The UNIVERSITY *of* EDINBURGH
SCHOOL *of* INFORMATICS

CS4/MSc

Distributed Systems

Björn Franke

bfranke@inf.ed.ac.uk

Room 2414

(Lecture 9: Global States, 31st October 2005)

Global States

There are times when we need to know whether a condition is true for a distributed system and the condition cannot be judged *locally* but requires *global* knowledge of the state of the system. For example,

- distributed garbage collection
- distributed deadlock detection
- distributed termination detection
- distributed debugging

These problems motivate the need for ways of observing (and reasoning about) global states.

Global States without Global Time

If we had some absolute global time (perfectly synchronised clocks) we could

- obtain a global state by getting each process to record its local state at an agreed time;
- obtain a sequence of global states from suitably interleaved sequences of local states.

Instead we seek to form a global state from local states recorded at different times.

At a process P_i we can observe a sequence of events $\langle e_i^0, e_i^1, e_i^2, \ldots \rangle$, termed a **history** and denoted h_i.

Recall that each event either updates local state or corresponds to sending or receiving a message.

Histories and Cuts

If, in a system of N processes, we have a history h_i for each process P_i, $i = 1, \ldots, N$, we can form a **global history**:

$$H = h_1 \cup h_2 \cup \ldots \cup h_N$$

In order to construct a global state—a consistent set of local states—we form a **cut**, which is a truncation of each local history.

$$C = h_1^{c_1} \cup h_2^{c_2} \cup \ldots \cup h_N^{c_N}$$

where $h_i^k = \langle e_i^0, e_i^1, \ldots, e_i^k \rangle$.

A global state S is then the set of local states s_i:

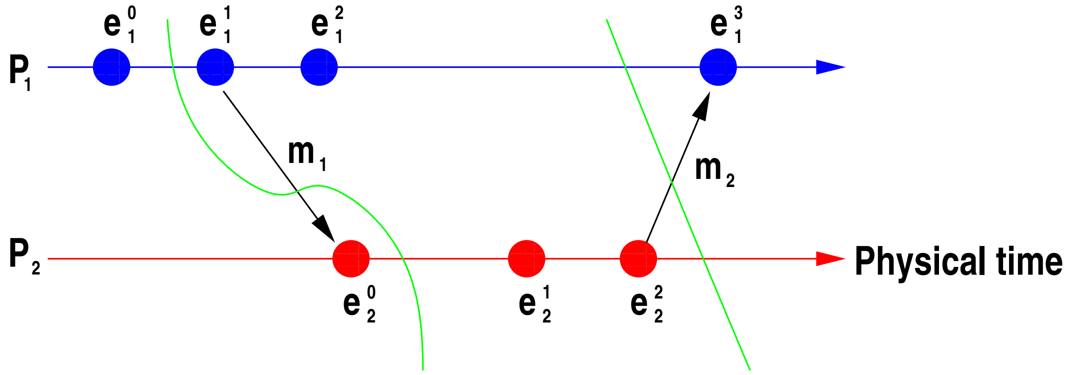
$$S = \{ s_i \mid i = 1, \ldots, N \}$$

where the state s_i of P_i corresponds to the state of P_i immediately after the last event before the cut: $e_i^{c_i}$.

The set of events $\{ e_i^{c_i} \mid i = 1, \ldots N \}$ is called the **frontier** of the cut.

Consistent Cuts and States

A cut is **consistent** if, for each event, it also contains all the events that *happened-before* that event, i.e. for all $e \in C$, $f \rightarrow e \Rightarrow f \in C$.



The cut on the right is consistent whilst the cut on the left is inconsistent.

A **consistent global state** is one that corresponds to a consistent cut.

Runs and Linearizations

- The execution of a distributed system can be regarded as a sequence of global states: $S_0 \rightarrow S_1 \rightarrow S_2 \rightarrow \ldots$
- Each transition corresponds to **one** event in **one** process: if two events occur simultaneously they must be concurrent and we can place them in either order.
- A **run** is a total ordering of all events in a global history that is consistent with each local history's happened-before ordering.
- A **linearization** or **consistent run** is an ordering of the events in a global history that is consistent with the happened-before relation on H.
- Linearizations only pass through consistent global states (not necessarily true for runs).
- We say that a state S' is **reachable** from a state S if there is a linearization that passes through S then S'.

Stability, Safety and Liveness

Once we have established notions of global states and linearizations we can use them to reason about the behaviour of systems.

