CES-27 Processamento Distribuído

Termination Detection

Prof Juliana Bezerra
Prof Celso Hirata
Prof Vitor Curtis

Outline

- Motivation
- Dijkstra's Token Algorithm
- Dijkstra-Scholten Algorithm
- Shavit-Francez Algorithm
- Rana's Algorithm

Motivation

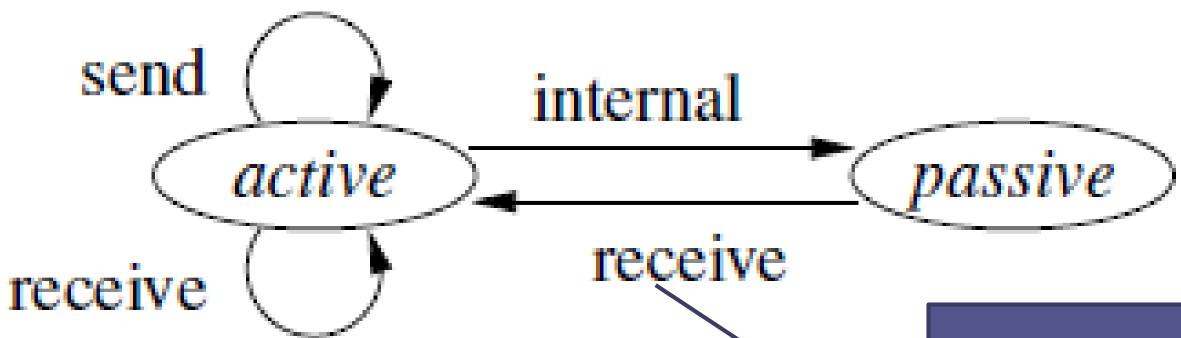
- **Termination detection for <u>diffusing computation</u>**
  - "The activity of a finite computation may propagate over a network of machines, when machines may delegate (repeatedly) subtasks to their neighbors; when such a computation is fired from a single machine, we call it a <u>diffusing computation</u>" (Dijkstra & Scholten, 1979)

- A <u>distributed algorithm</u> is **terminated** if all **processes are in a terminal state and no messages (basic) are in transit**
  - The **basic** algorithm is the algorithm for which termination is being detected
  - The **control** algorithm is the termination detection algorithm employed

- The **control algorithm** in general consists of two parts:
  - Termination detection ⇨ our focus
  - Announcement ⇨ simple
    - The process, which detects termination, can act as a master
    - Send a terminate message to everyone
    - Collect the results and print them
    - Send a shutdown message to everyone
    - Stop

Motivation

- **Termination detection** is a fundamental and <u>challenging problem</u> in distributed computing, because **no process has complete knowledge of the global configuration** of the network
  - A **terminated process** may be **reactivated** by a message from another process
  - Ideally, termination detection does not require freezing the basic execution
  - Process model:



Only messages of **basic** algorithm make the process *active* again

Concept: Wave Algorithms

The way the wave flows is up to you!
It depends on the chosen topology.
E.g. Directed ring
E.g. Undirected ring

- In **wave algorithms**, each computation gives rise to one or more **decisions** in which **all processes have a say**
  - **They can not complete** if any process ***p*** **refuses** to take part in its execution
    - In this case, *p* refused because *p* did something (after the wave algorithm started) that not characterizes the decision
- A wave is **initiated by one process**, and in the end one *decide event* happens at the initiator
  - There can be concurrent calls of a **wave** algorithm, **initiated by different processes**
    - Then usually for each **wave** the messages are **marked with the ID of its initiator**
  - If a wave does not complete, then typically another wave will complete successfully later on

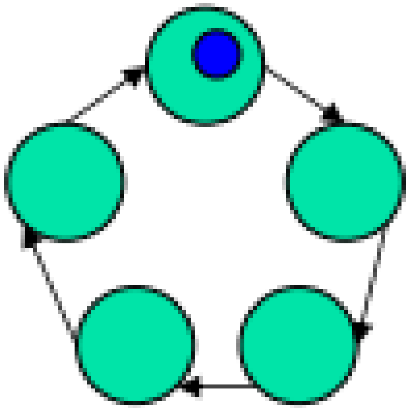
Outline

- Motivation
- Dijkstra’s Token Algorithm
- Dijkstra-Scholten Algorithm
- Shavit-Francez Algorithm
- Rana’s Algorithm

Dijkstra’s Token Algorithm

- An initial idea:
  - Imagine the processes to be arranged in a ring
  - Process 1 inserts a token, traveling from process i to process i + 1 and back to process 1
  - The token only leaves a process if the process turns inactive
  - Process 1 determines whether the system can be shut down

- That does not work…
  - A process may become active after having sent the token
  - To fix this: Dijkstra’s Token Algorithm!

