Distributed Systems

Summer Term 2020

Roland Wismüller
Universität Siegen
roland.wismueller@uni-siegen.de
Tel.: 0271/740-4050, Büro: H-B 8404

Stand: June 8, 2020

Distributed Systems

Summer Term 2020

7 Coordination

Contents

- Election algorithms
- Mutual exclusion
- Group communication (multicast)
- Transactions

Literature

- Tanenbaum, van Steen: Kap. 5.4-5.6
- Colouris, Dollimore, Kindberg: Kap. 11, 12
- Stallings: Kap 14.3

7.1 Election Algorithms

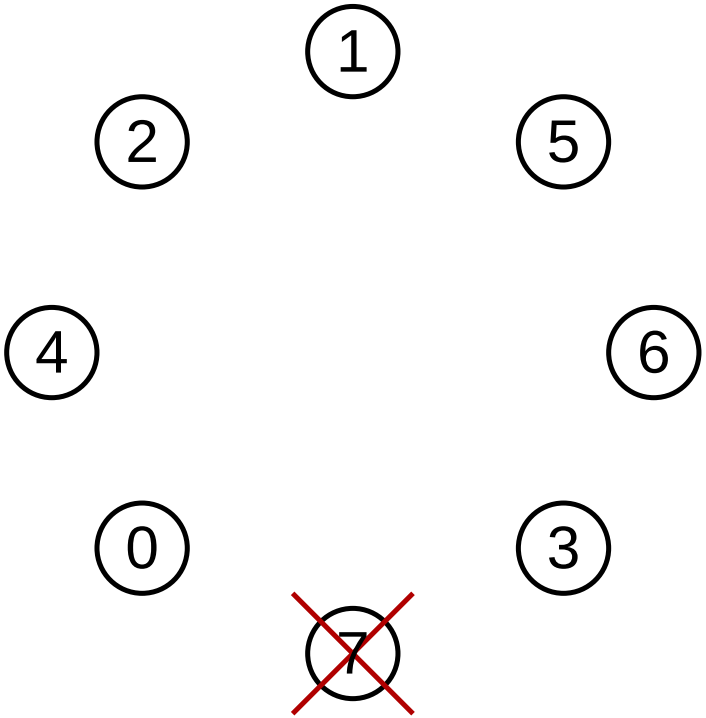
[Tanenbaum/Steen, 5.4]

- In many distributed algorithms **one** arbitrary process must play an exceptional role
  - e.g. central coordinator, initiator, ...
- Question: how to choose this process unambiguously?
  - processes must be distinguishable, e.g. via a unique ID.
  - then select e.g. the process with the highest ID
- Prerequisites / requirements:
  - election can be initiated by multiple processes simultaneously
    - e.g. after failure or recovery of a process
  - after the election all processes must have the same result
  - each process knows the IDs of all other processes, but does not know whether they are running or not

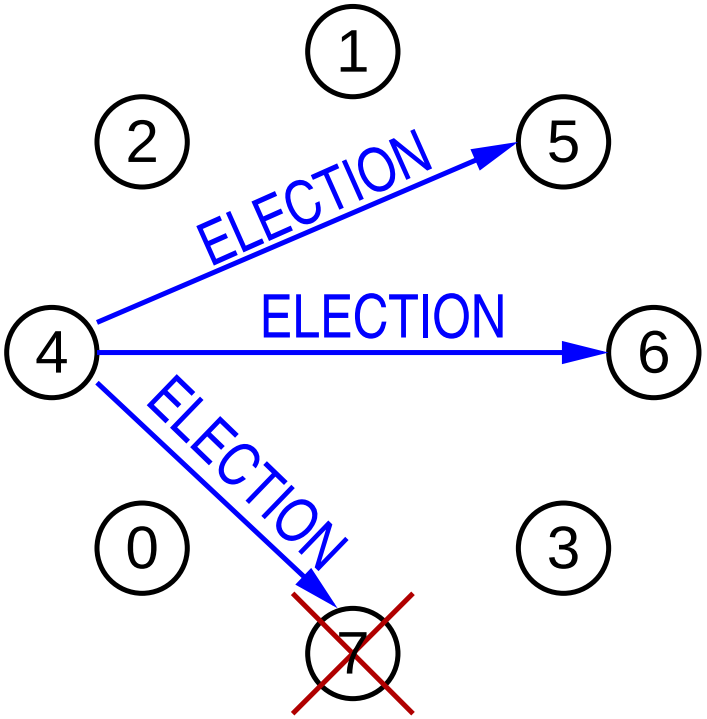
The Bully Algorithm

- A process P holds an election as follows:
  - P sends an ELECTION message to all processes with a larger ID
  - if none of the processes reacts, P wins the election
  - if a process responds: P loses the election
- When a process receives an ELECTION message:
  - (message comes from a process with a lower ID)
  - return an OK message
  - hold an election of your own
- At some point, there is only one process left
  - this wins the election and sends the result to all others

Bully Algorithm: Example

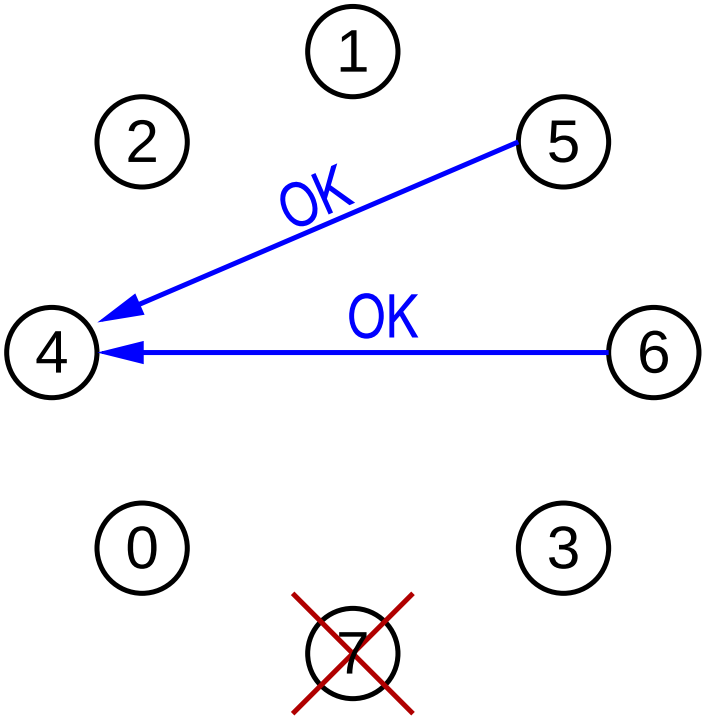


Bully Algorithm: Example



Process 4 holds an election

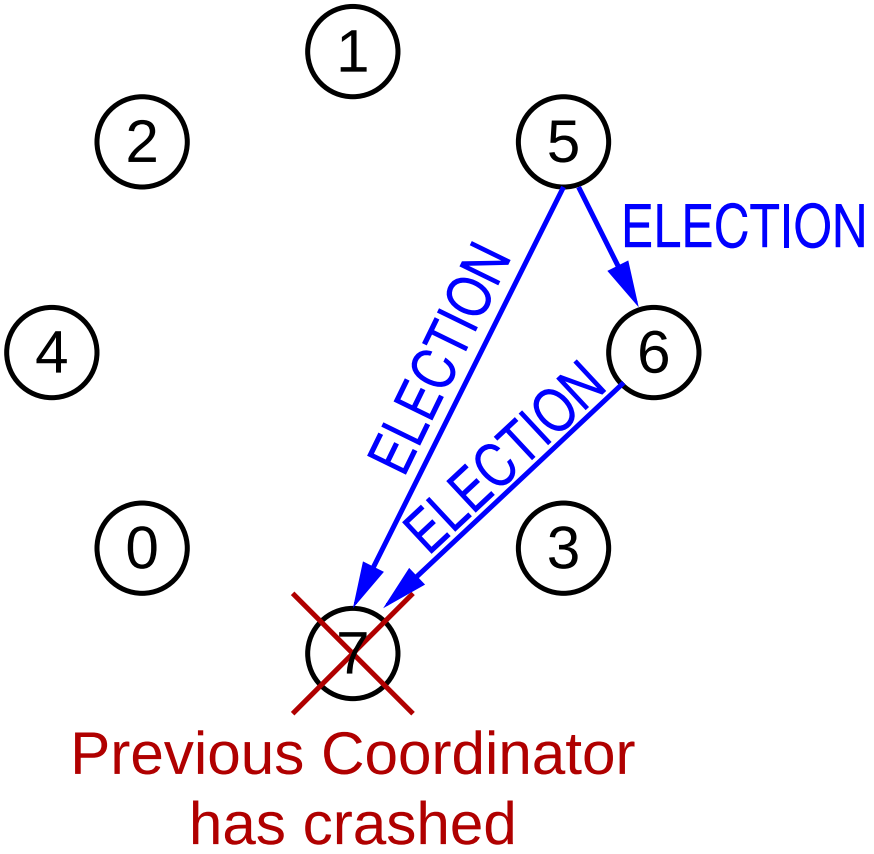
Bully Algorithm: Example



Process 4 holds an election

Processes 5 and 6 reply,
Process 4 terminates its election

Bully Algorithm: Example

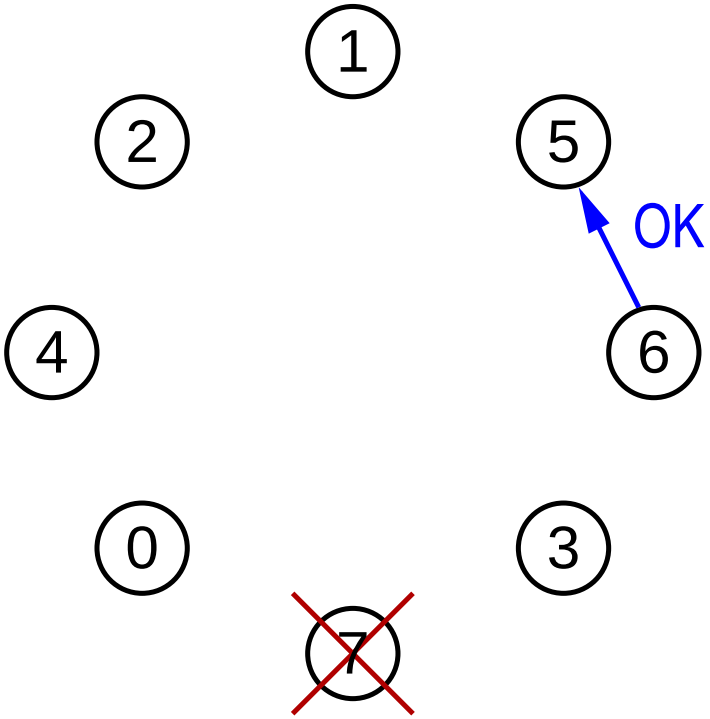


Process 4 holds an election

Processes 5 and 6 reply,
Process 4 terminates its election

Processes 5 and 6 simultaneously hold an election

Bully Algorithm: Example



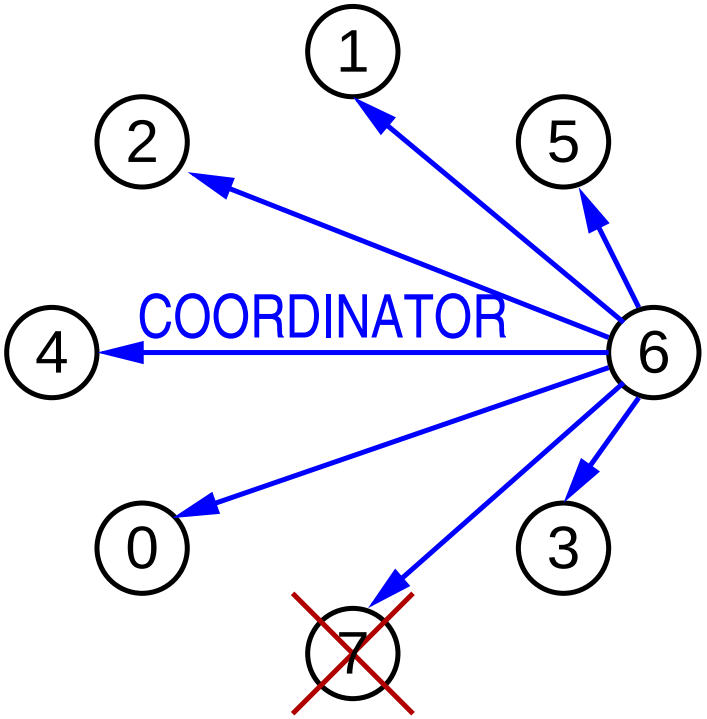
Process 4 holds an election

Processes 5 and 6 reply,
Process 4 terminates its election

Processes 5 and 6 simultaneously
hold an election

Process 6 replies to 5
Process 5 terminates its election

Bully Algorithm: Example



Process 4 holds an election

Processes 5 and 6 reply,
Process 4 terminates its election

Processes 5 and 6 simultaneously hold an election

Process 6 replies to 5
Process 5 terminates its election

Noone replied to the election of process 6, thus, this process wins the election and communicates the result to all others

