Cuts, Global Snapshots and Termination

Carlos Baquero
Distributed Systems Group
Universidade do Minho

MAPI 2007

Plan

- Cuts and Global States
- Global Snapshot algorithms
- Termination detection

Cuts

Carlos Baquero
Distributed Systems Group
Universidade do Minho

Cuts and Global States

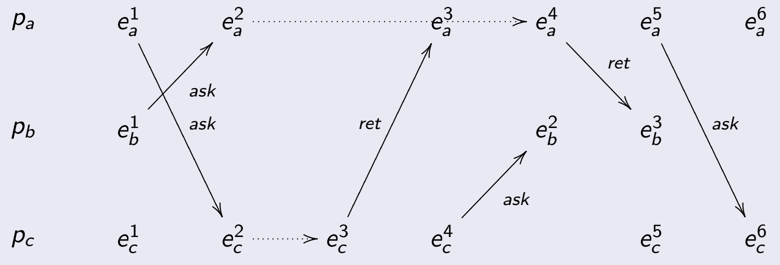
- σ_i^k is the local state of p_i right after event e_i^k.
- The global state is a n-tupple of local states. $\Sigma = \langle \sigma_1, \ldots, \sigma_n \rangle$.
- A cut is a subset C of global history H containing an initial prefix of each local history.
- $C = h_1^{c_1} \cup \cdots \cup h_n^{c_n}$
- The cut frontier is the set of last events $\{e_1^{c_1}, \ldots, e_n^{c_n}\}$ included in the cut $\langle c_1, \ldots, c_n \rangle$. The corresponding global state is $\langle \sigma_1^{c_1}, \ldots, \sigma_n^{c_n} \rangle$.
- A run is a total order on the events H that is consistent with each local history.

Cuts

Consider a system with blocking remote procedure calls (or call it remote object invocations). Such a system is prone to deadlocks if a wait-for graph of dependencies is formed. This can be detected by collecting and inspecting the global state Σ looking for cycles.

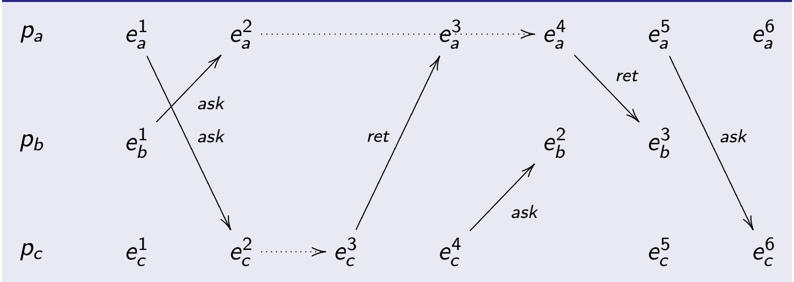
An omniscent observer can check that the following execution has no such cycles.

Execution with blocking RPCs



Cuts

Execution with blocking RPCs



Consider an extra monitoring process p_m that asks each process for its state. p_m can collect a cut with frontier:

$\{e_a^3, e_b^2, e_c^6\}$

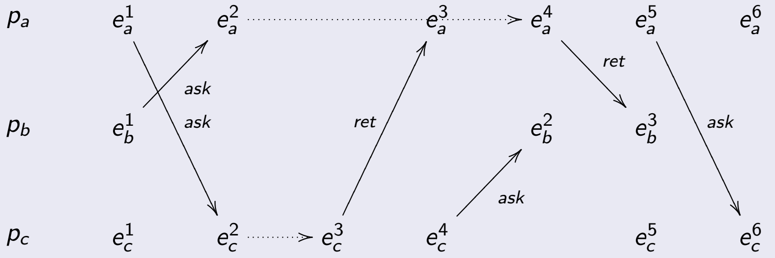
processes respond with indication of received calls waiting response.

$\{b$ **waitsfor** a, c **waitsfor** b, a **waitsfor** $c\}$

here we witness a *ghost deadlock* that is fictitious.

Cuts

Execution with blocking RPCs



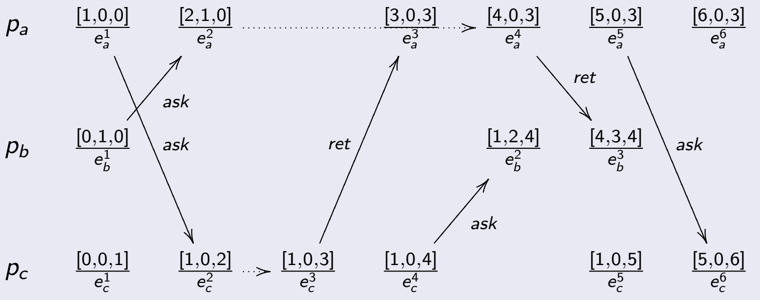
A cut with frontier $\{e_a^3, e_b^2, e_c^6\}$ makes no sense and could never occur since it includes a message received in e_c^6 that is not sent up to e_a^3. This cut is inconsistent (check rubber lines analogy).