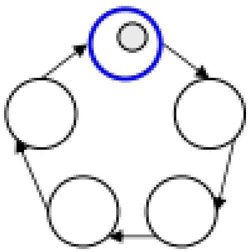


Remember:

- The ring is used only by the control algorithm (i.e. termination detection)

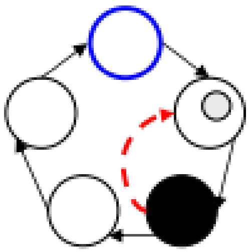
- In the basic algorithm, processes can communicate with each other (without the ring)

Dijkstra's Token Algorithm

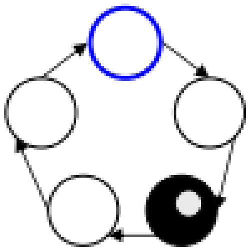


- Algorithm:
  - 1) When P1 turns inactive, it turns white and sends a white token to P2

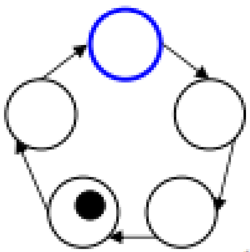
Basic message!



  - 2) If Pi sends a message to Pj and j < i, then Pi turns black
    - Pi is a suspect for reactivating process Pj



  - 3) If Pi has the token and it is idle, it passes the token. The token becomes black if Pi is black



  - 4) After passing tokens, processes become white
    - Indicating that the process is inactive
  - 5) The algorithm terminates when P1 receives a white token and it is idle

Dijkstra’s Token Algorithm

- Performance: O(N) in time
  - N is the number of processes

- P1 may become active again before getting back the token
  - Is it possible to detect termination wrongly? No!

- For a small number of processes, algorithm is acceptable.
- For large numbers of processes, this becomes a significant overhead

- **Messages must be delivered in order for the protocol to work!**

- There are similar approaches for this algorithm:
  - J. Misra, “**Detecting Termination of Distributed Computations Using Markers**”, 2nd ACM Symposium on PODC, Aug. 1983.

Outline

- Motivation
- Dijkstra's Token Algorithm
- Dijkstra-Scholten Algorithm
- Shavit-Francez Algorithm
- Rana's Algorithm

Dijkstra-Scholten Algorithm

- Reference: E. W. Dijkstra and C. S. Scholten "**Termination Detection for Diffusing Computations**", Information Processing Letters, Vol. 11, No. 1, Aug. **1980**.

- Termination detection algorithm for a **centralized basic algorithm** on an **undirected network**.

One process initiated the basic algorithm

We will track messages of the basic algorithm

A process can send/receive messages to/of any other process

Dijkstra-Scholten Algorithm

An use of spanning tree, where processes are nodes of tree and edges are connections

- The main idea:
  - To build a tree, rooted in the initiator of the basic algorithm
  - The tree contains:
    - all active processes, and
    - passive processes that have active descendants in the tree
  - If a basic message from a process *p* makes a process *q* active, then *q* becomes a child of *p* in the tree
  - A process can quit the tree only if it is passive and has no children left in the tree
    - In that case, it informs its parent that it is no longer a child
  - Termination is detected by the initiator when the tree has disappeared

There are other algorithms that use the concept of spanning tree!

Dijkstra-Scholten Algorithm

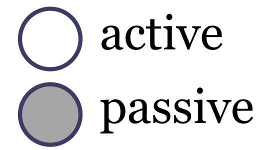
You decide how to implement the tree!