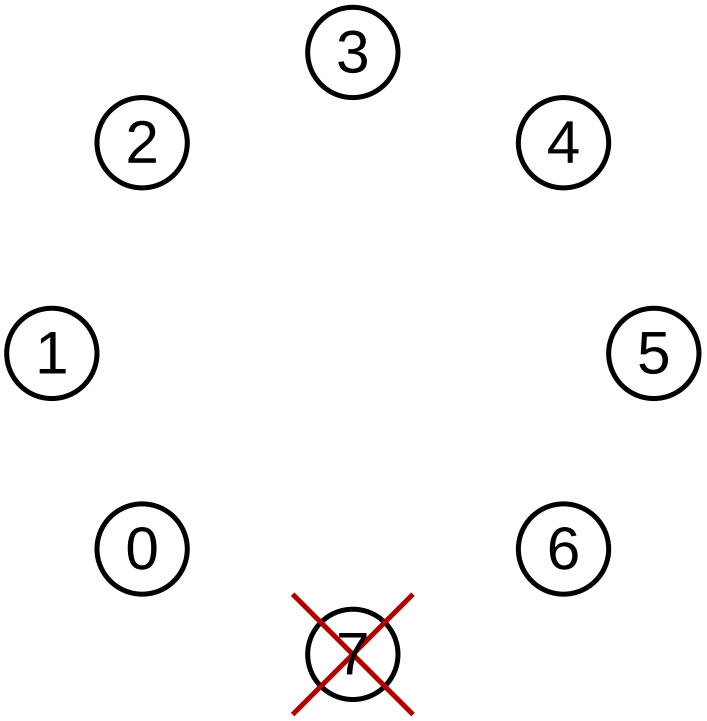
A Ring Algorithm

- Assumption: processes form a logical ring, i.e. each process knows its successors in the ring
- Messages are sent along the ring as follows:
  - a process tries to send the message to its direct successor
  - if this process is not active, the message will be sent to the next process in the ring, etc.
- ELECTION messages contain a list of process IDs

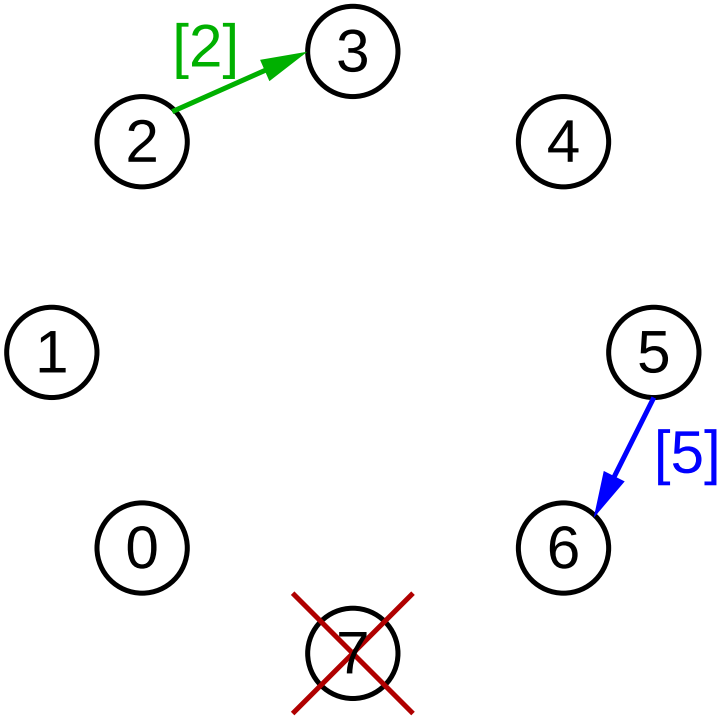
A Ring Algorithm ...

- A process that initiates the election sends an ELECTION message with its own ID along the ring
- When an ELECTION message is received by a process:
  - if its own ID is not in the list of IDs:
    - append the own ID to the list
    - continue sending message along the ring
  - else (message came back to the initiator):
    - determine highest ID in the list
    - send this ID in a COORDINATOR message along the ring

Ring Algorithm: Example

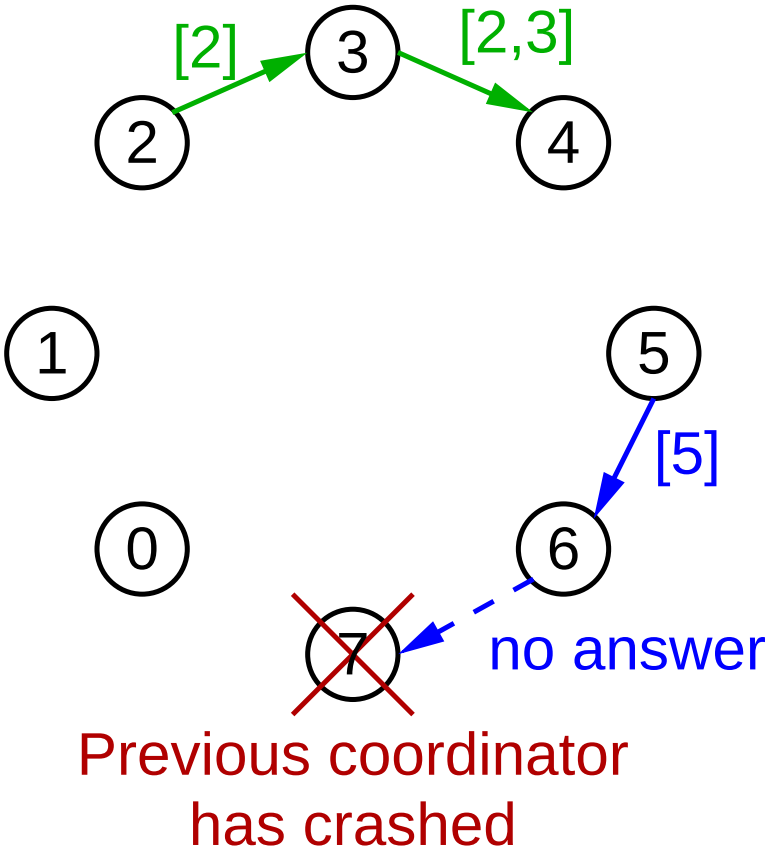


Ring Algorithm: Example



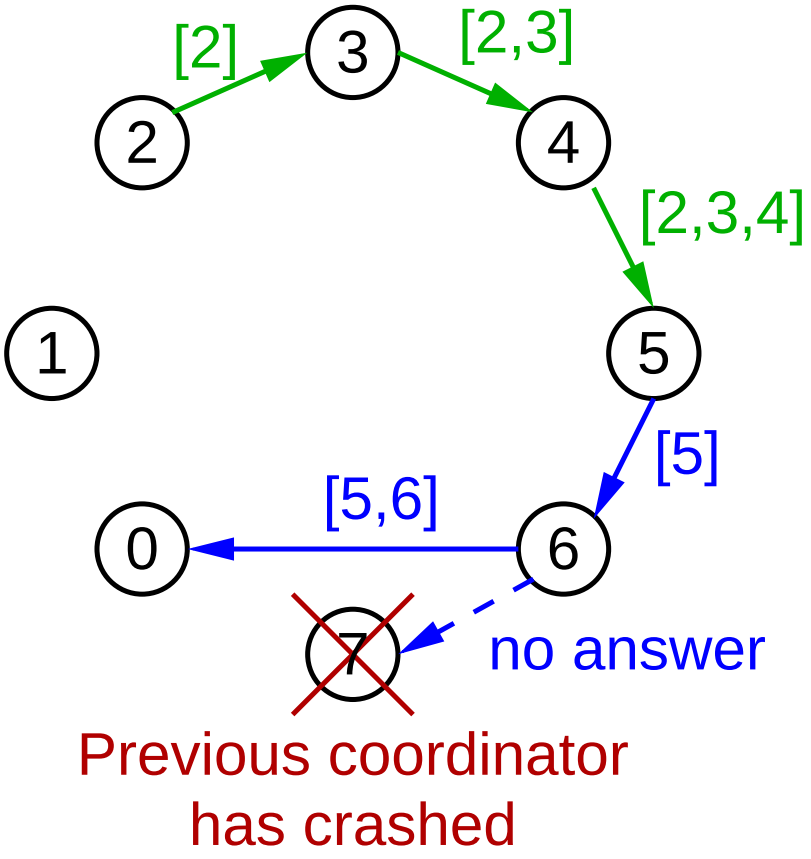
Processes 2 and 5 concurrently initiate an election

Ring Algorithm: Example



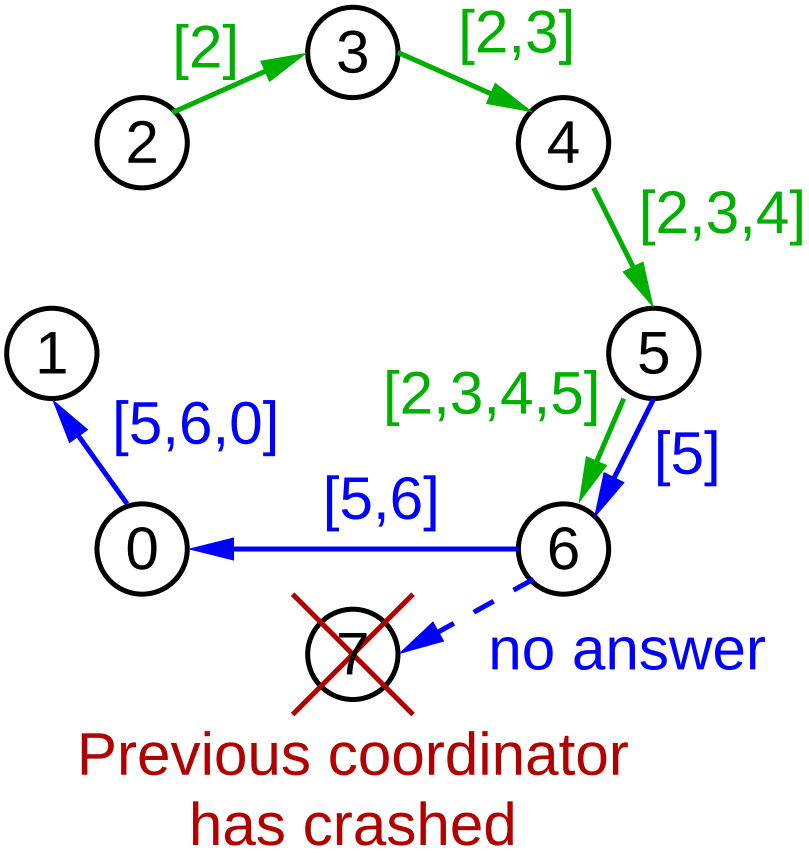
Processes 2 and 5 concurrently initiate an election

Ring Algorithm: Example



Processes 2 and 5 concurrently initiate an election

Ring Algorithm: Example



Processes 2 and 5 concurrently initiate an election

Eventually both processes get their ELECTION messages back and send a COORDINATOR message (with identical contents!)

