Technische Universität Berlin

Distributed Algorithms 2016/17
Consistent Snapshots and Snapshot Application

Odej Kao | Complex and Distributed IT Systems

With material of R. Karnapke | KBS

Overview

- Snapshot problem
- Consistency criterion for consistent cuts
- Snapshot algorithms
  - Lai and Yang
  - Chandy and Lamport
- Snapshot applications

Technische Universität Berlin

CONSISTENT SNAPSHOTS

CIT
Complex and Distributed IT Systems

The Snapshot Problem

- 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?

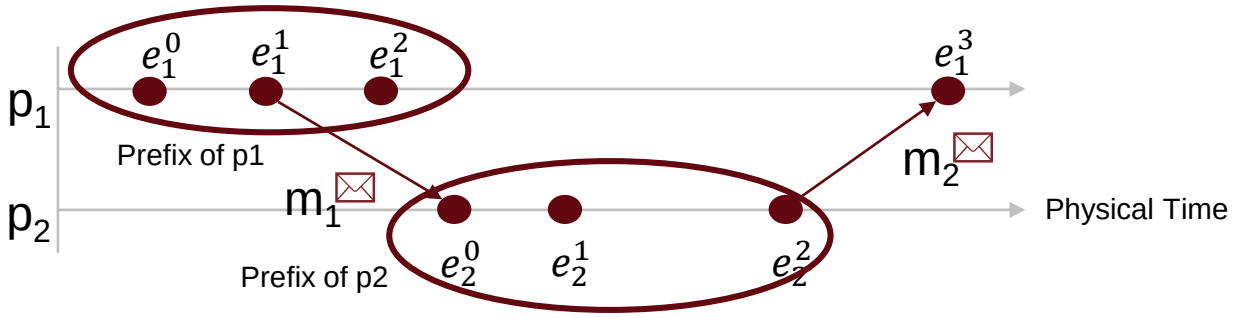
- How can a “consistent” snapshot be determined?

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  - is generally out of date,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

Global State: Definition?

- The execution of each process in our Distributed System can be characterized by its history
  - $history(p_i) = \ h_i = < e_i^0, e_i^1, e_i^2, \ldots >$
  - We define a prefix of the process history as: $h_i^k = < e_i^0, e_i^1, \ldots e_i^k >$
  - We define s_i as the state of the process p_i, each event causes a state transition
    - s_i^k denotes the state of pi immediately before the *kth* event (s_i^0 is the initial state)



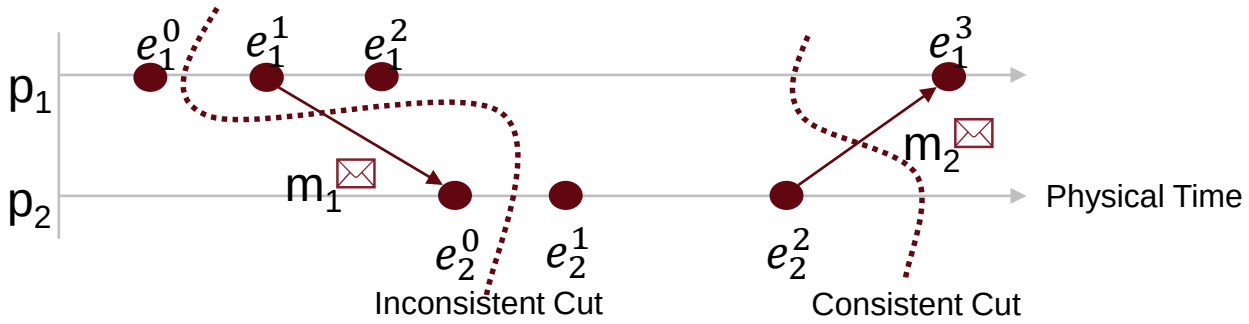
Global State

- The global history *H* is the union of the individual process histories
  - $H = h_1 \cup h_2 \cup \ldots \cup h_N$; $S = (s_1, s_2, \ldots s_3)$
  - Accordingly, a global state *S* is represented by any set of individual process states
- Basically, we can aggregate any set of states of the individual processes to form a global state
  - Which global states are meaningful? -> which process states could have occurred at the same time?
  - A global state corresponds to initial prefixes of the individual process histories
  - A Cut *C* of the system´s execution is a union of prefixes of process histories

$$C = h_1^{c1} \cup h_2^{c2} \cup \ldots \cup h_N^{cn}$$

Global State

- A meaningful global state is represented by a Consistent Cut
  - As a result of C = $h_1^{c1} \cup h_2^{c2} \cup \ldots \cup h_N^{cn}$, the cut contains all events up to e_i^{ci}
  - State s_i of each process contained in global state S corresponding to the Cut C is that of p_i immediately after the last event e_i^{ci} has been processed
  - The set of last events of the individual process prefixes is called frontier
  - Example – Inconsistent Cut: *frontier* $< e_1^0, e_2^0 >$
    - This Cut is inconsistent because it shows an effect without cause
    - p_2 includes the receipt of message m_1, but p_1 does not include the sending