To count my children

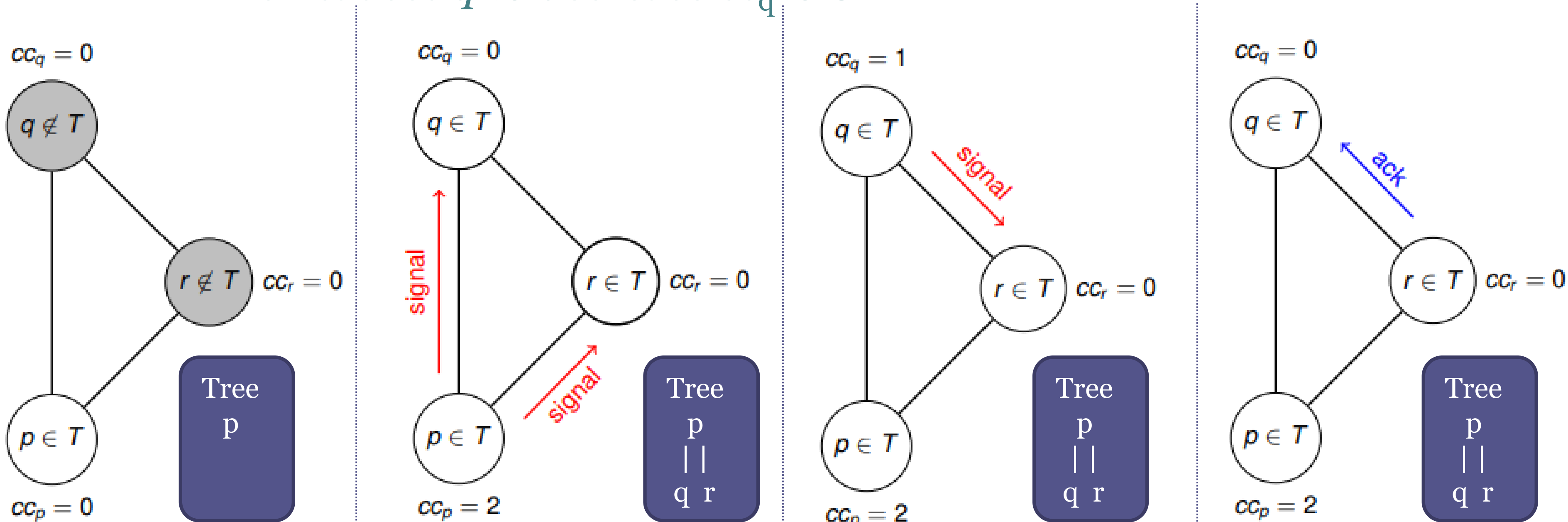
- Algorithm:
  - Initially the tree T consists only of the initiator
  - When p sends a basic message, $cc_p = cc_p + 1$.
  - Let this message be received by q:
    - If q isn't yet in T, it joins T with parent p and $cc_q = 0$
    - If q is already in T, it sends a control message to p that it isn't a new child of p. Upon receipt of this message, $cc_p = cc_p - 1$
  - When a **non-initiator** p is **passive** and $\mathbf{cc_p = 0}$, it quits T and informs its parent that it is no longer a child
    - Note: p informs its parent using a control message. So its parent decreases cc
  - When the **initiator** is passive and $cc_{initiator} = 0$, termination is detected

Drawback: Requires one control message for every basic message
Note: In the next examples, control message is the *ack* (acknowledgement)

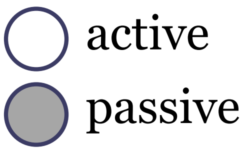
Dijkstra-Scholten Algorithm



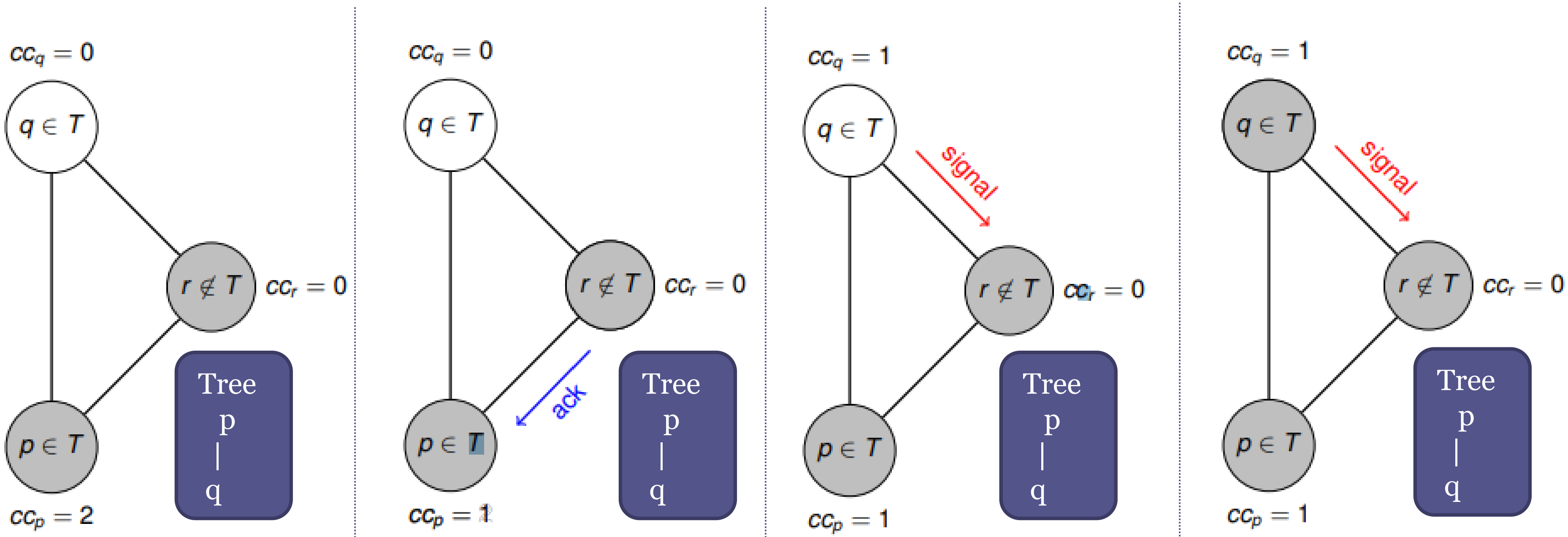
- Example:
  - Three processes *p*, *q*, and *r*
  - Chosen topology: undirected ring
  - At the start, the initiator *p* sends basic messages to *q* and *r*, and sets cc_p to 2
    - Upon receipt of these messages, *q* and *r* both become active, and join T with parent *p*
  - *q* sends a basic message to *r*, and sets cc_q to 1.
  - Upon receipt of this message, *r* sends back an acknowledgment, which causes *q* to decrease cc_q to 0