7.2 Mutual Exclusion

- Here mainly: use / allocation of exclusive resources
- Requirements:
  - safety: only one process can use the resource at any one time
  - liveness: any process that requests the resource will eventually get it
  - fairness: access to resources in 'FIFO' order
- Solution approaches:
  - centralized server
  - distributed algorithm with Lamport clock
  - token ring algorithm

Centralized server

- An special coordinator process manages the resource and a queue for waiting processes
  - determined e.g. via an election algorithm
- Resource is requested by sending a message to the coordinator
  - if resource is free: coordinator answers with OK
  - otherwise: coordinator does not answer
    - requesting process is blocked (waiting for reply)
- Resource is released by sending a message to the coordinator
  - if processes wait: coordinator sends an OK to one of them
- Problem: processes cannot detect failure of the coordinator
  - this could be done using negative replies and polling

A Distributed Algorithm (Ricart / Agrawala)

- Idea: a process that wants to have a resource asks all other processes for their OK
  - a process replies with OK, if
    - it does not want the resource, or
    - it wants the resource, but the other process has requested it “earlier”
- Requires **total** order of request events
  - order must be consistent with causality
  - realizable e.g. via a time stamp (Lamport time, process ID) with lexicographic order
    - in the example of slide 207 this results in the event order: $b, c, a, e, g, d, j, f, l, h, i, k$

A Distributed Algorithm (Ricart / Agrawala) ...

- To request a resource, a process sends the following message to all other processes:
  - resource ID
  - time stamp T of the request
    - pair: (current Lamport time, own process ID)

  (the message must be delivered reliably)
- The process then waits until it receives an OK message from all other processes
- After that it can use the resource (exclusively)

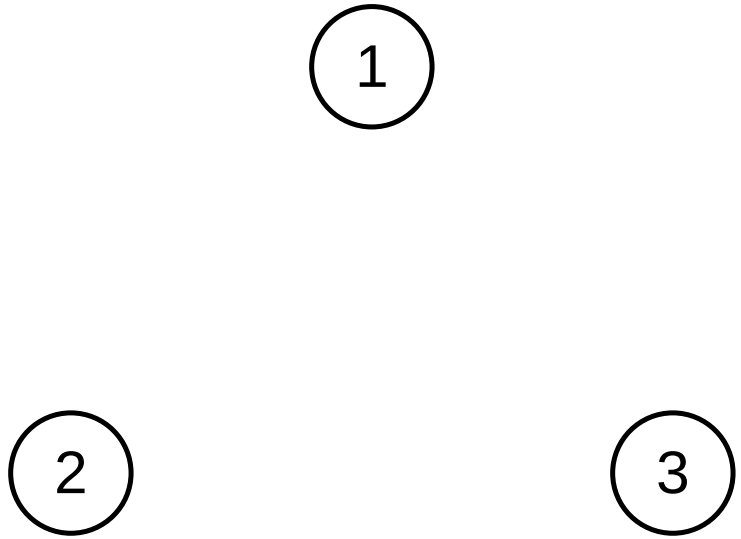
A Distributed Algorithm (Ricart / Agrawala) ...

- Each process responds to request messages as follows:
  - resource is not used and not requested by the process:
    - return OK message
  - resource is used by the process:
    - do not send a reply
    - put the request in a queue
  - Resource id not used, but requested by the process:
    - if T(incoming message) $<$ T(own request):
      - return OK message
    - or else:
      - do not send a reply
      - put the request in a queue

A Distributed Algorithm (Ricart / Agrawala) ...

- When a process releases the resource:
  - send an OK message to **all** processes in the queue
  - delete the queue

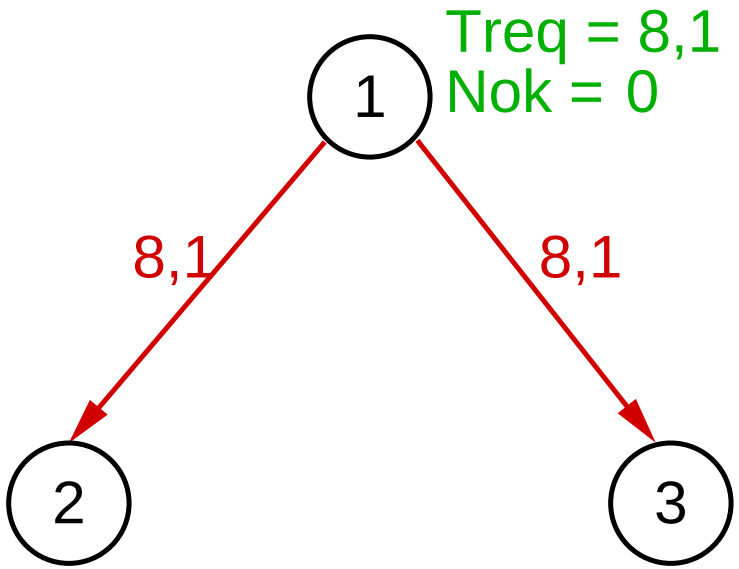
Example for the Algorithm of Ricart / Agrawala



Both P1 and P3 want
the resource

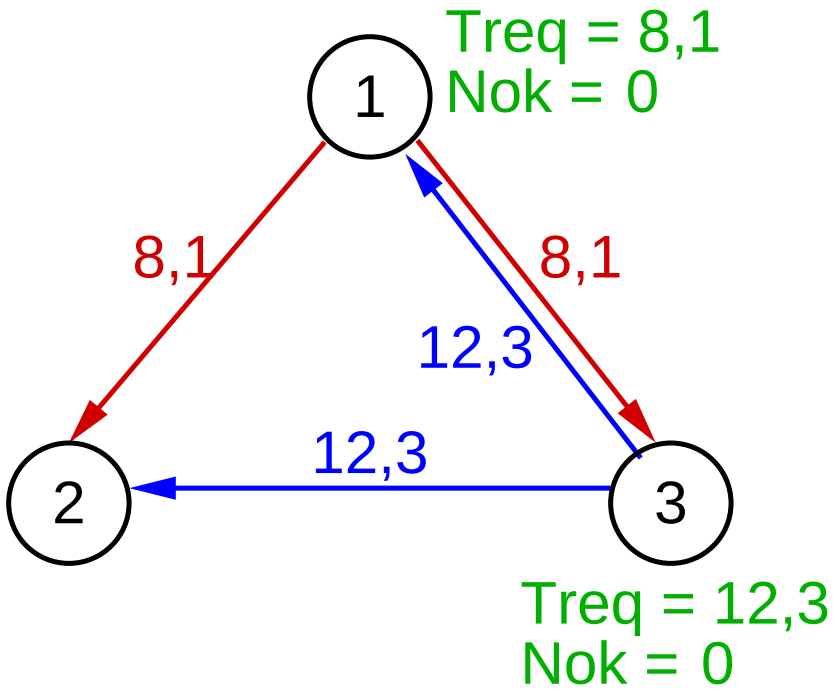
Example for the Algorithm of Ricart / Agrawala

1. P1 sends request to all others



Both P1 and P3 want the resource

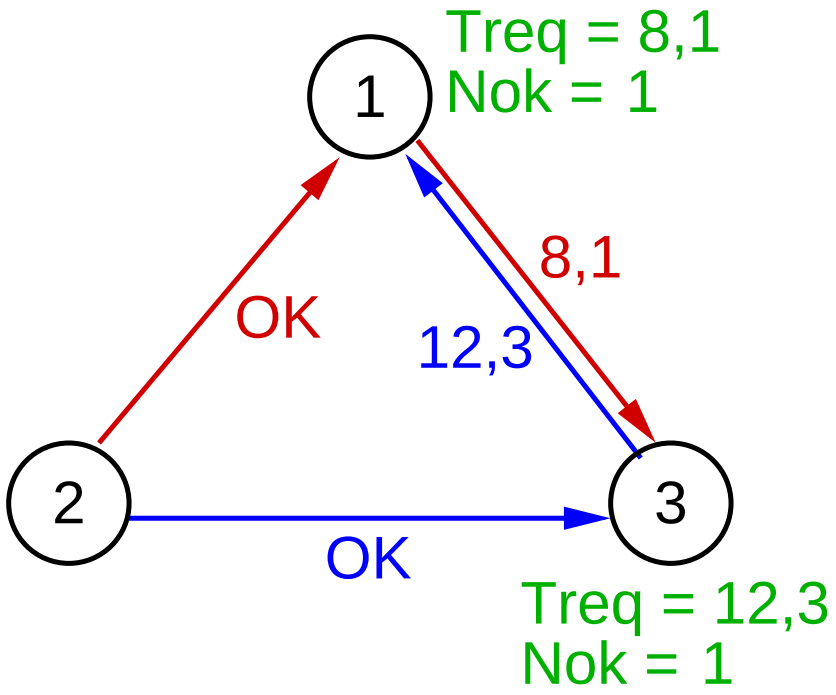
Example for the Algorithm of Ricart / Agrawala



Both P1 and P3 want the resource

1. P1 sends request to all others
2. P3 sends request to all others

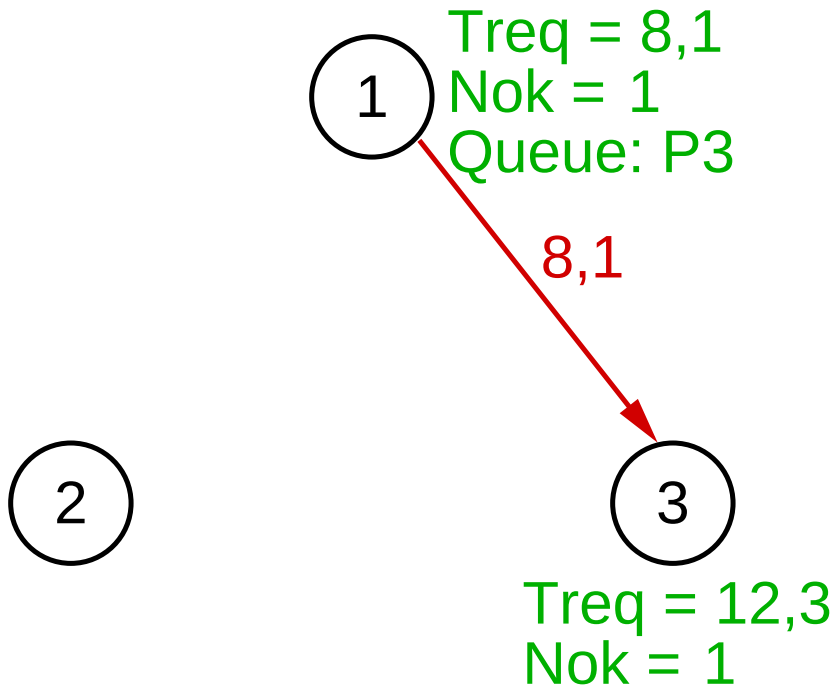
Example for the Algorithm of Ricart / Agrawala



Both P1 and P3 want the resource

1. P1 sends request to all others
2. P3 sends request to all others
3. P2 sends OK to P1 and P3, since it doesn,t want the resource