Consistent Cut

A cut C is consistent if for all events e, e':
$e \in C$ and $e' \rightarrow e$ implies $e' \in C$.

Cuts

Execution with blocking RPCs



Inconsistent cut $\{e_a^3, e_b^2, e_c^6\}$ has clocks $\langle [3,0,3], [1,2,4], [5,0,6] \rangle$. Where process p_c knowns that p_a is at event 5 while p_a only knowns up to event 3.

Consistent Cut

A cut $\langle c_1, \ldots, c_n \rangle$ is consistent iff: $\forall i, j : \mathcal{V}(e_i^{c_i})[i] \geq \mathcal{V}(e_j^{c_j})[i]$.

Cuts

Carlos Baquero Distributed Systems Group Universidade do Minho

Consistent Cut

A cut $\langle c_1, \ldots, c_n \rangle$ is consistent iff: $\forall i, j : \mathcal{V}(e_i^{c_i})[i] \geq \mathcal{V}(e_j^{c_j})[i]$.

One approach to find a consistent cut would be p_m asking in cycle for snapshots, tagged with vector clocks, until eventually a consistent cut is found. Such cut would not include channel state.
Next we consider distributed snapshot algorithms that take into account channel state and derive consistent cuts. For simplicity they assume reliable FIFO channels but can be addapted for non FIFO settings.

Snapshot Protocols

Carlos Baquero
Distributed Systems Group
Universidade do Minho

- Set o processes $\{p_1, \ldots, p_n\}$.
- Process p_i:
  - IN_i depicts the processes that have channels to p_i.
  - $OUTi$ depicts the processes to which p_i has channels.
- In each snapshot execution p_i records local state σ_i and the state of incoming channels, $\chi_{j,i}$, for all $p_j \in IN_i$.

Snapshot Protocols

With global real time clock $\mathcal{T}$

- All messages are tagged with the time of the send event.
- Process p_m sends *take snapshot at time t* message to all processes. (t should be far enough in the future.)
- When clock reaches t each p_i:
  - Records σ_i.
  - Sends *sweep* message on all OUT_i (these messages have timestamp greather than t).
  - Starts recording messages from each process in IN_i.
- When a message with timestamp greather than t is received from process p_j, stop recording messages for this channel and produce $\chi_{j,i}$.

Snapshot Protocols

With global real time clock $\mathcal{T}$

- Since we have FIFO, all messages in $\chi_{j,i}$ collected by p_i contains the messages sent by p_j before time t and received by p_i after t. The protocol sweeps the channel.
- The *sweep* message enshures liveness, since this messages is eventually delivered and will carry a timestamp greather than t that concludes the recording for the respective channel.
- The protocol collects a consistent snapshot that did occur; a nice property of real time.
- To collect a consistent snapshot it suffices a timestamp mechanism consistent with causality. Real time is just one realization of this.
- One can try to substitute real time for a logical clock.

Snapshot Protocols

With logical time

- We consider that p_m can establish a logical time ω that is far enough in the future.
- Process p_m sends *take snapshot at time* ω message to all processes.
- When clock reaches ω each p_i:
  - Records σ_i.
  - Sends *sweep* message on all OUT_i (these messages have timestamp greather than ω).
  - Starts recording messages from each process in IN_i.
- When a message with timestamp greather than ω is received from process p_j, stop recording messages for this channel and produce $\chi_{j,i}$.

Now we will try to avoid the explicit logical time.

Snapshot Protocols
Chandy-Lamport

- Process p_m sends a *take snapshot* message to itself.
- When p_i receives the first *take snapshot* message and p_j is the sender:
  - Record σ_i.
  - Relay *take snapshot* to all OUT_i.
  - Set $\chi_{j,i}$ to $\langle\rangle$ and start recording all IN_i channels.
- When p_i receives the a subsequent *take snapshot* message and p_s is the sender. Stop recording messages from p_s and establish $\chi_{s,i}$ as the recorded messages.

Notice that *take snapshot* traverses each channel exactly once. When a process receives the message in all channels its snapshot is complete and it can send it to the initiator.

Carlos Baquero
Distributed Systems Group
Universidade do Minho

A *diffusing algorithm* is one where activity starts at a node (e.g. after an external input) and diffuses along the network. Dijkstra Scholten termination detection is suitable for these cases.