Dijkstra-Scholten Algorithm



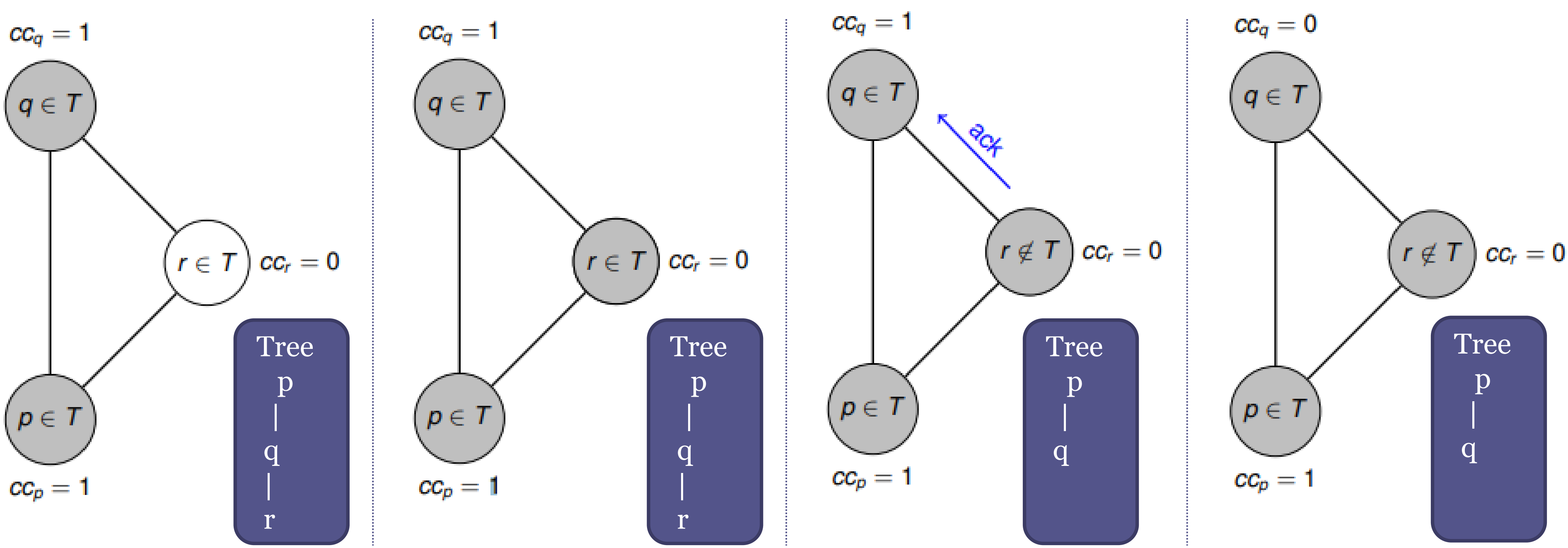
- Example (cont):
  - p becomes passive. (Since $cc_p = 2$, it remains in T)
  - r becomes passive. Since $cc_r = 0$, it sends an acknowledgment to its parent *p*, which causes *p* to decrease cc_p to 1
  - *q* sends a basic message to *r*, and sets cc_q to 1
  - *q* becomes passive (Since $cc_q = 1$, it remains in T)
  - Note that all three processes are now passive, but there is still a message traveling from *q to r*