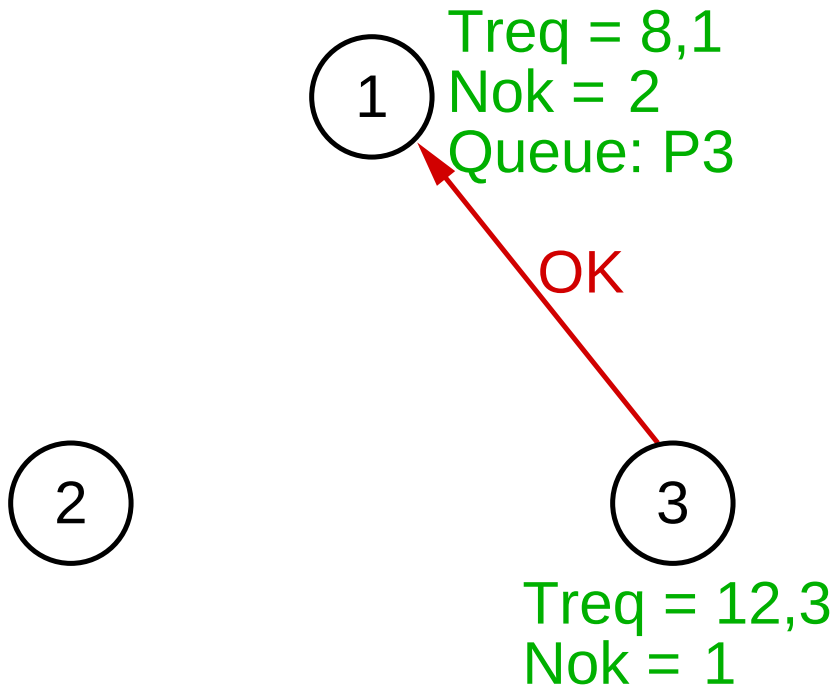
Example for the Algorithm of Ricart / Agrawala



Both P1 and P3 want the resource

1. P1 sends request to all others
2. P3 sends request to all others
3. P2 sends OK to P1 and P3, since it doesn,t want the resource
4. P1 doesn’t send an OK to P3, since (12,3) > (8,1). P1 adds P3 to its queue

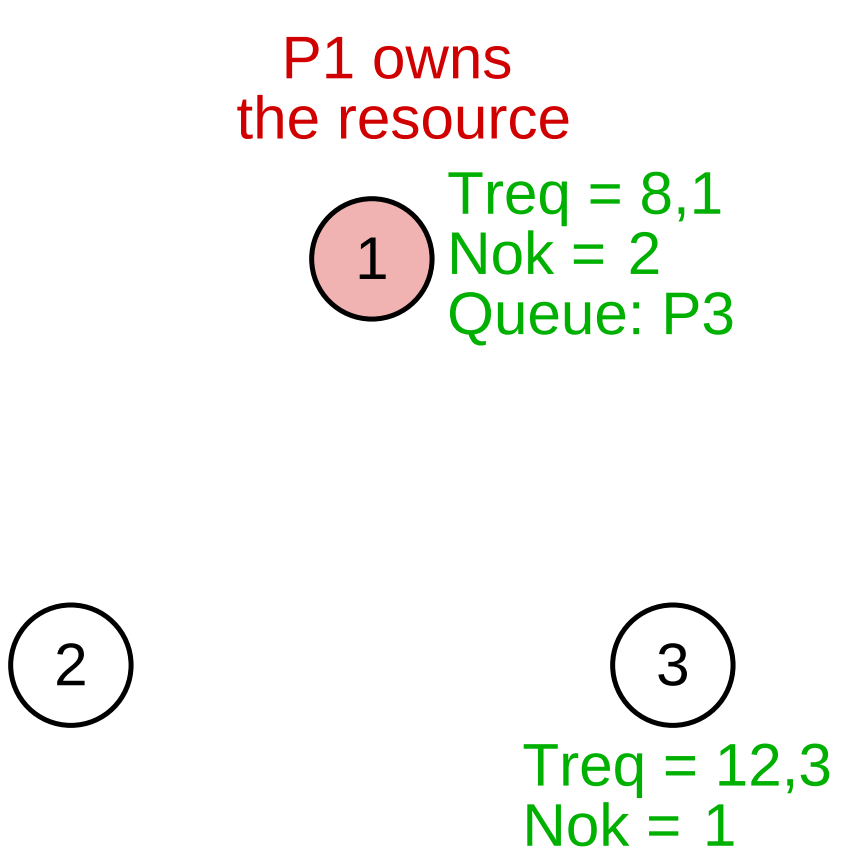
Example for the Algorithm of Ricart / Agrawala



Both P1 and P3 want the resource

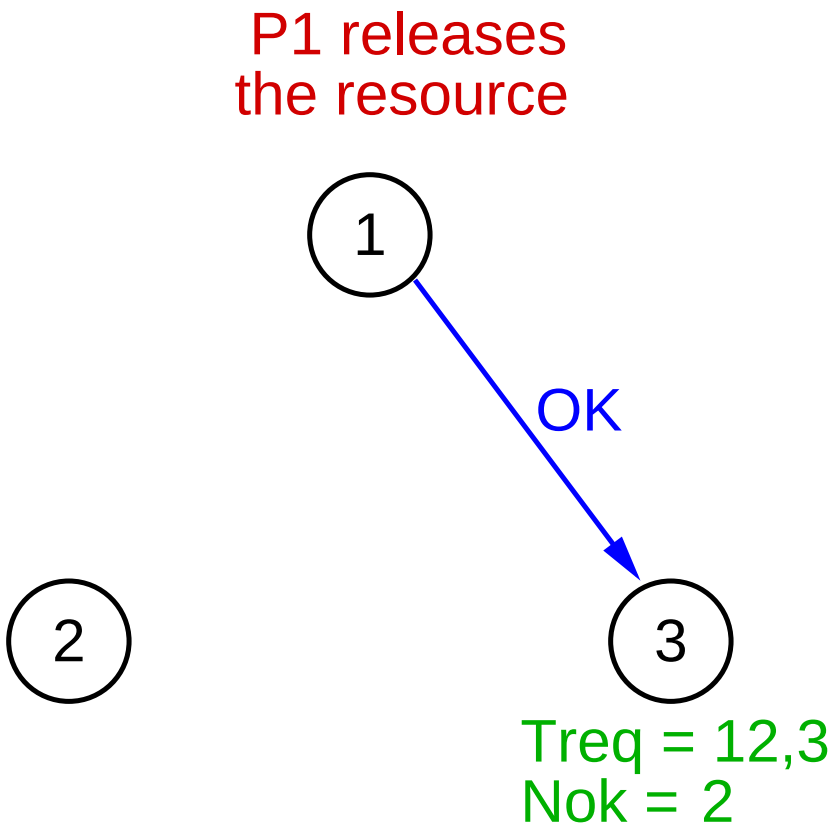
1. P1 sends request to all others
2. P3 sends request to all others
3. P2 sends OK to P1 and P3, since it doesn,t want the resource
4. P1 doesn’t send an OK to P3, since (12,3) > (8,1). P1 adds P3 to its queue
5. P3 sends OK to P1, since (8,1) < (12,3)

Example for the Algorithm of Ricart / Agrawala



1. P1 sends request to all others
2. P3 sends request to all others
3. P2 sends OK to P1 and P3, since it doesn,t want the resource
4. P1 doesn't send an OK to P3, since (12,3) > (8,1). P1 adds P3 to its queue
5. P3 sends OK to P1, since (8,1) < (12,3)
6. P1 received all OKs and uses the resource

Example for the Algorithm of Ricart / Agrawala



Both P1 and P3 want
the resource

1. P1 sends request to all others
2. P3 sends request to all others
3. P2 sends OK to P1 and P3,
   since it doesn,t want the resource
4. P1 doesn't send an OK to P3,
   since (12,3) > (8,1).
   P1 adds P3 to its queue
5. P3 sends OK to P1, since
   (8,1) < (12,3)
6. P1 received all OKs
   and uses the resource
7. P1 releases the resource
   and sends an OK to P3

A Token Ring Algorithm

- The processes form a logical ring
- A **token** circles in the ring
  - authorization for (exclusive) use of the resource
  - token is initially generated by one of the processes
- On arrival of the token: process checks whether it wants the resource
  - if so:
    - use the resource
    - after releasing the resource:
      - pass token to successor in the ring
  - else:
    - pass token immediately to successor in the ring

Comparison of algorithms

- Centralized server:
  - server is *single point of failure* and may be a performance bottleneck
  - clients cannot distinguish (without additional measures) between server failure and occupied resource
  - only little communication necessary
- Distributed algorithm:
  - failure of **any** node is problematic
  - any node can become a performance bottleneck
  - high communication effort
  - just a proof that a distributed, symmetrical algorithm is possible

Comparison of algorithms ...

- Token ring algorithm:
  - problem: loss of the token (detection, re-creation)
  - failure of nodes is problematic
  - communication, even if resource is not used

Algorithm	Messages per allocation	Delay before allocation	Problems
centralized	3	2	server failure
distributed	$2(n-1)$	$2(n-1)$	failure of any process
token ring	$1 \ldots \infty$	$0 \ldots n-1$	lost token, failure of any process

7.3 Group Communication (Multicast)

[Coulouris, 4.5, 11.4]

- In distributed systems, communication with a **group** of processes (**multicast**) is often also important, e.g. for:
  - fault tolerance based on replicated services
    - service realized by group of servers
    - all servers receive and process the requests
  - finding of services (especially discovery / name services)
    - multicast is a possible approach for this
  - better performance through replicated data
    - changes must be sent to all copies
  - sending event notifications
    - all subscribers receive the event

Questions / Problems

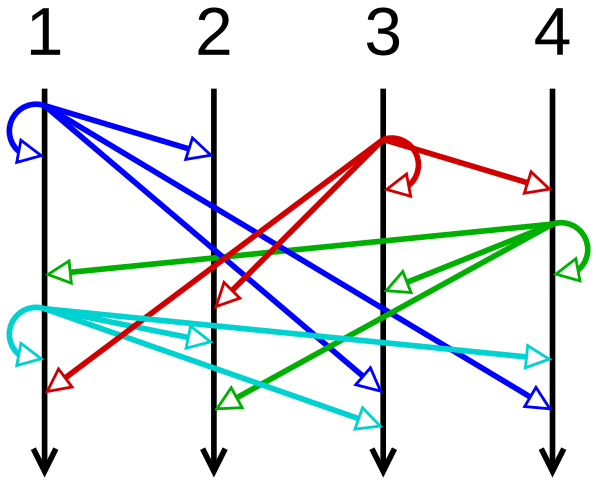
- Addressing the recipients
  - explicit list of all recipients
  - addressing a process group
    - static / dynamic groups
- Reliability
  - reasonable guarantees that messages will reach their recipients
- Order
  - adequate guarantees as to the order in which multicast messages arrive at the various recipients

Reliability

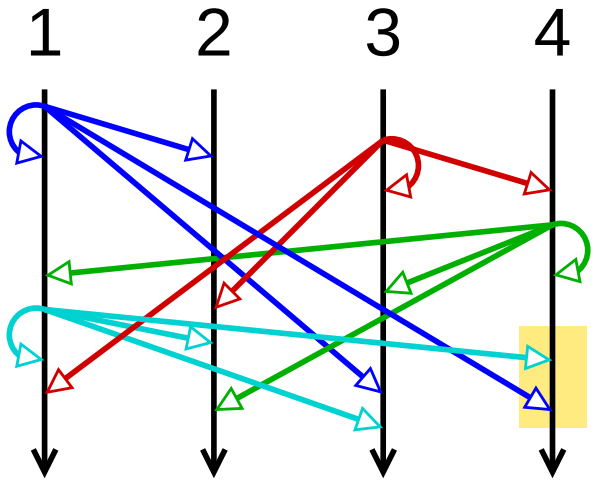
- **Unreliable multicast**:
  - some processes may not receive the message (e.g. due to packet loss)
- **Reliable multicast**:
  - apart from network and process failures, the message is delivered to all processes in the group
- **Atomic multicast**:
  - the message is (under all circumstances) received either by **all** processes of the group or by **none** of them
  - required if all processes in the group must be kept consistent (e.g., operations on replicated data)

Order

- **Unordered**
  - receiving order is undefined and can be different in different processes