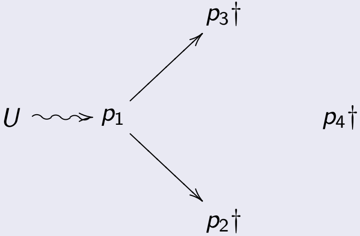
Termination can be formalized as:

If, sometime after an input occurs at some process p_i, the monitored algorithm ever reaches a quiescent global state, then eventually a $done_i$ is produced at node p_i.

We now consider the *AsynchSpanningTree* algorithm and add *Ack* messages. *search* messages are used and each process (non root) designates its first received contact as parent. Any subsequent *search* messages receive *Acks*, but not the first contact.
Acks are only reported to the first contact (designated parent) once the state of the process is quiescent and all outgoing messages have been aknowledged. After reporting the node forgets all protocol state, and can participate again in a tree construction.
The protocol allows the spanning tree to grow and shrink, until they shrink into the root node and the whole algorithm terminates.

Carlos Baquero
Distributed Systems Group
Universidade do Minho

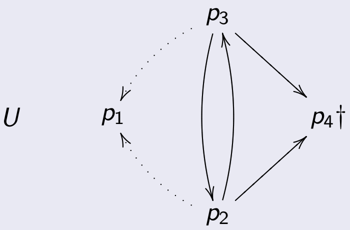
Run



User input asks p_1 to form a tree. So p_1 searches its neighbours. Initially all process are quiescent $\dagger$.

Carlos Baquero
Distributed Systems Group
Universidade do Minho

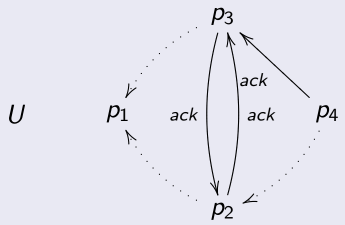
Run



p_2 and p_3 set p_1 as parent (but dont ack) and search its neighbours.

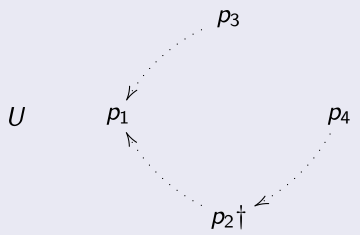
Carlos Baquero
Distributed Systems Group
Universidade do Minho

Run



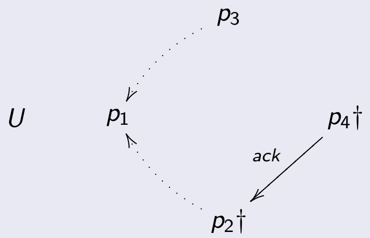
p_2 *search* arrived first at p_4, so it is set as parent and a *ack* is later sent to p_3 when its *search* arrives. The crossed searches between p_2 and p_3 produce acks as expected.

Run



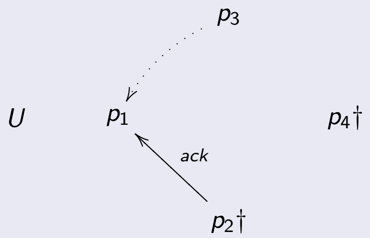
The processes can proceed a target computation sending and aknowledging messages among them. Later on p_2 becomes quiescent. He cannot report it since its son p_4 did not yet ack.

Run



p_4 becomes quiescent, acks and resets to initial conditions.

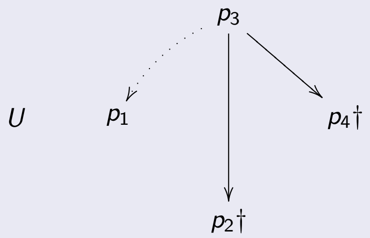
Run



p_2 acks and resets to initial conditions.

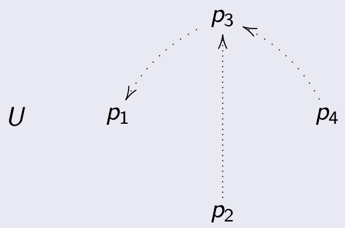
Termination Detection
Dijkstra Scholten Algorithm

Run



But p_3, that was not terminated, continues running and messages its neighbours.

Run



All processes are active again and form a new spanning tree. When all processes become quiescent, eventually p_1 will report done to the user.

Termination Detection
Dijkstra Scholten Algorithm

Run

$p_3 \dagger$

$U \leftsquigarrow p_1 \dagger$ $\qquad p_4 \dagger$

$p_2 \dagger$

Termination.

Bibliography

- Consistent Global States of Distributed Systems: Fundamental Concepts and Mechanisms. Özalp Babaoğlu, Keith Marzullo.
- Distributed Algorithms. Nancy Lynch.