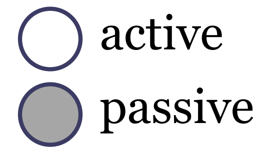
Dijkstra-Scholten Algorithm

active
passive

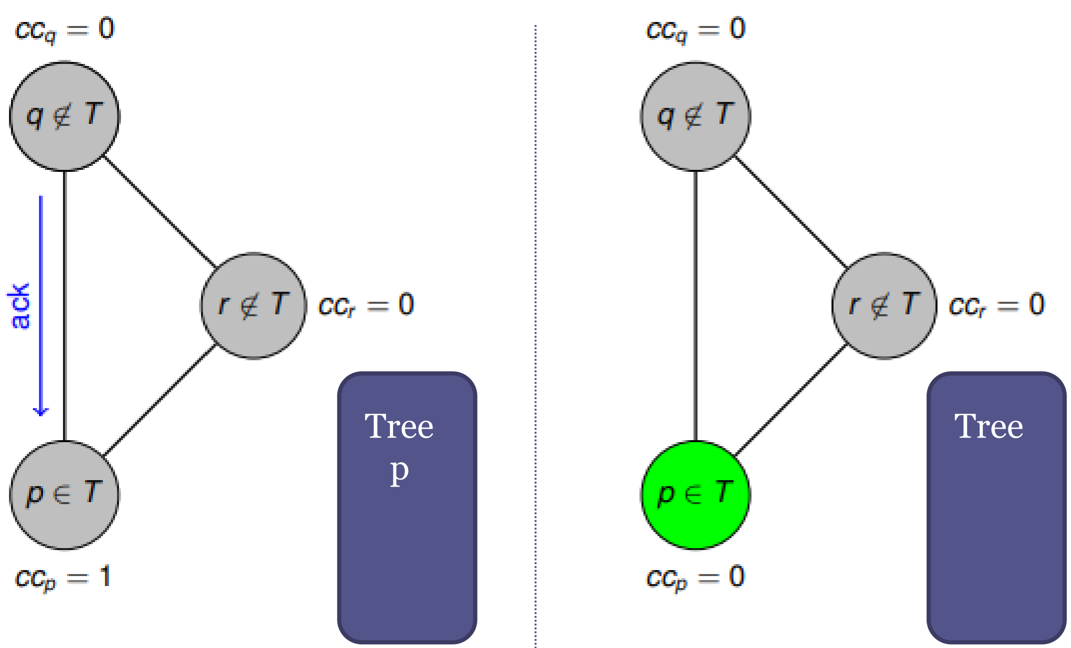
- Example (cont):
  - Upon receipt of message from q to r, r becomes active again, and joins T with parent q
  - r becomes passive. Since $cc_r = 0$, it sends an acknowledgment to its parent q which causes q to decrease cc_q to 0



Dijkstra-Scholten Algorithm



- Example (cont):
  - Since *q* is passive and $cc_q = 0$, it sends an acknowledgment to its parent *p*, which causes *p* to decrease cc_p to 0
  - Since *p* is passive and $cc_p = 0$, it detects termination



Outline

- Motivation
- Dijkstra's Token Algorithm
- Dijkstra-Scholten Algorithm
- Shavit-Francez Algorithm
- Rana's Algorithm

Shavit-Francez Algorithm

- It generalizes Dijkstra-Scholten Algorithm to **decentralized** basic computations (**multiple initiators**)

- The idea:
  - To maintain not one tree, but a forest of (disjoint) trees, one for each initiator
  - Initially, each initiator constitutes a tree in the forest
  - A process can only join a tree if it is not yet in a tree in the forest
  - For the rest, the algorithm proceeds exactly as the Dijkstra-Scholten algorithm

Shavit-Francez Algorithm

- The difference:
  - When an initiator detects that its tree has disappeared, it cannot immediately detect termination
    - There can be other trees (with active processes)
  - Instead, this initiator starts a wave, tagged with its ID
    - Only processes, which are not in a tree, participate
    - If such a wave does not complete
      - Another initiator of which the tree has not yet disappeared will start a subsequent wave
      - And the last tree to disappear is guaranteed to start a wave that will complete

Outline

- Motivation
- Dijkstra's Token Algorithm
- Dijkstra-Scholten Algorithm
- Shavit-Francez Algorithm
- Rana's Algorithm

Rana's Algorithm

- Termination detection algorithm for a **decentralized basic algorithm** on an **undirected network**

Any process can initiate the termination detection

A process can send/receive messages to/of any other process

- It is an example of a **wave algorithm**

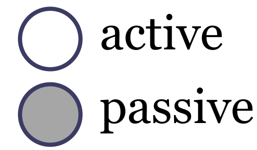
Rana's Algorithm

- The idea:
  - Each basic message is **acknowledged**
  - Lamport's logical clock provides (basic and control) events with a **timestamp**
    - The timestamp of a process is the **highest timestamp** of its events so far (initially it is 0)
  - Let process *p* at time *t* become quiet, i.e. (1) *p* is passive, and (2) all basic messages it sent have been acknowledged
    - Then *p* starts a wave (of control messages), tagged with *t* (and *p*)
    - Only processes that have been quiet in a **time ≤ *t*** take part in the wave
    - If the wave completes, termination is detected

