Technische Universität Berlin

Distributed Algorithms 2015/16

Consistent Snapshots and Deadlock Detection

Reinhardt Karnapke | Communication and Operating Systems Group

Overview

- Snapshot problem
- Consistency criterion for consistent cuts
- Snapshot algorithms
  - Lai and Yang
  - Chandy and Lamport
- Deadlock problem
- Distributed deadlock detection
  - Chandy, Misra and Haas

CONSISTENT SNAPSHOTS

The Snapshot Problem

- Aim: Determine “current” *snapshot* of the global state *without* stopping the system
- *Global State*: Local states + messages
- Consistent snapshots are important
  - Determine safety points for a distributed database
  - Find out the current load of a distributed system
  - Does a deadlock exist?
  - Has the algorithm terminated?
  - Can an object be collected?

Stable predicates

- How can a “consistent” snapshot be determined?

Problems with Determining the Snapshot

- One cannot catch all processes at the same time
- Messages that are on the way cannot be seen
- The determined state
  - is generally out of date,
  - under certain circumstances has never “really” been like that
  - Is probably inconsistent because messages from the future were received

- Requirement: The determined state should at least be consistent, i.e., the saved state should not be influenced by messages from the future

Snapshot Algorithms – Purpose

- Snapshot algorithms provide a potential consistent past global state
- Global predicates can only be evaluated by means of consistent snapshots
- A predicate is called *stable* (or *monotonous*) if it continues to hold after it applied once
- A potential past state is useful for the detection of stable predicates: If a stable predicate applies for such a state, it now applies for sure!

Consistency Criterion for Cuts

Cut $X = \{c_1, \ldots, c_n)$

Vectors $\quad t_x = max(V(c_1), \ldots, V(c_n))$

$\quad\quad d_x = (V(c_1)[1], \ldots, V(c_n)[n])$

Maximum of rows

Elements of diagonal

***X* is consistent if and only if,**

$t_x = d_x$ for all *x*.

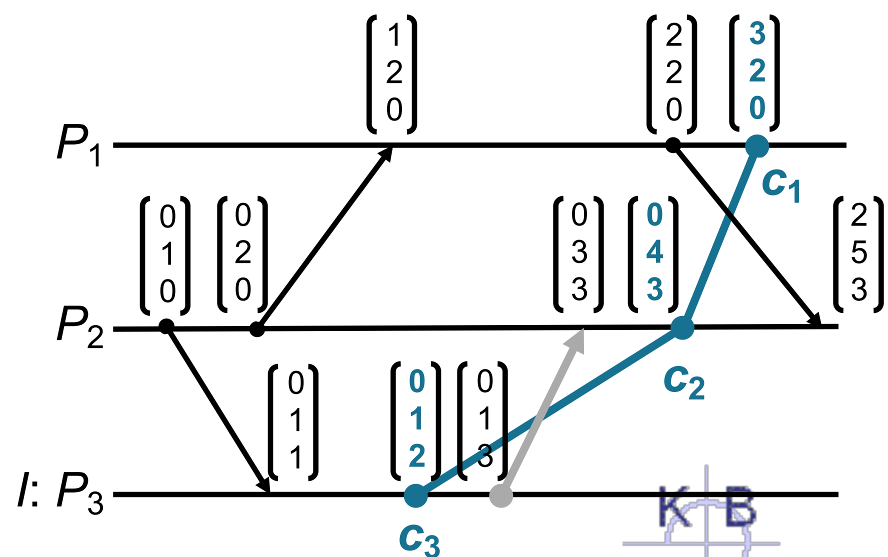
- Each element of the diagonal of the matrix must be equal to the maximum of the respective row.
- If $t_x[i] > d_x[i]$ applies, a process P_j received a message from P_i, that was sent after the visitor was at P_i, and that arrived before it was at P_j
- The case $t_x[i] < d_x[i]$ cannot occur due to the formation of the maximum.