Global State

- A meaningful global state is represented by a Consistent Cut
- Example – Consistent Cut: *frontier* $< e_1^2, e_2^2 >$
  - This Cut is consistent although it does not contain the receipt of message m_2
  - consistent with the actual execution, because messages may take some time to arrive

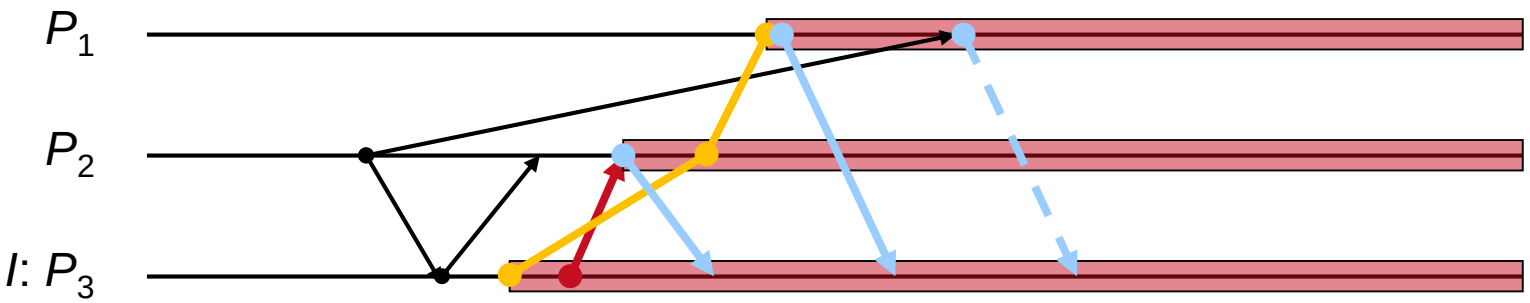
- A Cut C is consistent if, for each event it contains, it also contains all events that happened-before
  - $\forall e \in C, f \rightarrow e \Rightarrow f \in C$

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!

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, Channel state
- The snapshot then consists of all local states as well as all sequences

Algorithm by Chandy and Lamport

- The marker message (control message) rules of the snapshot algorithm:
  - All messages received on channels after recording local state belong to channel state

Marker receiving rule for process p_i

On receipt of a marker message at p_i 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 set;
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