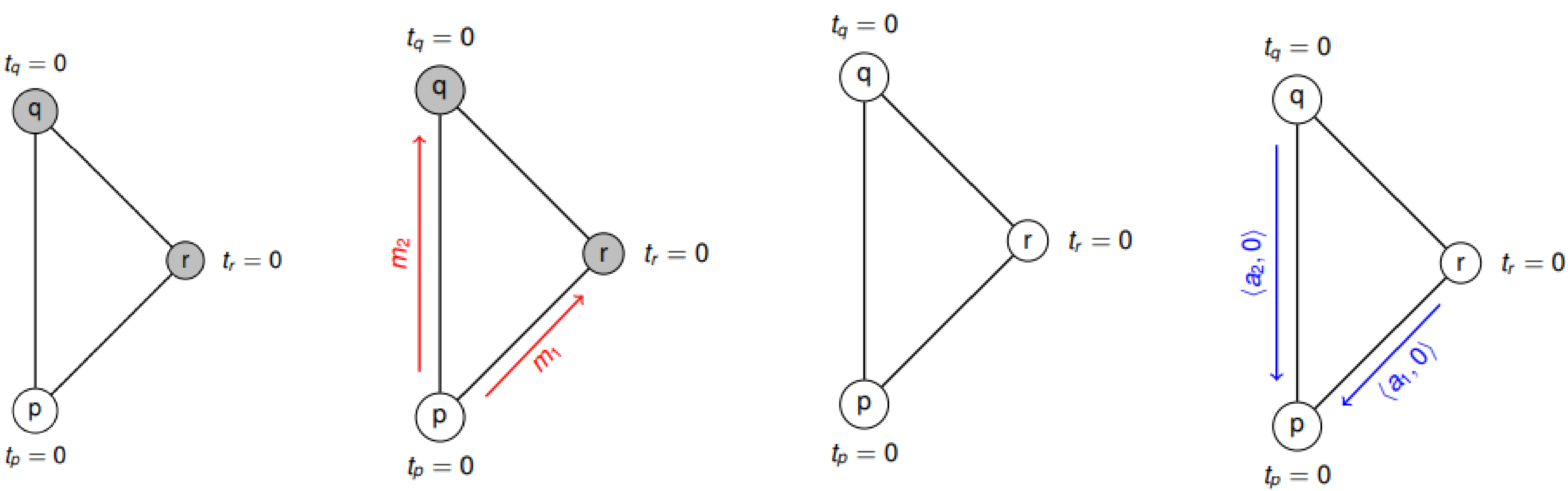
So basic messages do not contribute to logical clocks
To avoid a full-blown logical clock!

- It suffices that only the control messages (acknowledgments and wave messages) are taken into account in logical clocks
  - A wave tagged with timestamp *t* puts the clock of each recipient to *t* (if its value is not ≥ *t* already)
  - Each acknowledgment is tagged with the timestamp *t* of the sender and puts the clock of the receiver to *t* + 1 (if its value is not ≥ *t* + 1 already)