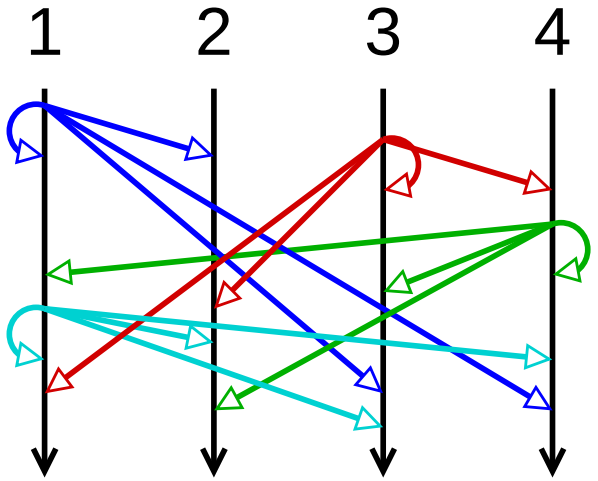


- **FIFO order**
  - messages from the **same** sender are received by all processes in FIFO order
  - i.e. introduction of sequence numbers **local to the sender**

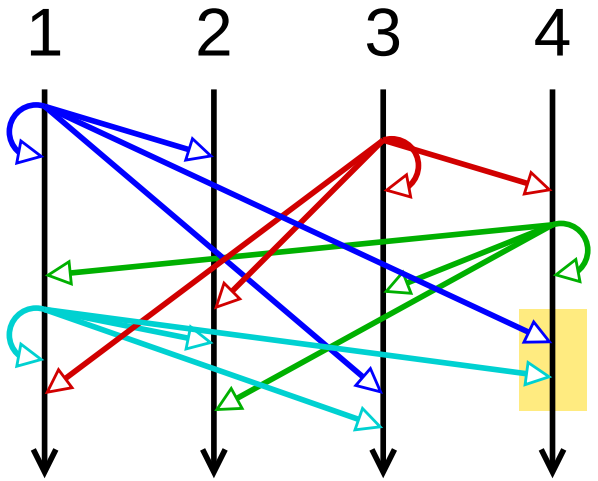


Order

- **Unordered**
  - receiving order is undefined and can be different in different processes

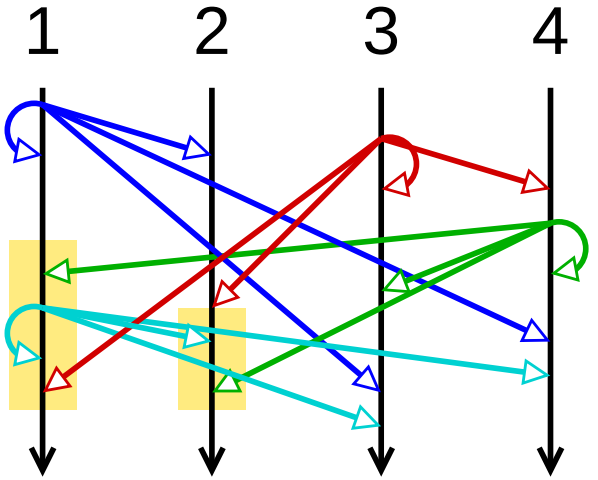


- **FIFO order**
  - messages from the **same** sender are received by all processes in FIFO order
  - i.e. introduction of sequence numbers **local to the sender**

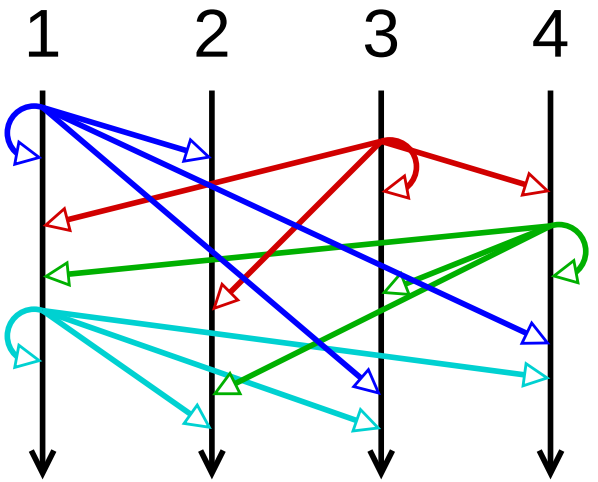


Order ...

- **Causal order**
  - if message m' can causally depend on m ($m \rightarrow m'$), then all processes receive m before m'
  - i.e. introduction of vector time stamps

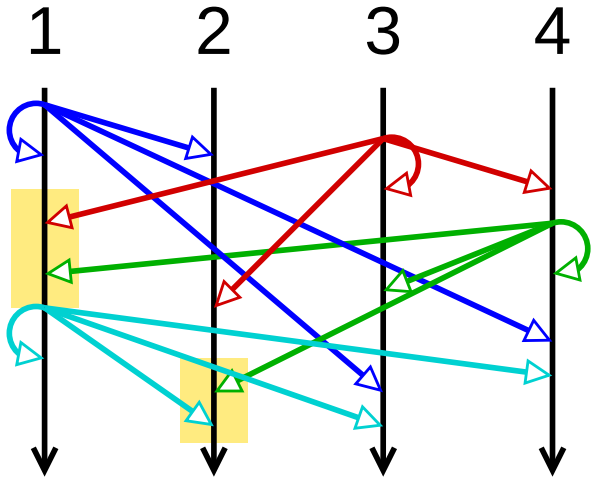


- **Total order**
  - **all** messages are received by all processes in the same order
  - i.e. introduction of **global** sequence numbers

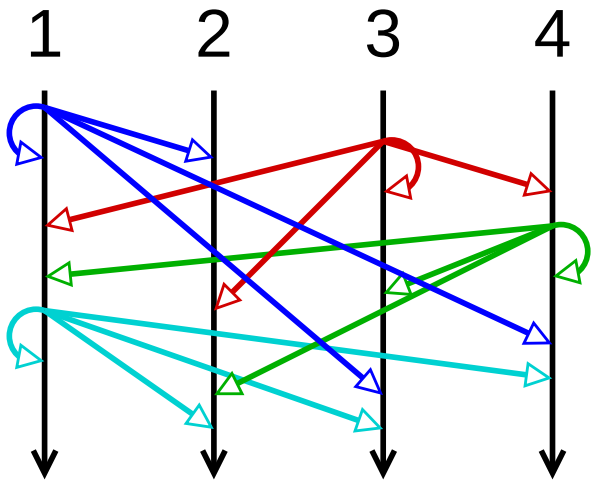


Order ...

- **Causal order**
  - if message m' can causally depend on m ($m \rightarrow m'$), then all processes receive m before m'
  - i.e. introduction of vector time stamps

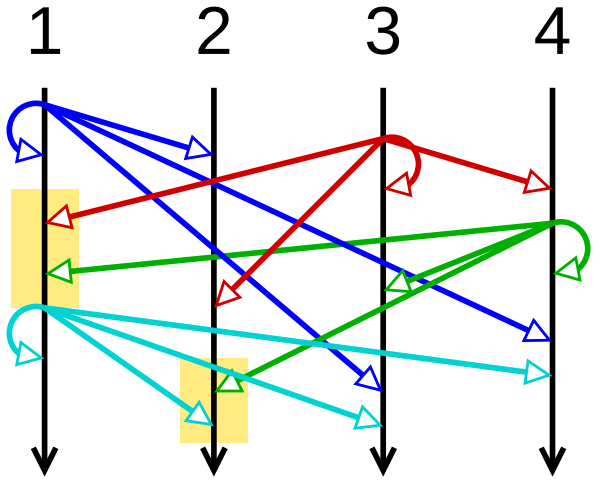


- **Total order**
  - **all** messages are received by all processes in the same order
  - i.e. introduction of **global** sequence numbers

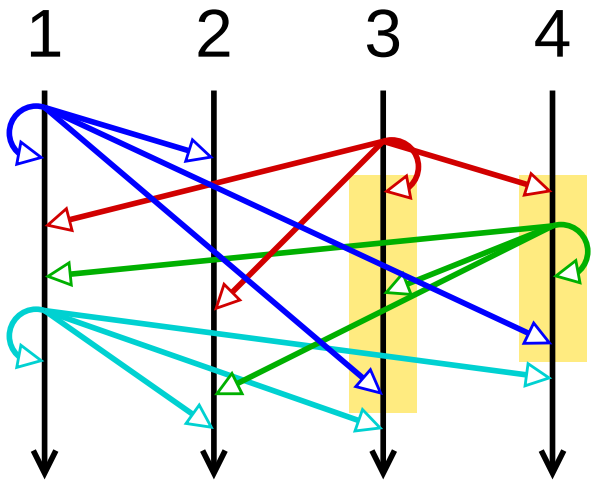


Order ...

- **Causal order**
  - if message m' can causally depend on m ($m \rightarrow m'$), then all processes receive m before m'
  - i.e. introduction of vector time stamps

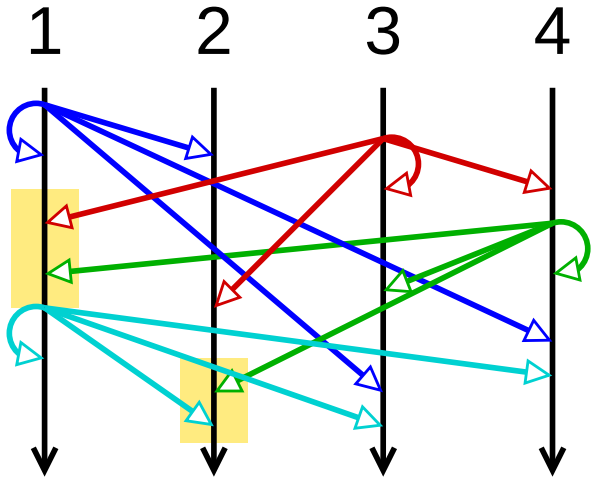


- **Total order**
  - **all** messages are received by all processes in the same order
  - i.e. introduction of **global** sequence numbers

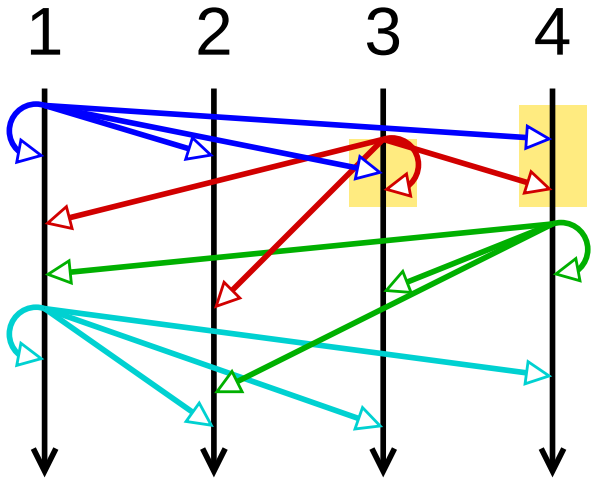


Order ...