- When we want to detect whether a condition holds for the whole system we use a **global state predicate**, i.e. a function which maps from the set of global states to {`true, false`}.
- A condition is **stable** if once the corresponding predicate becomes `true` it remains `true` in all reachable states. Otherwise a condition is termed **unstable**.
- If a predicate α expresses some undesirable property, **safety** of the system with respect to α means that α evaluates to `false` for all states reachable from S_0.
- If a predicate β expresses some desirable property, **liveness** of the system with respect to β means that, for any linearization starting in state S_0, β evaluates to `true` for some state S^* in the linearization.

Snapshot Algorithms

- Snapshot algorithms are intended to record the states of a set of processes and their interconnecting channels in a consistent manner, i.e. as if the recorded set of local states had occurred at the same time although this may not in fact be the case.
- They have applications in banking, deadlock detection, termination detection and distributed debugging.
- Several snapshot algorithms have been published, some using logical clocks (histories are divided into events before a chosen logical time t and events after t).
- Perhaps the most well-known snapshot algorithm is due to Chandy and Lamport, and instead of logical clocks it makes use of special purpose **marker** messages passed between processes, to indicate when the snapshot should be taken.

Assumptions

We assume that the system consists of N distinct processes, P_i, $i = 1, \ldots, N$. In addition the algorithm makes some additional assumptions about the system infrastructure.

- neither channels nor processes fail; communication is reliable so that every message sent is eventually received intact, exactly once.
- channels are unidirectional and provide FIFO-ordered message delivery;
- the graph of processes and channels is strongly connected so that there is a path between any two processes;
- any process may initiate a global snapshot at any time (but in this case marker messages should be distinguishable).
- the processes may continue their execution and send and receive normal messages whilst the snapshot takes place.

Chandy and Lamport's *Snapshot Algorithm*

- Assume that process P_i initiates the snapshot, that is it spontaneously behaves as if it had received a marker message.
- P_i records its local state: truncating its history and generating a cut. This will be consistent from P_i viewpoint in the sense that for any event e in the truncated history, any event e' that happened-before e must also be in the history.
- P_i immediately sends a marker message on each of its outgoing channels. These messages act as triggers to the receiving processes to record their local state if they have not already done so. They also divide messages on the channel into those before the cut (in front of the marker) and those after (behind the marker.
- P_i also keeps track of all arrivals on each of its incoming channels until a marker is received on that channel. The event of receiving a message before the marker is added to the cut, because the event of sending that message must have occurred before the sending process recorded its local state.

The algorithm

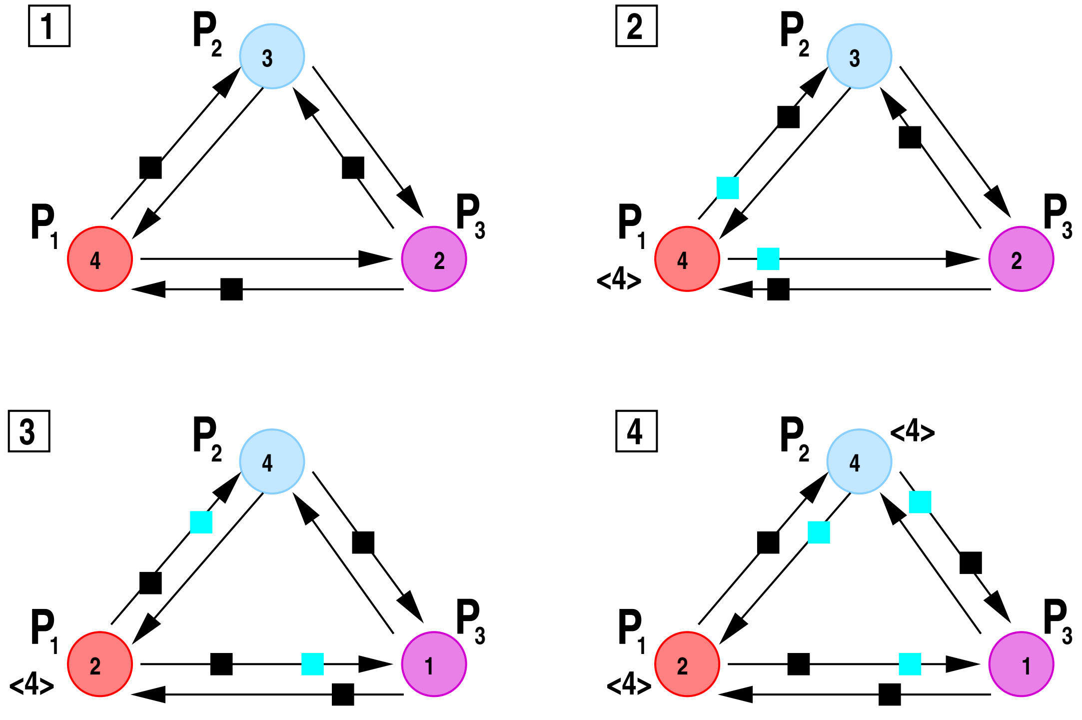
Marker receiving rule for process P_j

```
When P_j receives a marker message over channel c:
  if (P_j has not yet recorded its state) it
    records its process state now;
    records the state of c as the empty set;
    starts recording messages arriving over other channels;
  else
    records the state of c as the set of messages it has received
    over c since it saved its state.
  end if
```