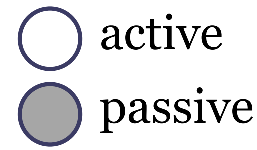
Rana's Algorithm



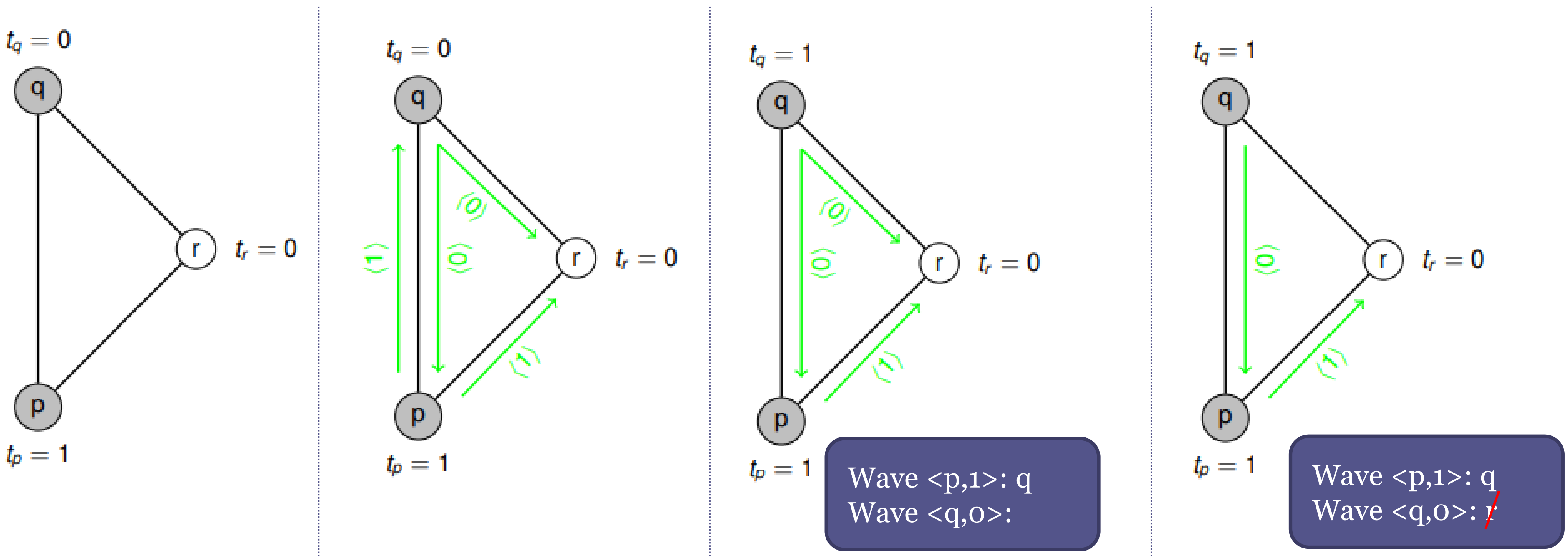
- Example
  - Consider three processes p, q, and r
  - Chosen topology: undirected ring
  - Initially, all processes have logical time 0, and only p is active
  - p sends basic messages m_1 to r and m_2 to q, which make q and r active.
  - Next, q and r send an acknowledgment $<a_2, 0>$ and $<a_1, 0>$ respectively



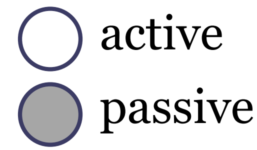
Rana’s Algorithm



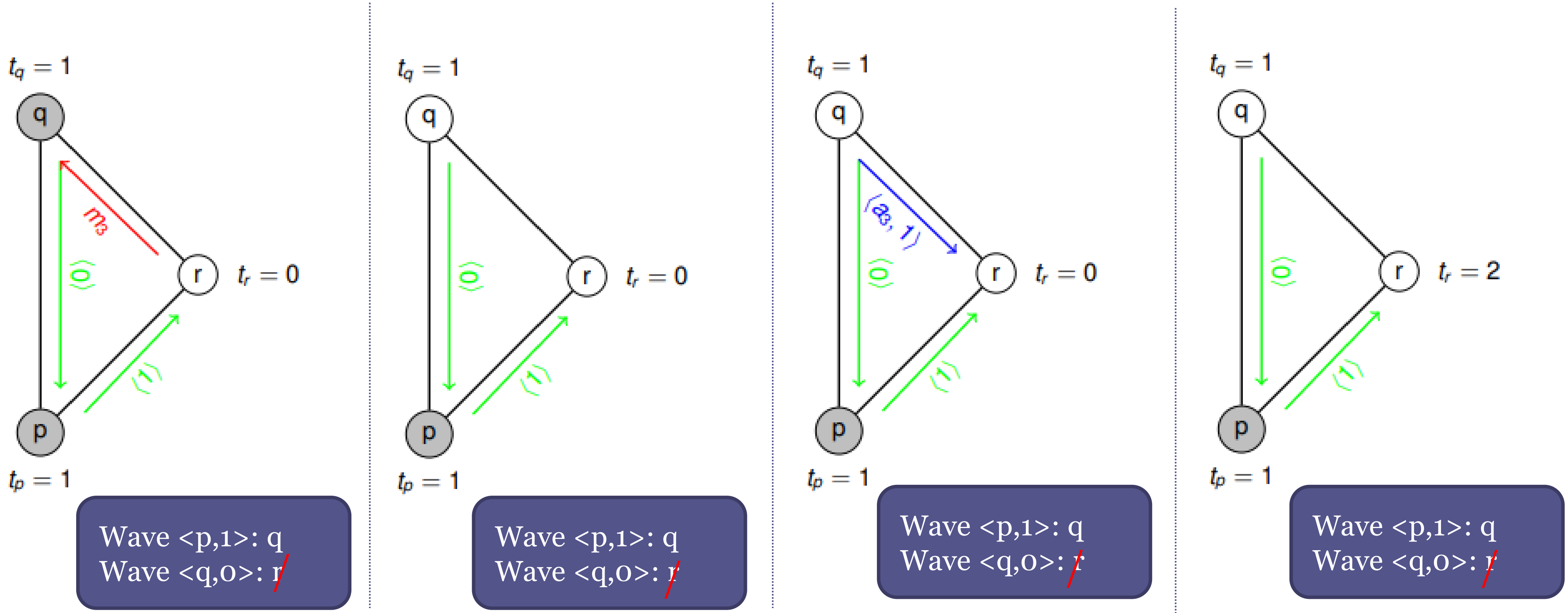
- Example (cont)
  - *p* receives both acknowledgments, setting its time to 1
  - *p* and *q* become passive.
    - Next, *p* and *q* both start a wave, tagged with 1 and 0, respectively
  - The wave of *p* first visits *q*, setting its time to 1
    - *q* takes part in the wave, because it is quiet from time 0 onward
  - The wave of *q* first visits *r*
    - *r* refuses to take part in the wave, because it is active