- **Causal order**
  - if message m' can causally depend on m ($m \rightarrow m'$), then all processes receive m before m'
  - i.e. introduction of vector time stamps

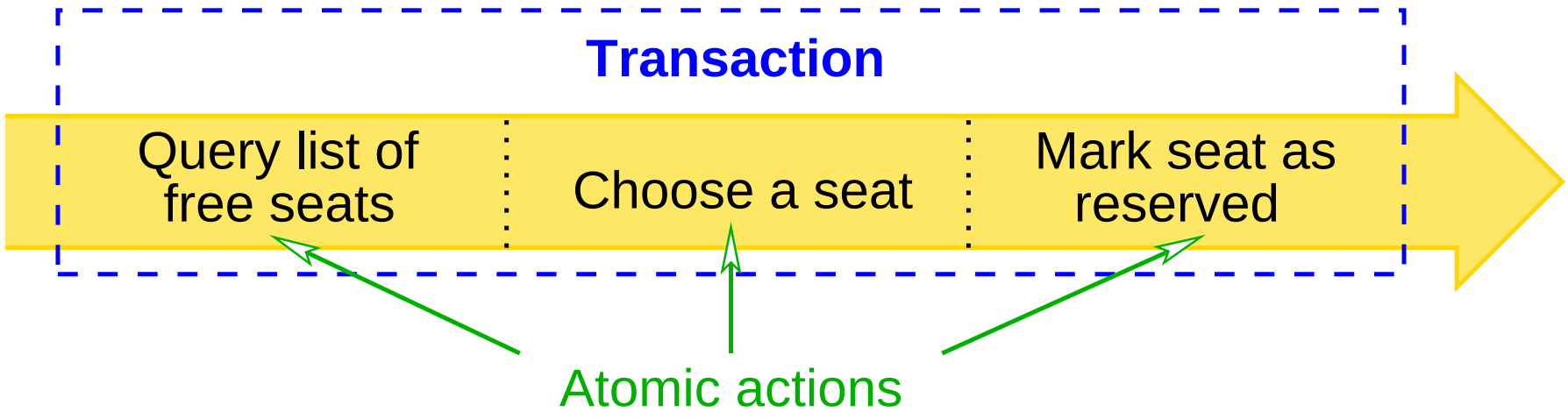


- **Total order**
  - **all** messages are received by all processes in the same order
  - i.e. introduction of **global** sequence numbers



7.4 Transactions

- Combining a sequence of atomic actions into a single unit
  - atomic actions: read, change, write data
- Example: seat reservation

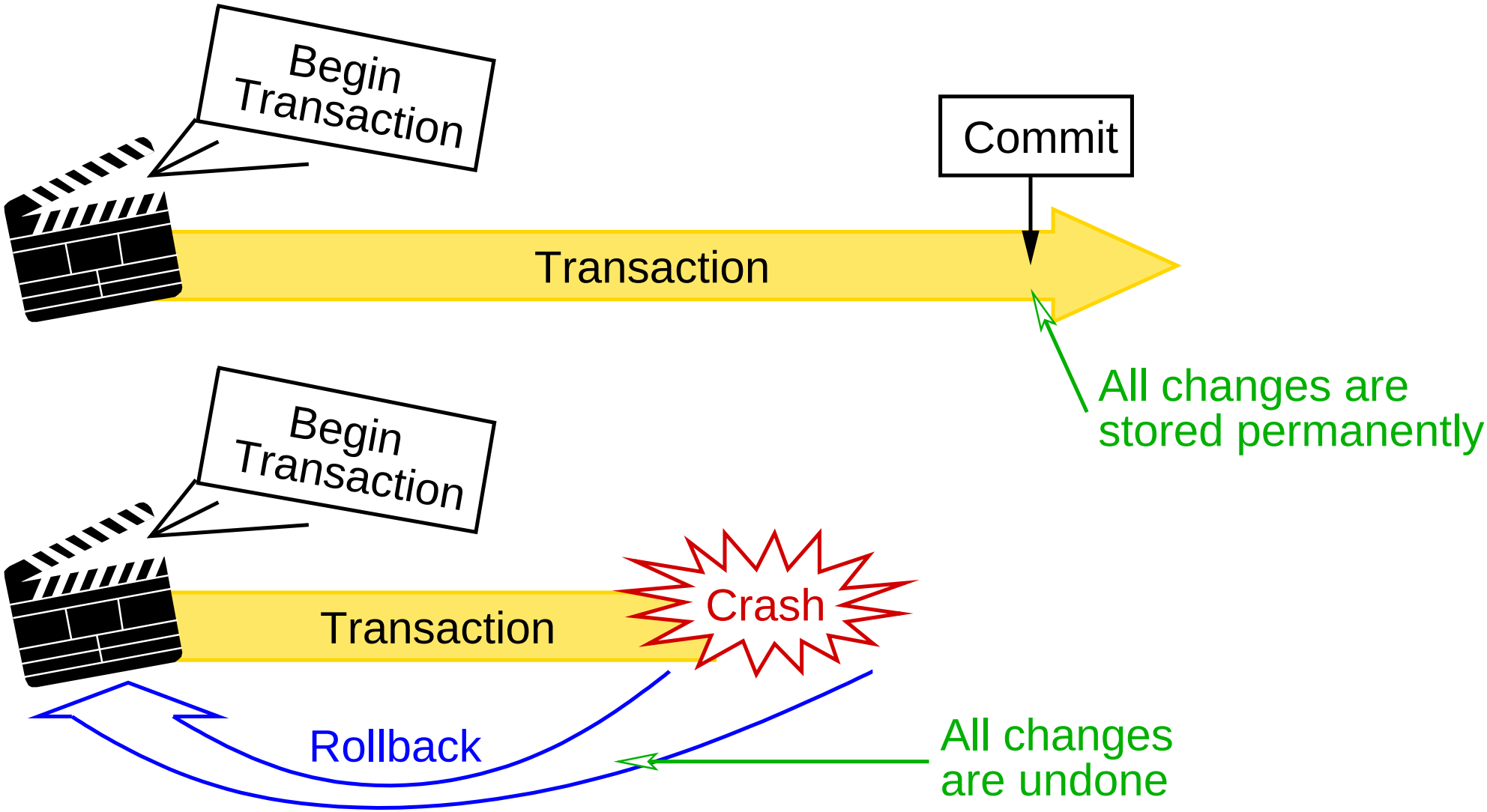


- Used not only in database systems

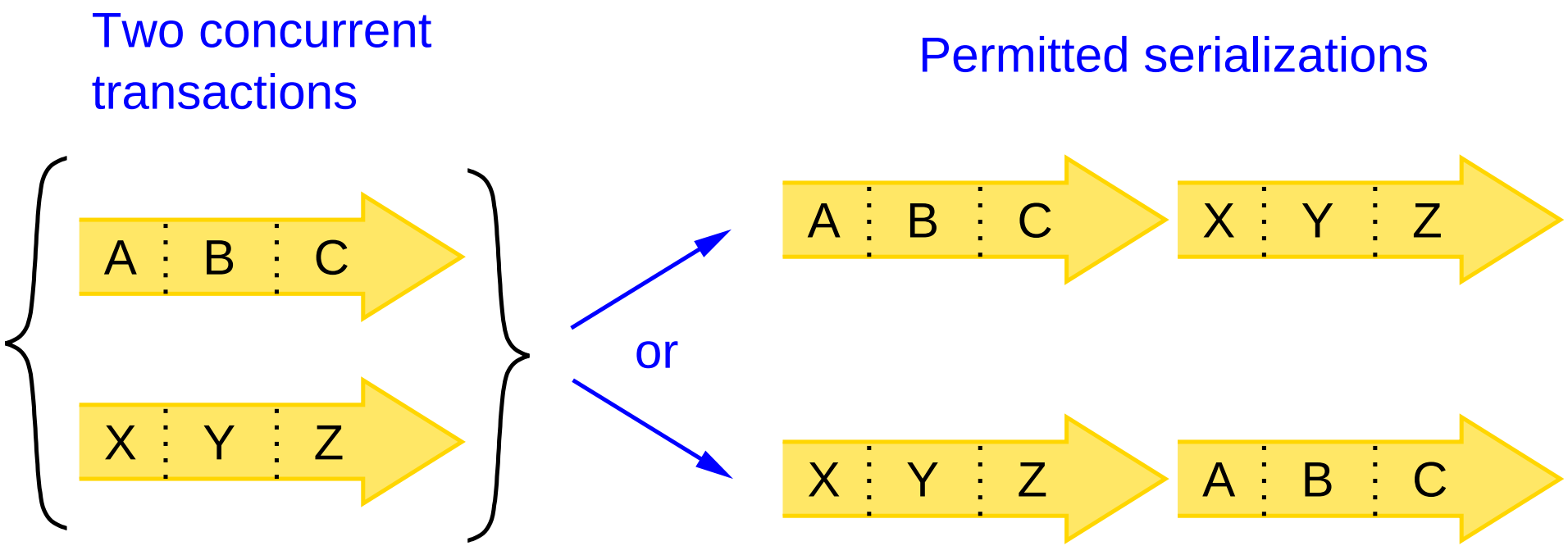
Properties of Transactions: ACID

- **Atomicity**
  - all-or-nothing principle: either all atomic actions are executed (correctly) or none at all
- **Consistency**
  - a transaction always transfers a consistent state back to consistent state
- **Isolation**
  - concurrent transactions do not affect each other; the result is the same as with sequential execution
- **Durability**
  - at the (successful) end of the transaction all changes are stored permanently

Atomicity


Begin Transaction
Commit
Transaction
All changes are stored permanently
Begin Transaction
Transaction
Crash
Rollback
All changes are undone

Isolation



- The result of the concurrent transactions corresponds to one of the two serializations

Isolation Levels

- Complete isolation of (database) transactions often is too restrictive / too little performant
- Therefore: SQL99 standard defines four isolation levels
- Goal: avoidance of unwanted phenomena
  - ***dirty reads:*** a transaction can read data of another transaction before they have been committed
  - ***unrepeatable reads:*** when reading repeatedly, a transaction can see commited changes of other transactions
  - ***phantom reads:*** when reading repeatedly, a transaction can see that other transactions have added or deleted records

Isolation Level According to ANSI/ISO-SQL99

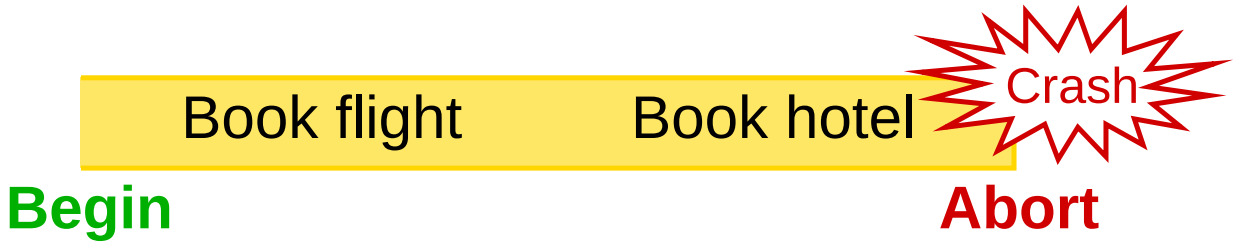
Isolation level \ Phenomenon	Dirty Reads	Unrepeatable Reads	Phantom Reads
Read Uncommitted	possible	possible	possible
Read Committed	not possible	possible	possible
Repeatable Read	not possible	not possible	possible
Serializable	not possible	not possible	not possible

➥ *Serializable* corresponds to complete isolation

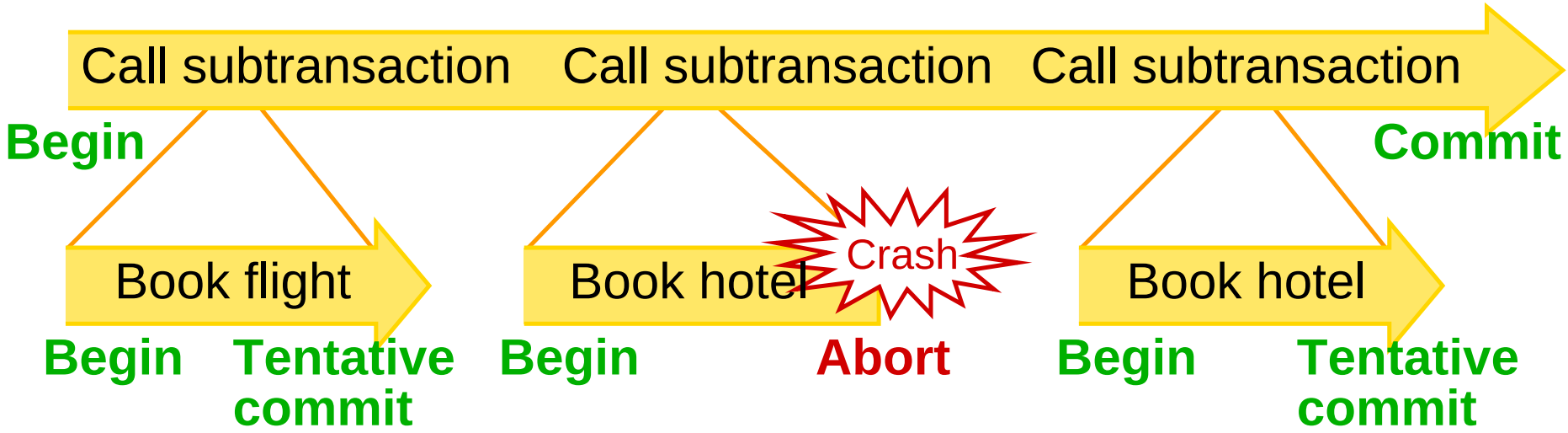
Nested Transactions

- Within a transaction, several subtransactions take place
- Higher-level transaction can run successfully to completion, even if subtransaction was terminated with an error
- Abort of the higher-level transaction results in aborting all subtransactions
- Example: booking of flight and hotel
  - booking of the flight should be maintained, even if hotel booking (in the first attempt) fails
- Nested transaction are supported by only a few transaction services