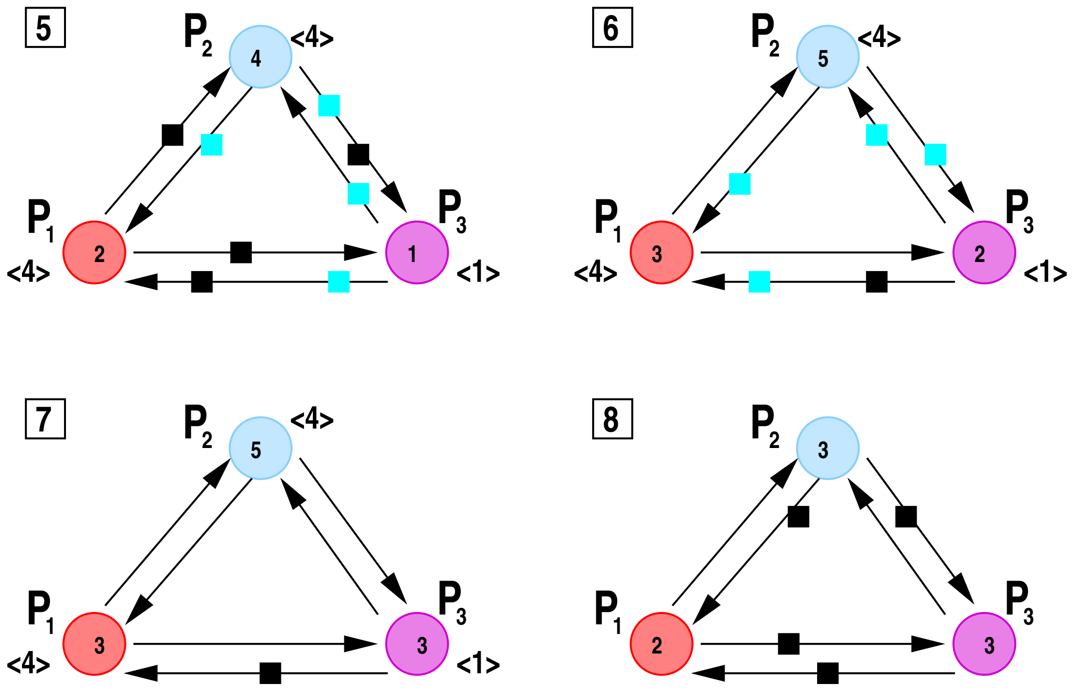
Marker sending rule for process P_j

```
After P_j has recorded its state, for each outgoing channel c:
    P_j sends a marker message over c
    (before it sends any other message over c).
```