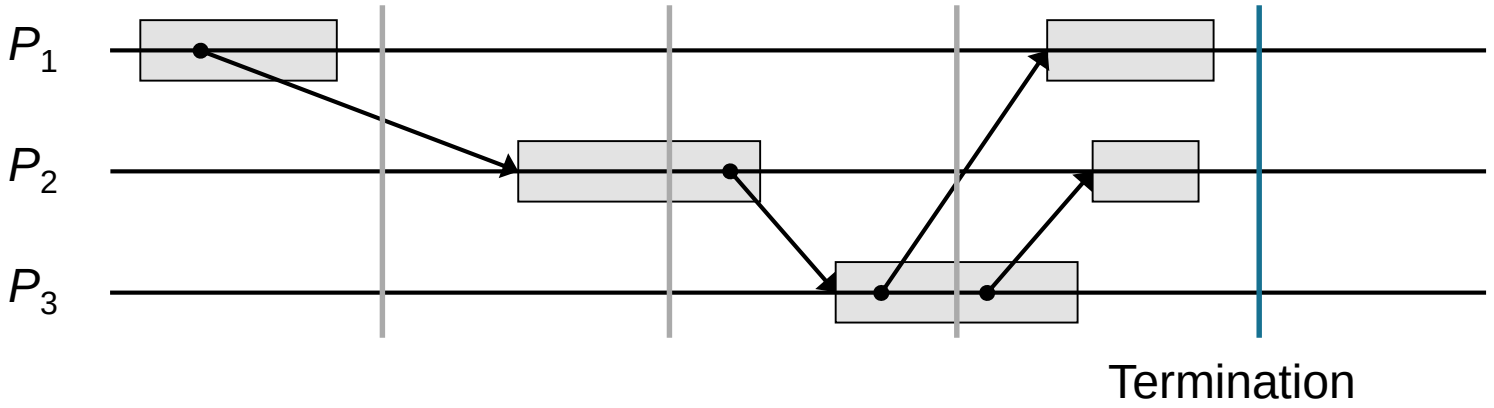
Marker sending rule for process p_i

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

DISTRIBUTED TERMINATION DETECTION

Asynchronous Model

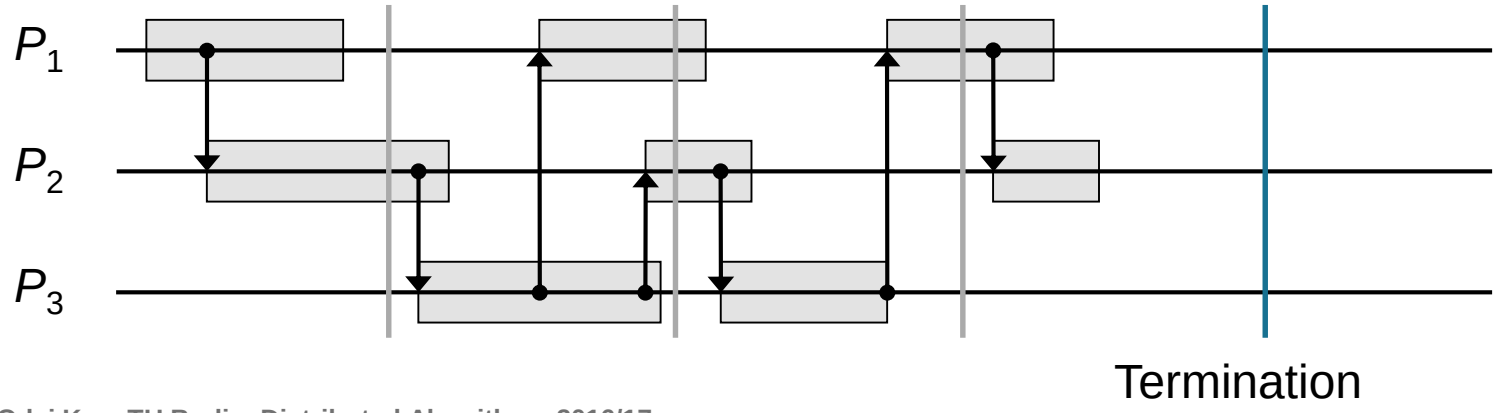
- Processes are *active* or *passive*
- Only active processes can send basic messages
- An active process can become passive at any time
- A passive process can only be reactivated by a basic message
- Termination detection: Determine whether all processes are passive and no messages are on the way at a certain point in time



Process Model

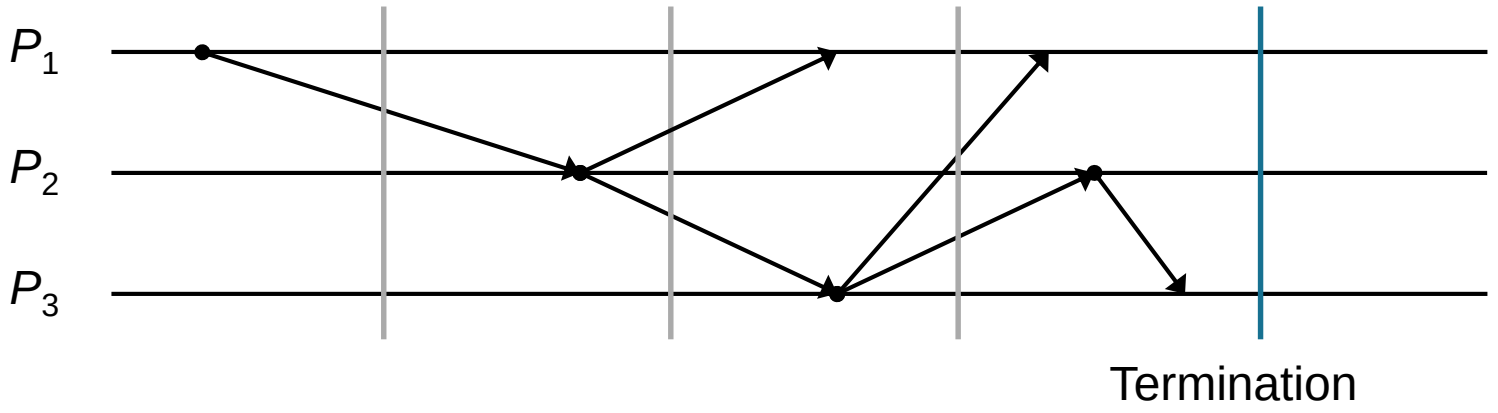
- Difference to the asynchronous model
  - Messages have no delay
  - This can be imitated by synchronous communication
    - Sender is blocked until the message is received