Flat transaction



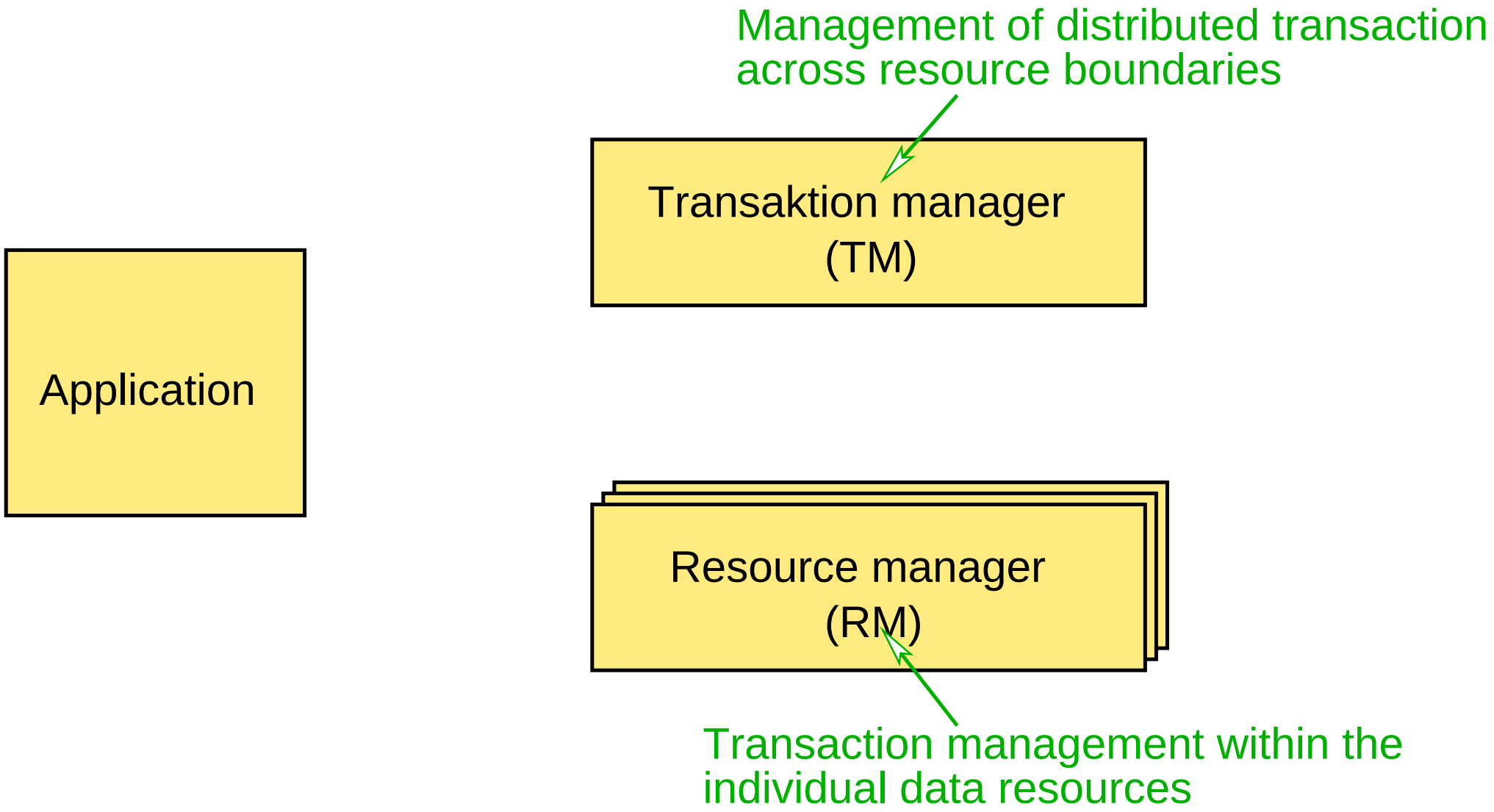
Nested transactions



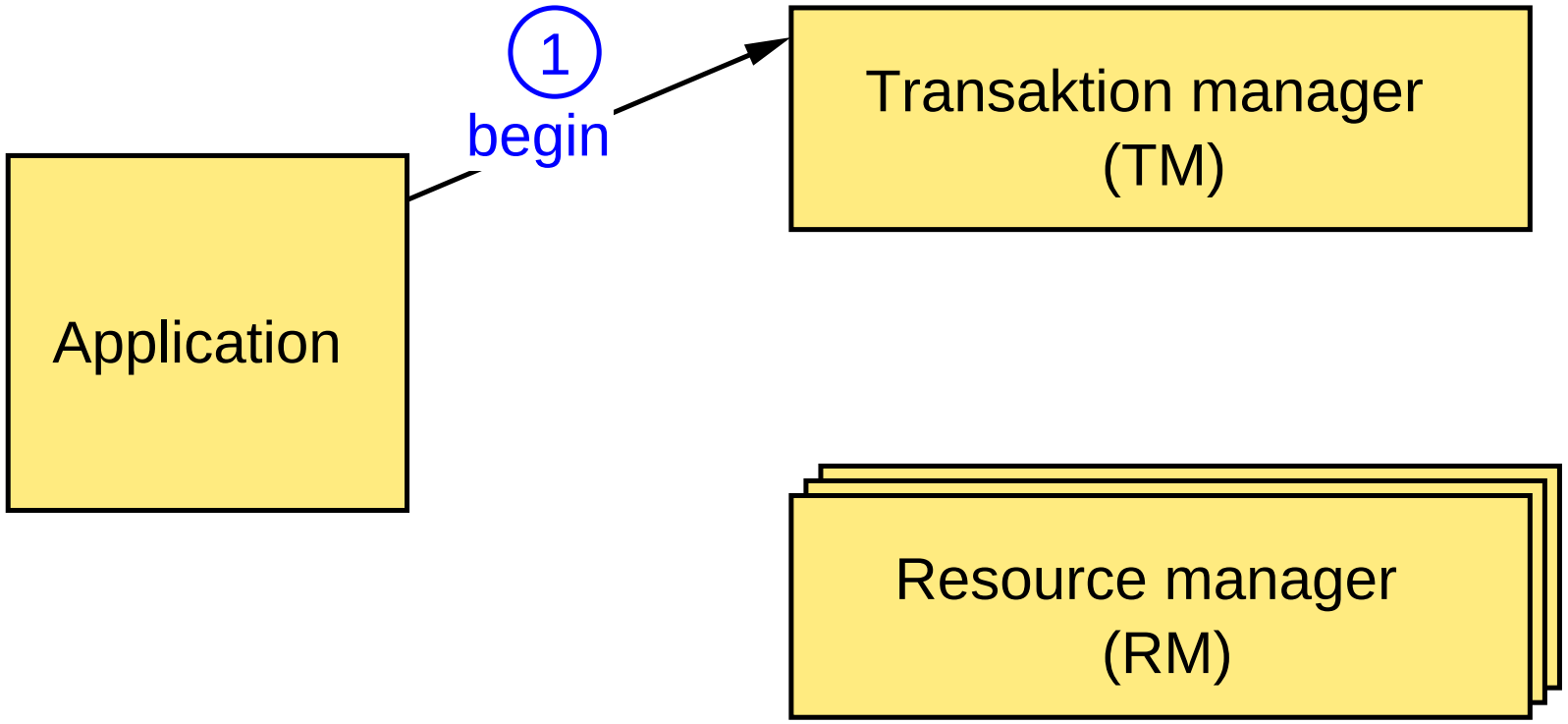
Distributed Transactions

- So far: data is stored at exactly one location
- Distributed transactions: data is stored distributed
- Realization of transactions on the individual data resources (databases) is no longer sufficient
  - distributed transaction management becomes necessary
- There is a generally accepted Open Group model for the management of distributed transactions
  - is implemented by most transaction services
  - most important feature: 2-Phase-Commit