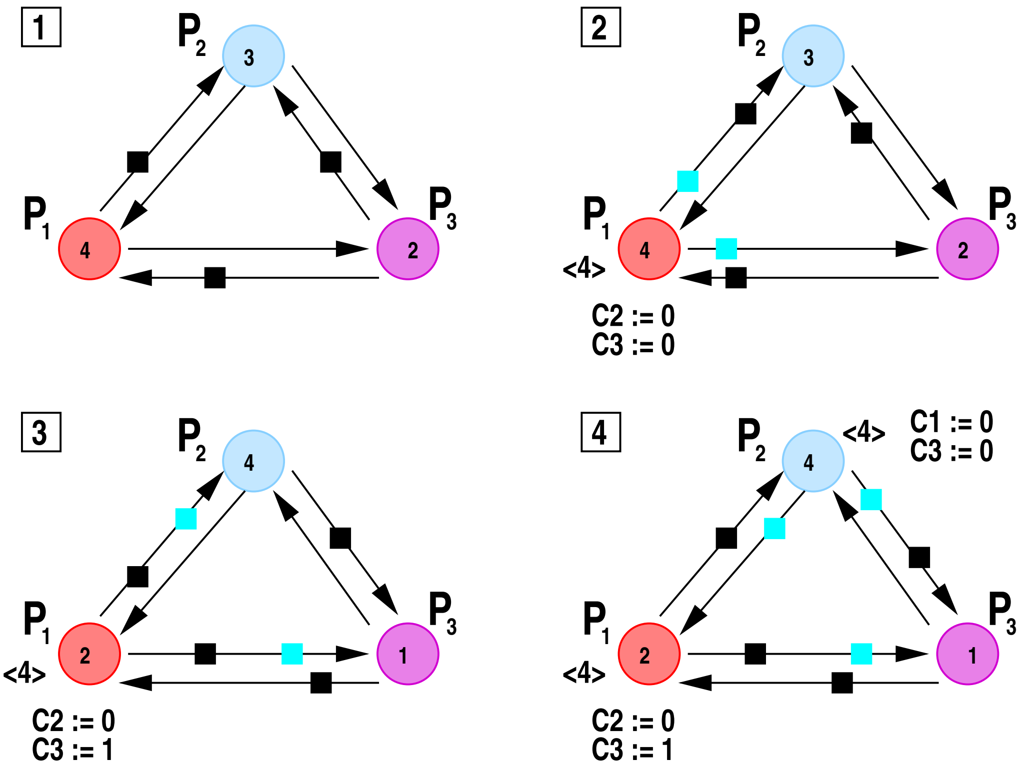
Example – recording states



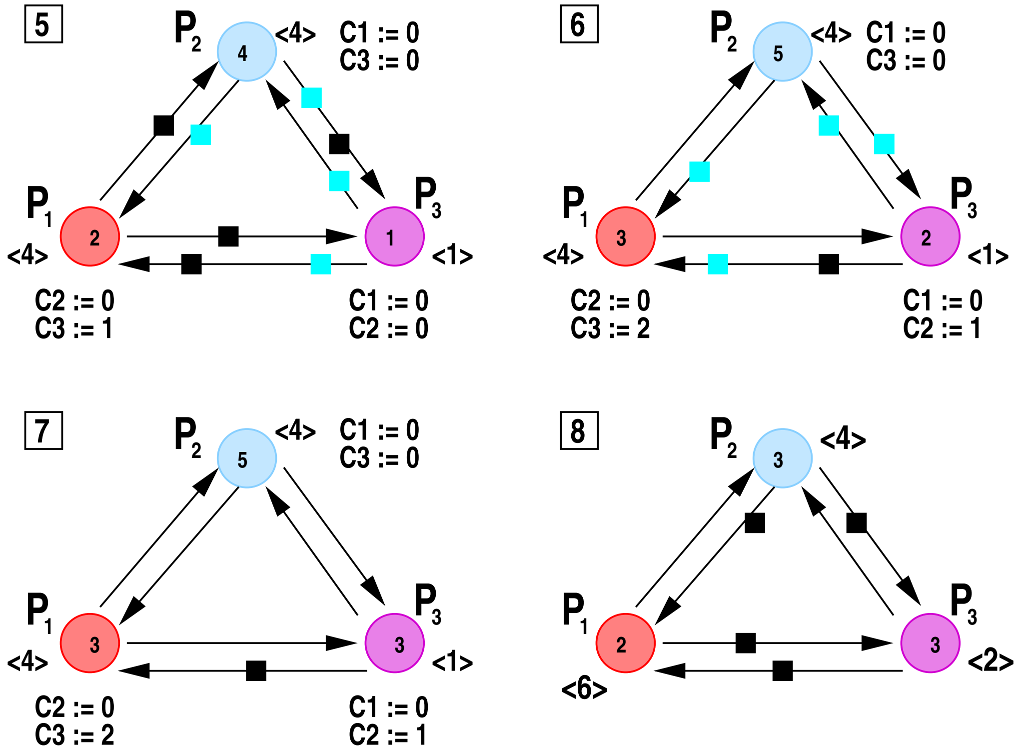
Example (cont.)



Example – taking channels into account



Example (cont.)



Distributed Debugging

- Chandy and Lamport's algorithm is useful for detecting stable predicates, for example such as detecting distributed termination.
- For distributed debugging we are generally more concerned with transitory conditions captured by non-stable predicates.
- We focus on *post hoc* analysis of the history of an execution to check whether a required safety property was violated or not.
- In some cases it is sufficient to detect that some predicate ϕ possibly held; in other cases we need to know that the predicate ϕ definitely held:

  possibly ϕ**:** there is a consistent global state S through which a linearisation passes such that $\phi(S) = \texttt{true}$.

  definitely ϕ**:** for all linearisations L there is a consistent global state S through which L passes such that $\phi(S) = \texttt{true}$.

