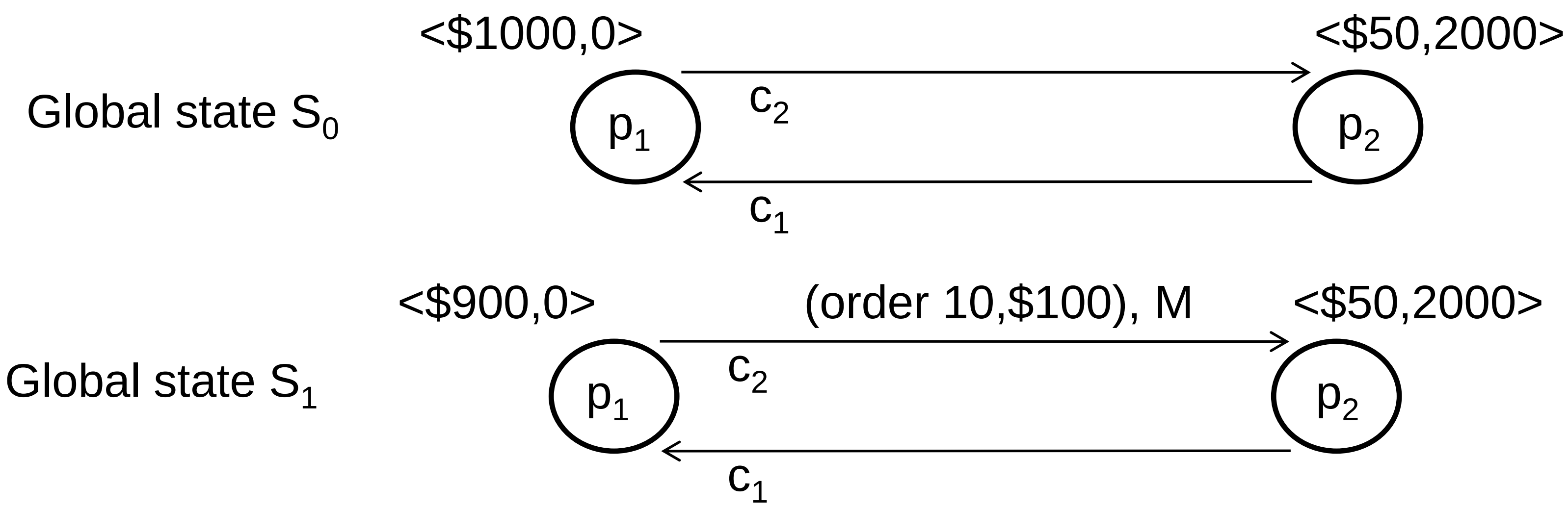
      end if
  - *Marker sending rule for process pi*
    - After p_i has recorded its state, for ach outgoing channel c:

      p_i sends one marker message over c

      (before it sends any other message over c)

Taking a snapshot

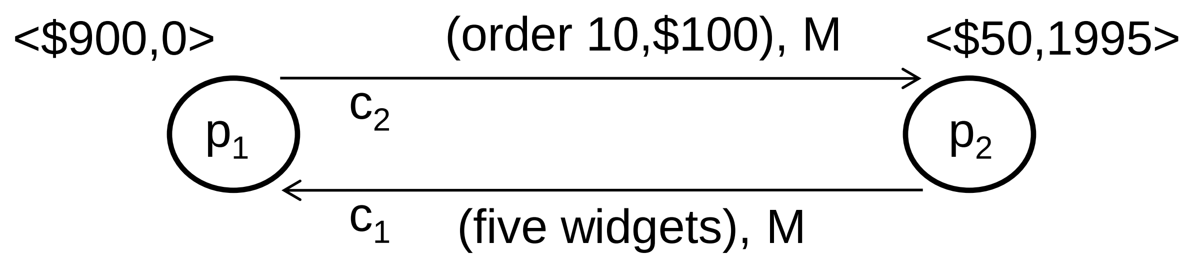
- Two processes trading in widgets at a rate of \$10 per piece
- p_2 has already received and order of 5 widgets
- p_1 records its state in S_0, emits marker and follows with another request



Taking a snapshot

- Before p_2 gets the marker it sends the 5 widgets
- Then gets the marker and record its state (<\$50, 1995>) and that of channel c_2 as empty
- Then sends a marker on c_1