  ⇒ Perpendicular arrows in space-time-diagram
- Termination detection: determine, whether all processes are *passive* at a *certain* point in time



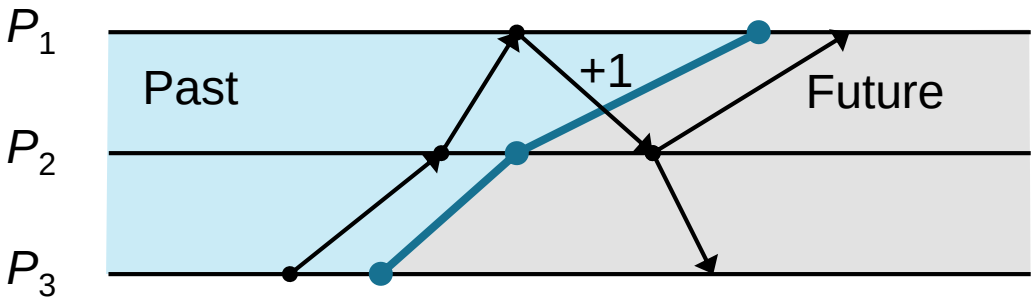
Atom Model

- Difference to the asynchronous model: Actions are atomic and need no time
- If a process receives a message, its local state changes accordingly to the respective action and it can send out messages instantly
- Termination detection: Determining whether no messages are on the way at a *certain* point in time



Simple Counting Algorithm

- An observer visits each node one after the other and separately sums up the basic messages sent and received
- Dissimilar sums indicate that a message was sent but has not arrived yet!
- Thus: If both sums are identical, no message is on its way and the basic algorithm is terminated, isn‘t it?

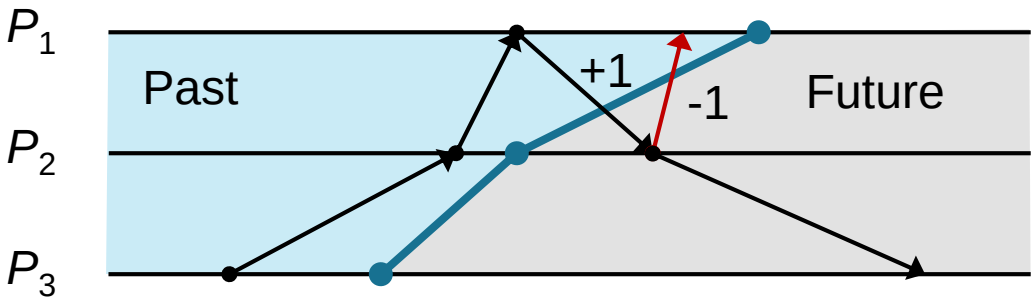


Observer visits every node and reports:
- 3 messages sent
- 2 messages received

⇒ sent - received= 1
⇒ no termination

Simple Counting Algorithm

- Unfortunately, this algorithm does not work because the condition “counters are equal” is *necessary but not sufficient* for termination
- Example: Observer reports 3 messages received and 3 messages sent → phantom termination
  - This is due to the message that was sent in the „future“, but received in the „past“
  - It balances the summation difference



Observer visits each node and reports:
- 3 messages received
- 3 messages sent

⇒ sent - received= 0
⇒ Termination (false)

Solution Ideas

- Freeze the communication in the dangerous area
  ⇒ strong decrease of the concurrency of the system
- Subsequent check ⇒ Double Counting Algorithm
- Detecting or avoiding of inconsistent time cuts through logical time stamps
  ⇒ Time Zone Algorithm
- Differentiated Counting ⇒ Vector Algorithm