- Marzullo and Neiger's algorithm is a centralised algorithm which relies on an external *monitor* process that collects state information from the distributed processes as they run.

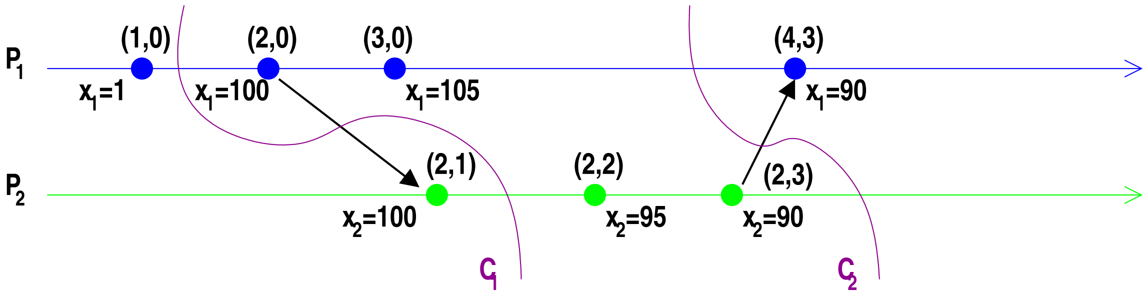
Collecting states

- All the observed processes send their initial states to the monitor, and then periodically send *state messages* (when it changes with respect to ϕ).
- As well as the state these messages include the vector clock of the process.
- The monitor maintains a separate queue for each observed process, in which messages are kept in sending order.
- These messages are used to construct globally consistent states (GCS): if s_i is a state of P_i and $S = (s_1, s_2, \ldots, s_N)$ is a global state, S is a GCS if and only if

$$V(s_i)[i] \geq V(s_j)[i] \quad \text{for } i, j = 1, 2, \ldots, N$$

- The GCS can be represented in a lattice which has one dimension for each process; levels of the lattice represent steps through a linearisation; transitions within the lattice capture the reachability relation.

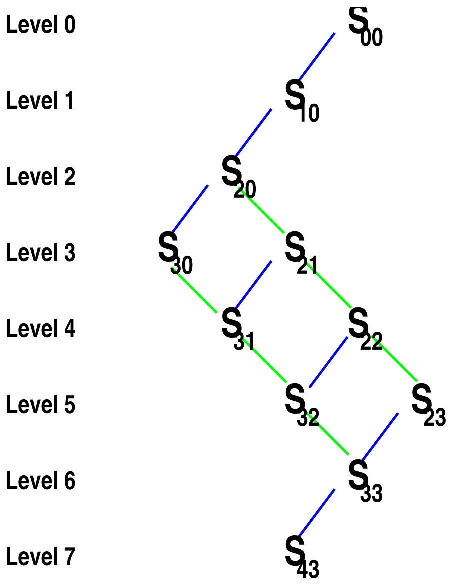
Example



$$\phi = (|x_i - x_j| \leq 50) \quad \text{for } i, j = 1, 2$$

- Large adjustments to the value of x_i triggers a message to process p_j.
- Cut C_1 would indicate that $\phi = \texttt{false}$ but this cut is inconsistent.
- Cut C_2 is a consistent cut; it shows values $x_1 = 105$ and $x_2 = 90$ — well within the constraint.

The lattice of GCS



- Using the vector clocks the monitor can construct the set of reachable states considering allowable interleavings of the events at each process.
- Global state S_{ij} is the slate when i events have occurred at process 1 and j events have occurred at process 2.
- A linearisation traverses the lattice from one level to the next using the reachable transitions which make up the lattice structure.

Evaluating ***Possibly*** ϕ

- The monitor traverses the lattice of reachable states, examining each state within a level before moving on to the next.
- The algorithm terminates as soon as a state satifying ϕ is found.

```
L:=0;
States:={S_01, S_02, ..., S_0N};
while ( phi(S) = False for all S in States)
   L:=L + 1;
   Reachable:={S' | level(S') = L and S' reachable from some S_Li}
   States:=Reachable
end while
output "possibly phi"
```

Evaluating ***Definitely*** ϕ

- Evaluating definitely ϕ is more strigent: for each linearization it is necessary to find a GCS satisfying ϕ

```
L:=0;
if (phi(S_01, S_02, ..., S_0N)) then States = emptyset
else States:={ S_01, S_02, ..., S_0N};
while (States not empty)
  L:=L + 1;
  Reachable:={S' | level(S') = L and S' reachable from some S_Li}
  States:={S in Reachable | phi(S) = false}
end while
output "definitely phi"
```