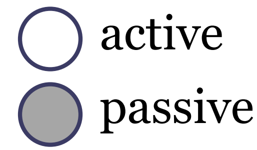
Rana's Algorithm



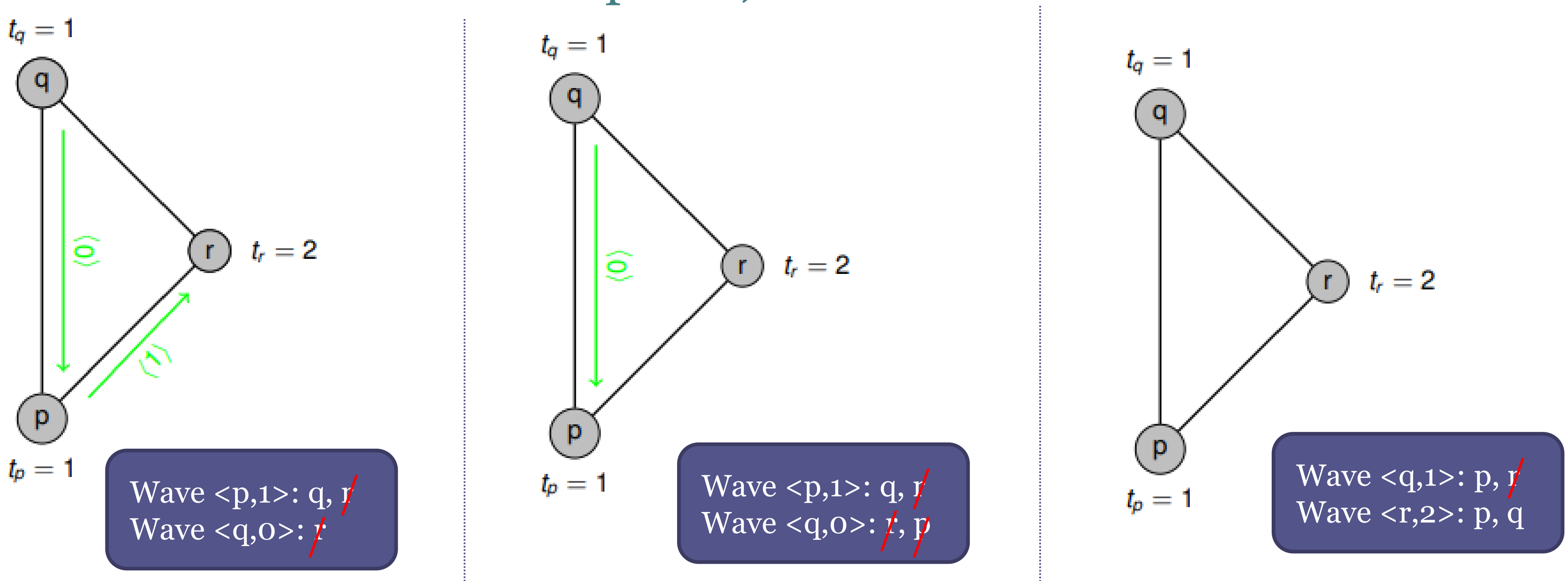
- Example (cont)
  - r sends a basic message m_3 to q
  - Upon receipt of this message, q becomes active and sends back an acknowledgment $<a_3, 1>$.
  - When r receives this acknowledgment, its clock value becomes 2



Rana's Algorithm



- Example
  - *q* and *r* become passive.
  - *r* refuses to take part in *p*'s wave, because *r* is quiet from time 2 onward, while the wave is tagged with 1
  - *p* refuses to take part in *q*'s wave. Similar reason!
  - *q* and *r* both start a wave, tagged with 1 and 2, respectively.
  - The wave of *r* completes, and r detects termination



Other algorithms in literature

- Papers:
  - Matocha, J. and Camp, T. A taxonomy of distributed termination detection algorithms. The Journal of Systems and Software, 43, **1998**

- More recent papers
  - Hasrat, I.R. and Atif, M. Formal Specification and Analysis of Termination Detection by Weight-throwing Protocol. International Journal of Advanced Computer Science and Applications(IJACSA), Volume 9 Issue 4, **2018**.