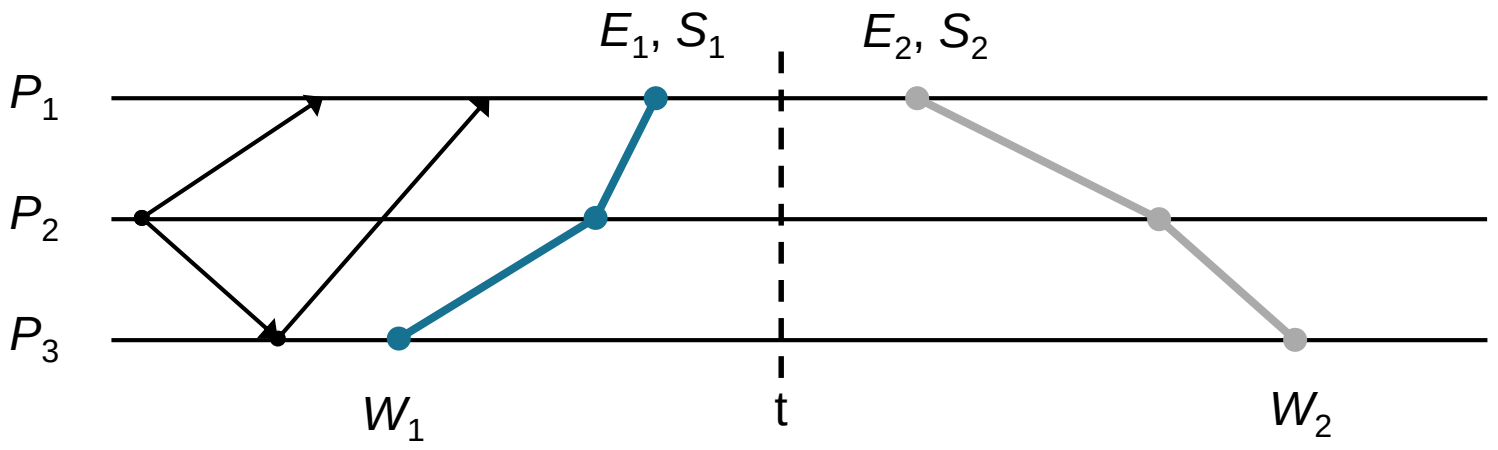
Freezing the System

- First wave freezes the system
  - No process accepts a message anymore
  - Messages arriving in the meantime are buffered and regarded as „still on the way“
  - ⇒ No messages are sent in the frozen system
- Second wave sums up the messages sent and received
  - If both sums are equal, the system is terminated
- Third wave unfreezes the system again
- Obvious disadvantage: massively decreased concurrency

Double Counting Algorithm

- An observer twice visits all nodes one after each other and determines the respective sums of the messages received and sent
- If all four sums are equal, the basic algorithm terminated:

  $E_1 = S_1 = E_2 = S_2$



Double Counting Algorithm – Characteristics

- If termination was not detected, use second wave of the previous round as the first wave of the new round
- Number of the control rounds is a priori not limited by the number of basic messages
  - There may be a very slow basic message; while it is on the way arbitrarily many control rounds may be started
  - The following variant removes this characteristic: a process containing a basic message, without sending a new one, starts a new round
- Double counting Algorithm is *re-entrant*
  - Local states of the visited processes are not changed
  - Several concurrent initiators can test for possible termination at the same time

Control Topologies

The waves for the termination detection can be realized differently

Sequential Waves

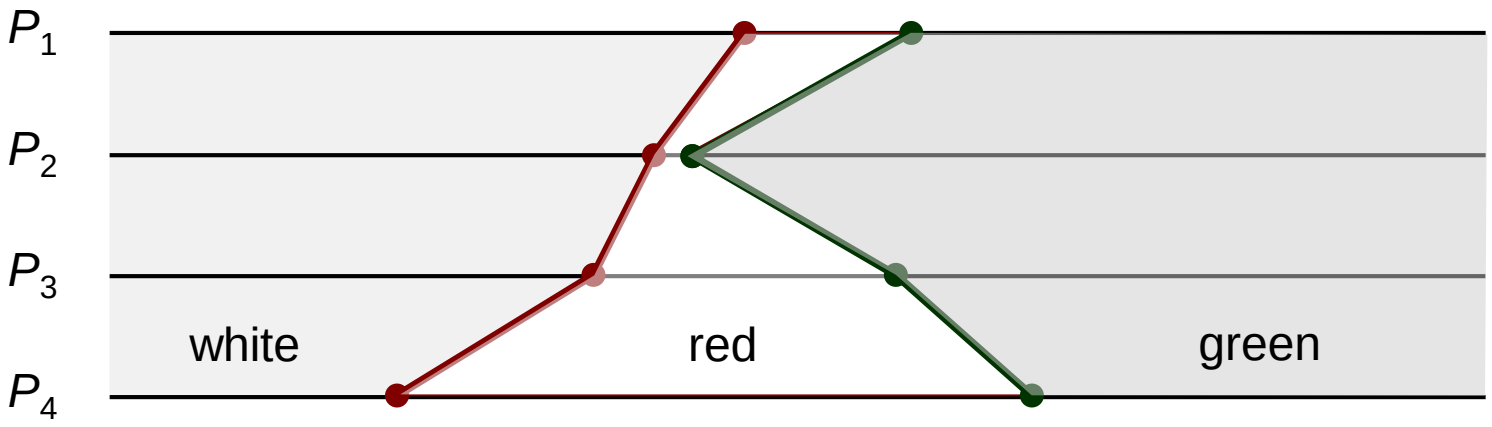
- E.g., through the construction of a logical ring and two subsequent sequential ring circuits of a token which sums up the counter reading separately for both circulations
- Time and message complexity $O(n)$