$$\begin{array}{ccc} V(c_1) & V(c_2) & V(c_3) \\ \hline \multicolumn{3}{c}{\begin{pmatrix} 3 & 0 & 0 \\ 2 & 4 & 1 \\ 0 & 3 & \end{pmatrix}} \end{array}$$

$$t_x = (3, 4, 3)^T$$

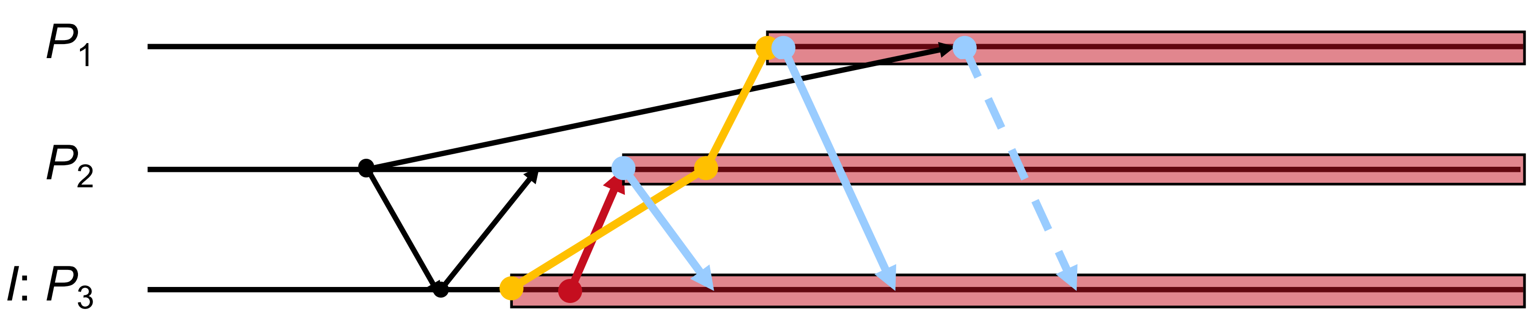
$$\neq$$

$$d_x = (3, 4, \;)^T$$



A Snapshot Algorithm (Lai and Yang, 1987)

- Initially, all nodes are black and they send black messages
- The initiator of the algorithm becomes red and stores its local state
- Red nodes do only send red messages
- Other nodes become red if they receive the order to snapshot *or* a red message
- Before a node becomes red, it saves its local state and sends it to the initiator
- If a red node receives a black message, it sends a copy of the message to the initiator → termination?



Termination of the Snapshot Algorithm

- The snapshot is complete
  - if the initiator has received the local states of all nodes, *and*
  - a copy of each black message that was on the way

- How does the initiator know that it has received all black messages?

- Deficit counter determines number of black messages that were still on the way
  - Each node counts the messages sent and received
  - Counter reading is part of the local state und is, thus, saved with the snapshot
  - The difference of both counters indicates the number of black messages to be expected

Algorithm by Chandy and Lamport, 1985

- Uses flooding as basic wave procedure

– Requires reliable FIFO-channels

- Uses the flushing principle for communication channels
  - A control message “pushes” the black messages that are still on the way out of the FIFO-channels
  - If a node has received a control message over a channel, it knows that it will receive no more black messages over that channel

Algorithm by Chandy and Lamport

Altogether, a process *P* receives exactly one control message from each of its neighbors

1. Case: A process *P* receives a control message for the first time
   - Let *Q* be the process, *P* received the control message from
   - *P* saves its state *SP* and notes the channel <*Q*, *P*> as empty
   - *P* sends a control message to all its neighbors
2. Case: A process *P* receives another (second, third, …) control message
   - Let *R* (≠ *Q*) be the process, *P* received that control message from
   - *P* notes for the channel <*R*, *P*> the sequence of basic messages which it received from *R* since the receipt of the very first control message

The snapshot then consists of all local states as well as all sequences

DISTRIBUTED DEADLOCK DETECTION

Deadlock

Three necessary conditions for the occurrence

1. Exclusive usage (mutual exclusion)
2. Processes own resources while waiting for others (hold and wait)
3. No preemption possible (no preemption)