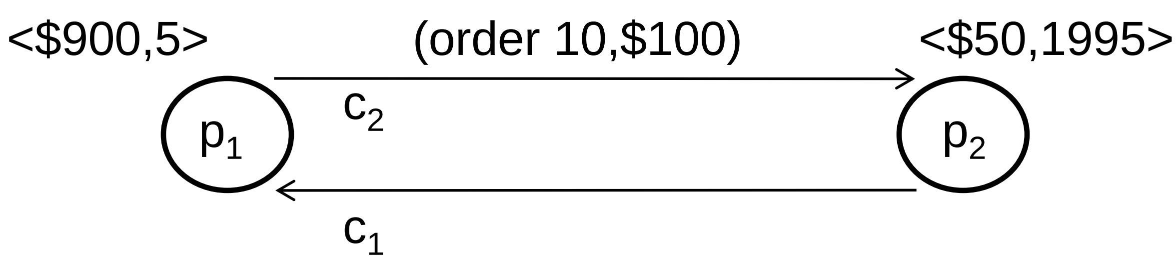
Global state S_2



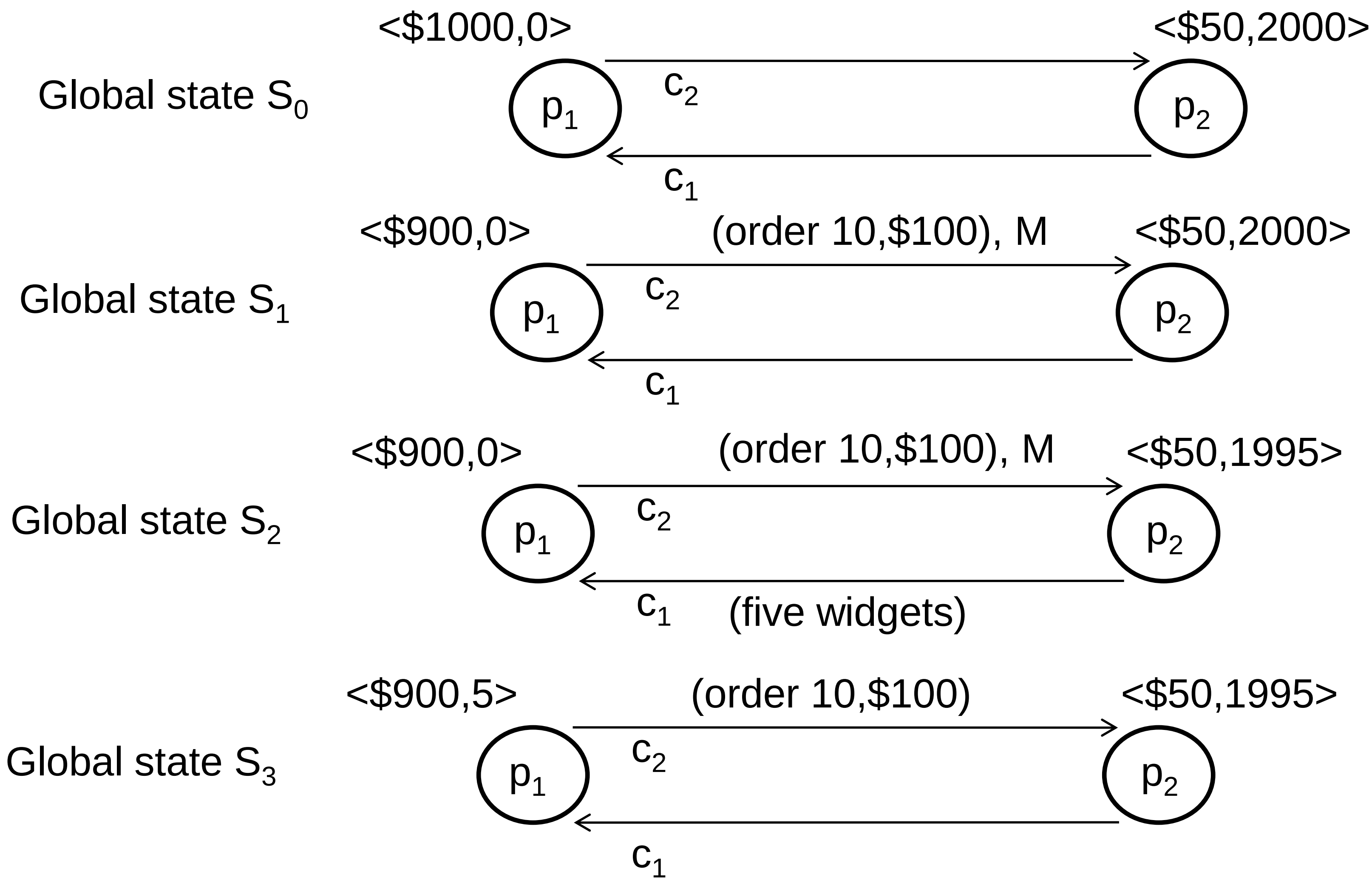
Taking a snapshot

- When p_1 gets the marker, it records the state of c_1
- Final recorded state is p_1: <\$1000, 0>, p_2: <\$50, 1995>, c_1: <(five widgets)>, c_2: <>
- State is consistent
- Note that it differs from all global states the system went through

Global state S_3



Taking a snapshot



Final recorded state is p_1: <$1000, 0>, p_2: <$50, 1995>, c_1: <(five widgets)>, c_2: <>