Parallel Waves

- E.g., through the construction of a span tree and two subsequent accumulations of the counter readings, *each* from the leafs to the initiator
- Achieves a better time complexity through parallel messages
- With well-balances trees time complexity is $O(\log n)$

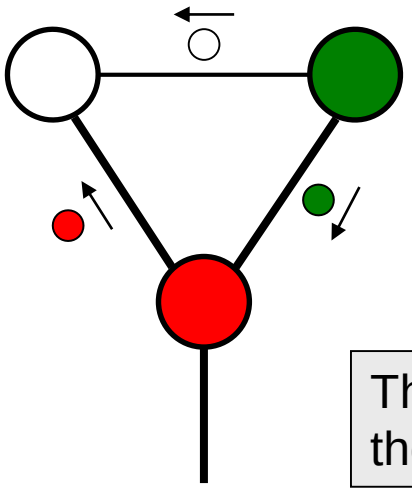
Using the Echo-Algorithm

- Usage of the forth wave (nodes become red) and the back wave (nodes become green) of a single run of the echo-algorithm for the accumulation of the counter readings
- That works because a green node cannot have a white neighbor; thus, a green message cannot reach a white node



Usage of the Echo-Algorithm

- Remark: with constructed spanning trees, the usage of the forth wave and the back wave of a single instance does not work correctly!
- Assume, a spanning tree is constructed on a topology that is not a tree
- Then, there is at least one edge that is not part of the spanning tree
- Over this edge, a basic message can get from a green node to a white node balancing the difference of the sums



Explorer

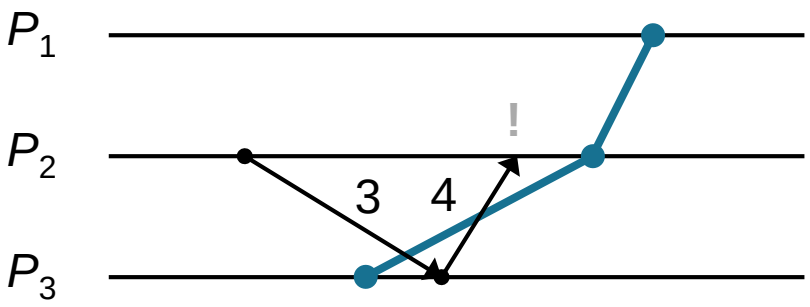
Echo

Basic Message

The basic message arrives before the explorer.

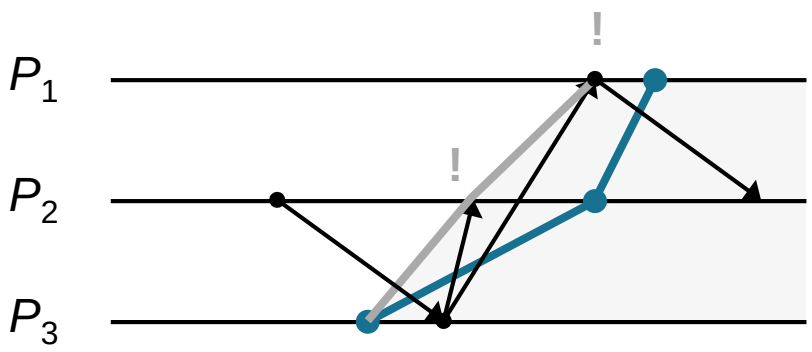
Time Zone Algorithm

- Again, an observer visits all nodes one after the other and builds a send and receive sum, respectively
  - But now, the visit of the observer increments a time zone counter on the visited node
  - Current value of time zone counter is attached to every basic message from the sending node
  - Thus, messages from the future can be recognized
    → They set a flag evaluated by the following wave and then reset
- Execute waves until both sums are equal and the flag is not set



Moving Forward the Intersection Line

- When the first message from the future is received on a node, the counters are saved
- If the observer passes by later, the saved counters are handed to it instead of the current counters
- Thus, the intersection line is moved forward
- Messages from the future then reside completely in the future



Time Zone Algorithm

- Recognizes inconsistent intersections
- Again, unlimited number of control rounds

- Disadvantages
  - Basic messages are affected
  - Not re-entrant because the local state of nodes is changed
  - Waves of several initiators can disturb each other