Additional sufficient condition

4. There is a circular waiting situation (circular wait)

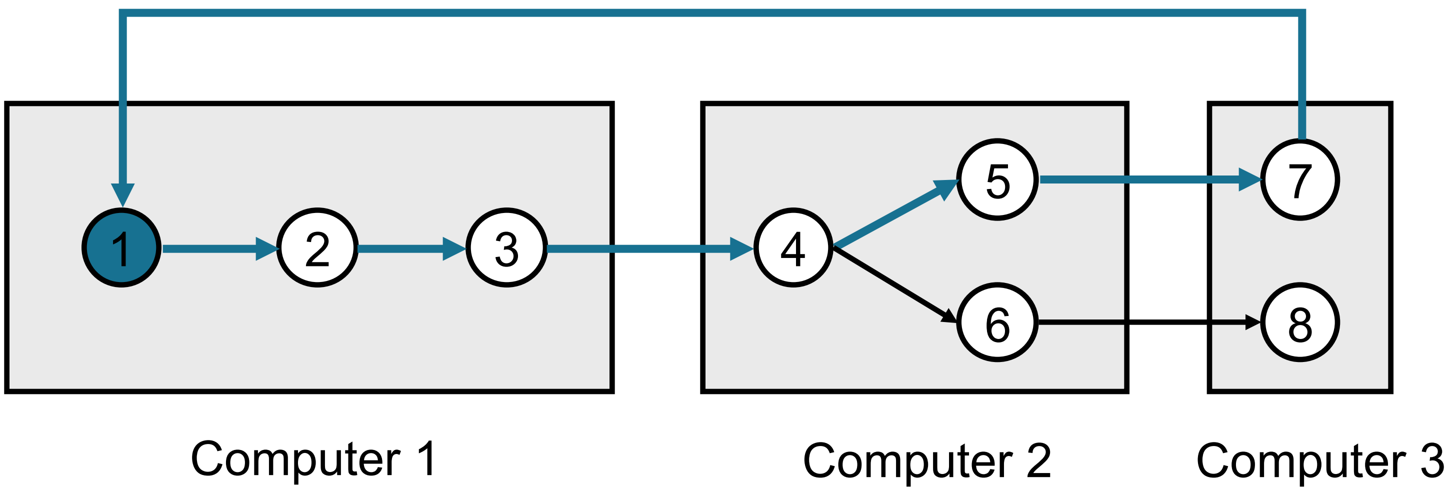
Distributed deadlock: several nodes involved

Algorithm by Chandy, Misra, and Haas, 1983

- If a process has to wait, it sends a message to the process currently using the required resource. If a process waits for several processes, it sends a message to each of them.
- The message contains the ID of the waiting process, the ID of the sender and the ID of the receiver
- The receiver checks whether it waits itself; if so it modifies the message
  - The first component remains
  - the second is substituted by its own ID
  - the third is the ID of the process it is waiting for (and the message goes to)
- If the message arrives at the original sender (recognizable at the first component), a circle exists

- Example?

Distributed Deadlock Detection



Literature

1. G. Coulouris, J. Dollimore, and T. Kindberg. Distributed Systems: Concepts and Design. Addison-Wesley, 4th edition, 2005. Chapter 11.5 + 11.6
2. A. S. Tanenbaum and M. van Steen. Distributed Systems: Principles and Paradigms. Prentice Hall, 2002. Chapter 5.3
3. N. Lynch. Distributed Algorithms. Morgan Kaufmann, 1996. Chapter 19
4. **F. Mattern. Verteilte Basisalgorithmen. Springer-Verlag, 1989. Kapitel 3: Das Schnappschussproblem**
5. G. Tel. Introduction to Distributed Algorithms. Cambridge University Press, 2nd edition, 2000. Chapter 10
6. K. M. Chandy, J. Misra, and L. M. Haas. Distributed deadlock detection. ACM Transactions on Computer Systems, 1(2):144--156, 1983.
7. K. M. Chandy and L. Lamport. Distributed Snapshots: Determining Global States of Distributed Systems. ACM Transactions on Computer Systems, 3(1):63--75, February 1985.
8. T. H. Lai and T. H. Yang. On distributed snapshots. Information Processing Letters, 25(3):153--158, 1987.