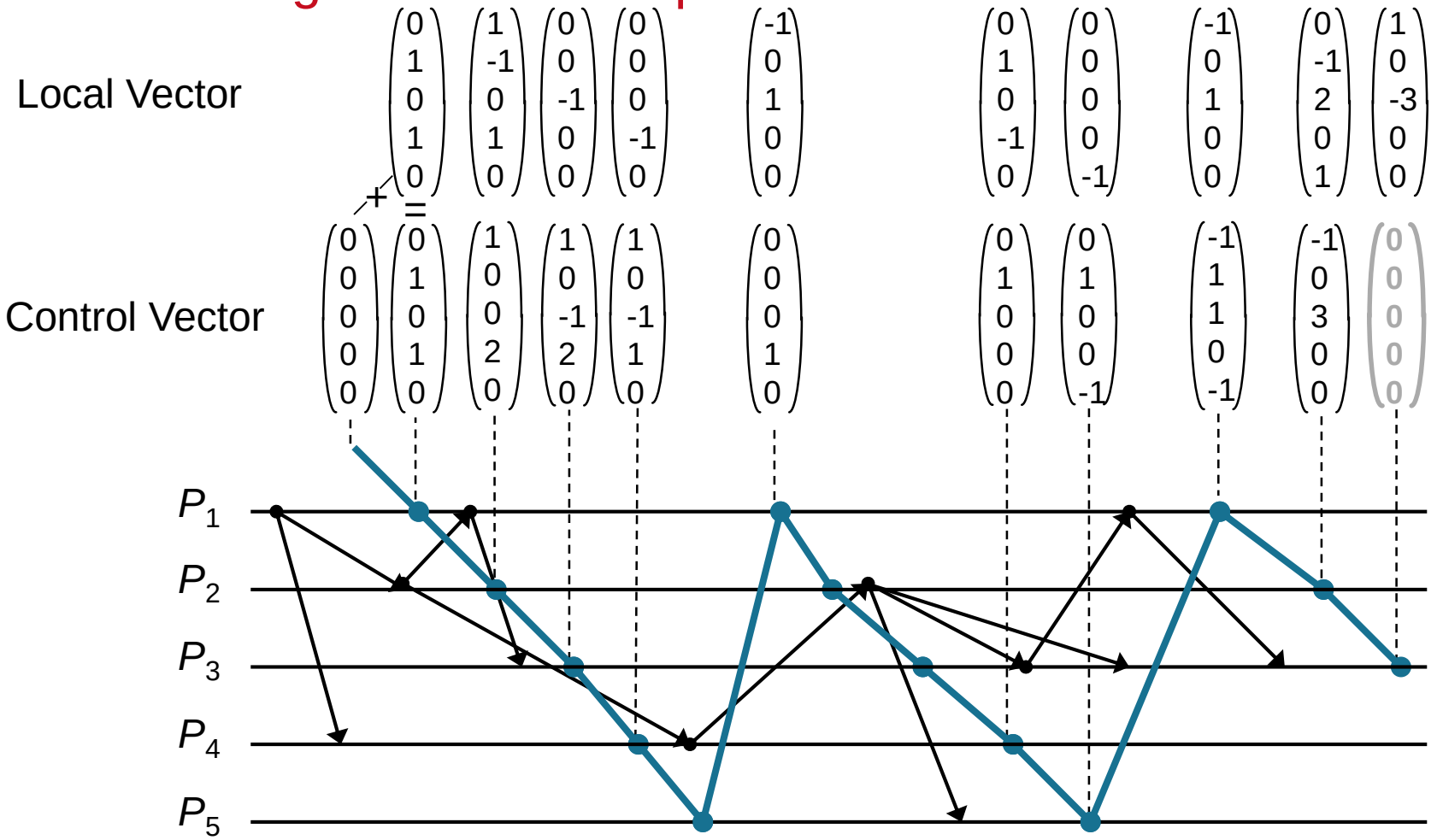
- Double counting algorithm is more elegant and more universal

Vector Algorithm

- Each process P_A has a local vector V_A with length n that is initialized to the zero vector
  - If P_A sends a basic message to P_B,$V_A[B]$ is increased by 1
  - If P_B receives a basic message, $V_B[B]$ is decreased by 1
  - A control vector C visits all processes subsequently.

- On a visit, the local vector V is added to C and V itself is set back to the zero vector
- Termination is detected, as soon as the control vector becomes a zero vector.
- Values in the vector can not lead to false-positive for termination

Vector Algorithm – Example Trace



Vector Algorithm

Improvement

- If the component of the next node to be visited is 0, skip that node
- ⇒ Potentially useless visits of such a node are avoided

Advantages

- Independent from the net topology
- Low number of control messages
- Basic messages remain untouched

Disadvantages

- Length of the control message (vector) → $O(n)$
- Algorithm is *not* re-entrant

Credit Algorithm

- Is not based on a wave algorithm, but on a global system invariant
- Primary process starts the distributed calculation with credit 1
- If a process sends a message, the message receives half of the current credit of the process
- If an active process receives a message, its credit increases by the credit of the message
- If a process becomes passive, it sends its current credit to the primary process

- The following presentation of the algorithm assumes the asynchronous model

Credit Algorithm

- Main Invariant
  - The credit sum is always 1

- Further Characteristics
  - Basic messages always carry a credit > 0 with them
  - Active processes always have a credit > 0

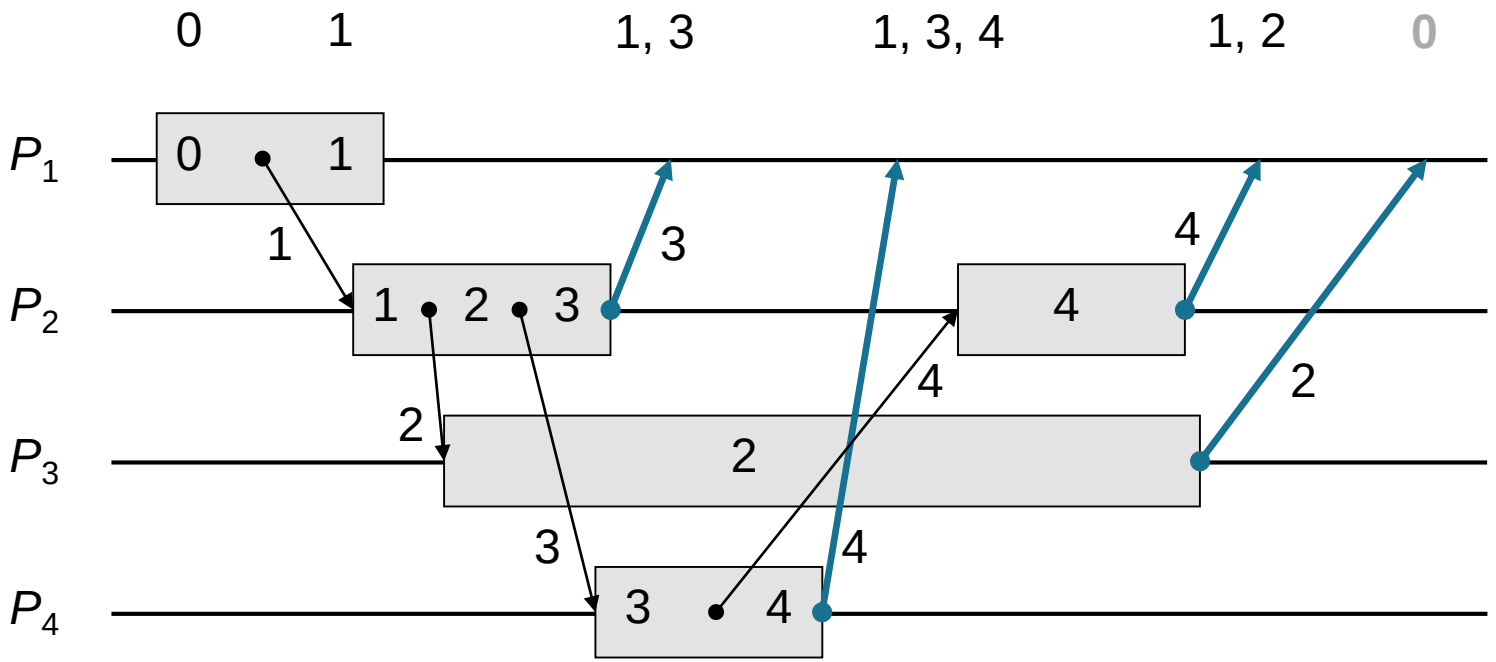
- Termination
  - If the primary process has credit 1 again

Representation of the Credit Portions

- Floating point number inconvenient
- Better: storage of the negative dual logarithm of the credit portion $c = - ld\ 2^{-d}$
  - $k = 1/1 \rightarrow c = 0$
  - $k = 1/2 \rightarrow c = 1$
  - $k = 1/4 \rightarrow c = 2$
  - …
- Halving of the credit $c := c + 1$

- Bit vector for the storage of the credit portions
- Recombination of credit portions through binary addition

Example Trace of the Credit Algorithm



Control Message

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 and L. Lamport. Distributed Snapshots: Determining Global States of Distributed Systems. ACM Transactions on Computer Systems, 3(1):63--75, February 1985.
7. T. H. Lai and T. H. Yang. On distributed snapshots. Information Processing Letters, 25(3):153--158, 1987.