Model for Managing Distributed Transactions

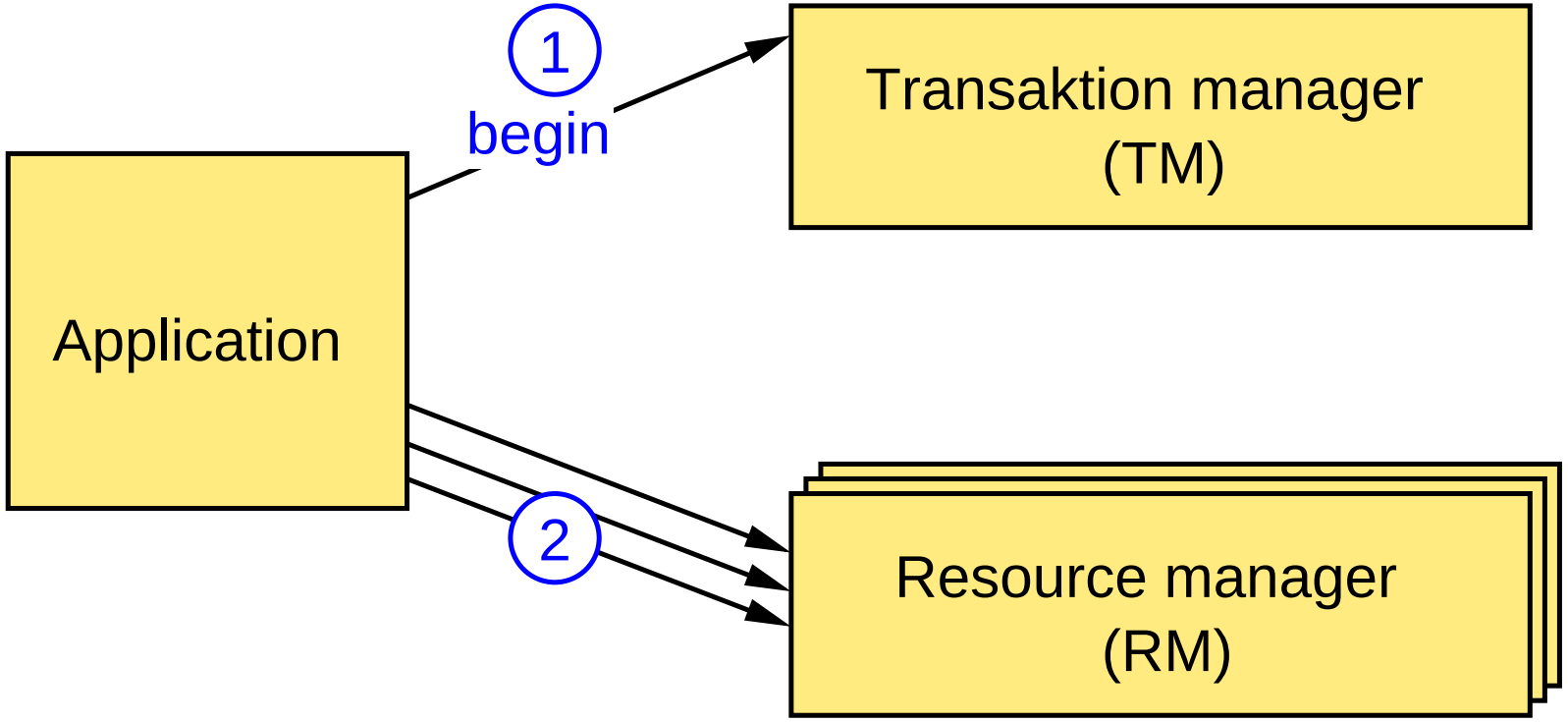


Model for Managing Distributed Transactions



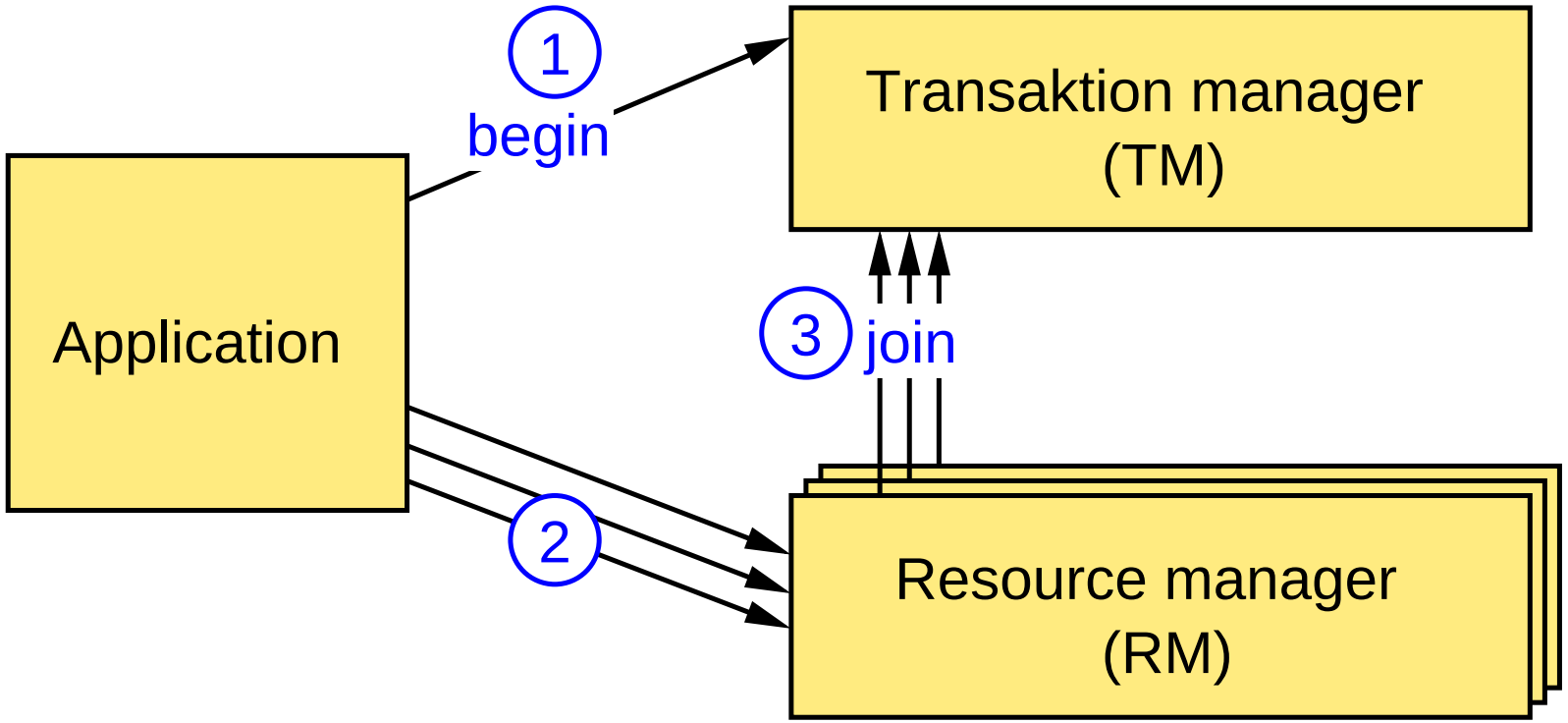
1. Application requests start of a new transaction.
   TM internally initializes a new transaction.

Model for Managing Distributed Transactions



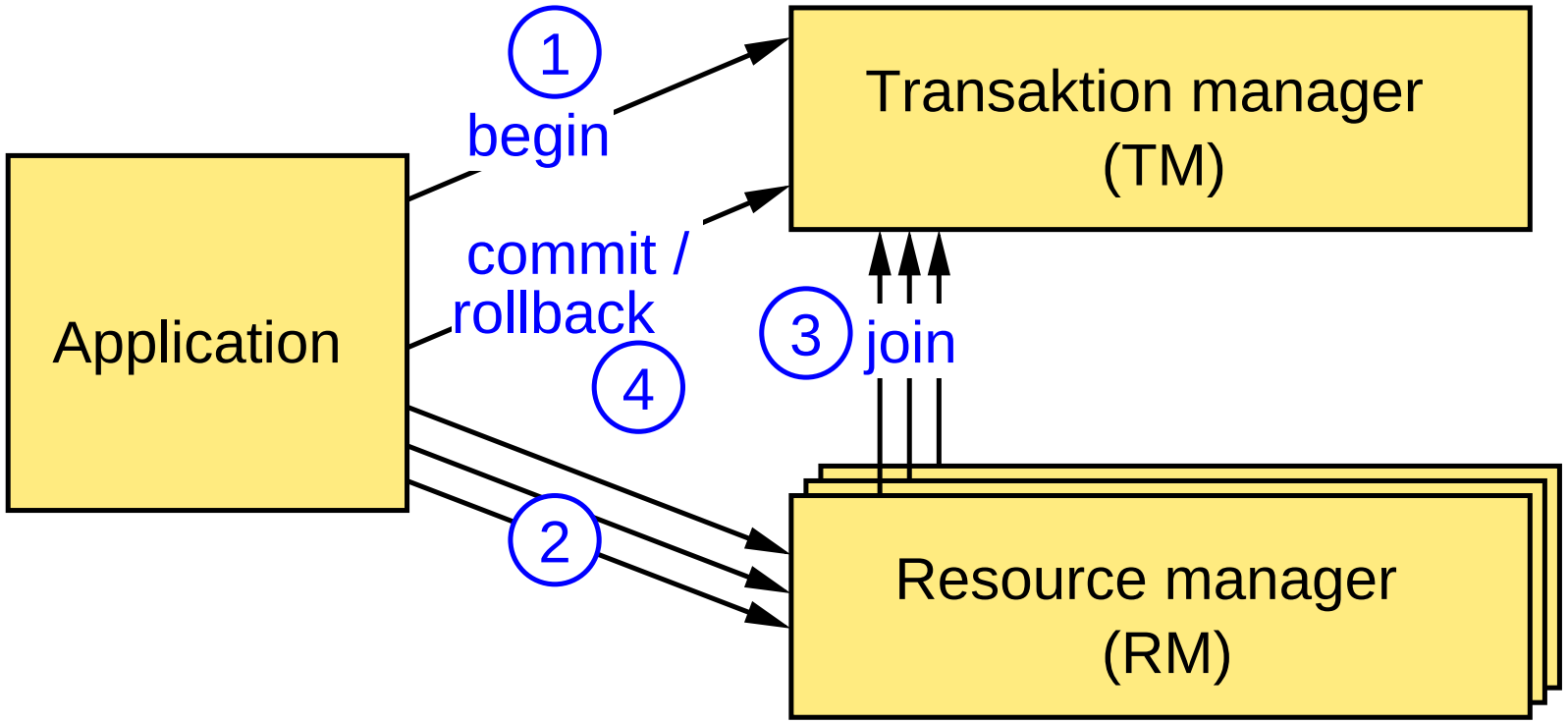
2. Transaction is active.
   Application can use the resources.

Model for Managing Distributed Transactions



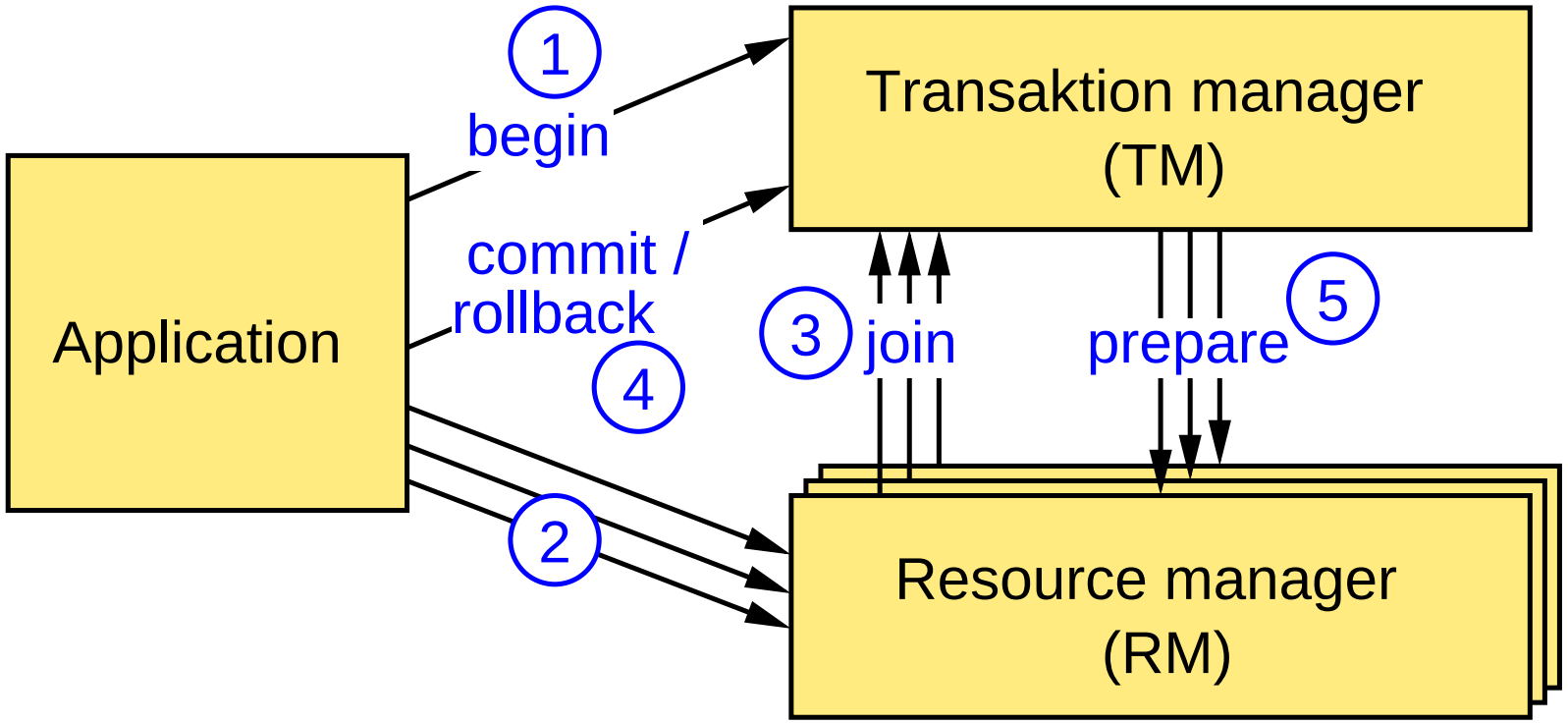
3. Each RM used by the applilcation
registers with the TM for the transactio.n

Model for Managing Distributed Transactions



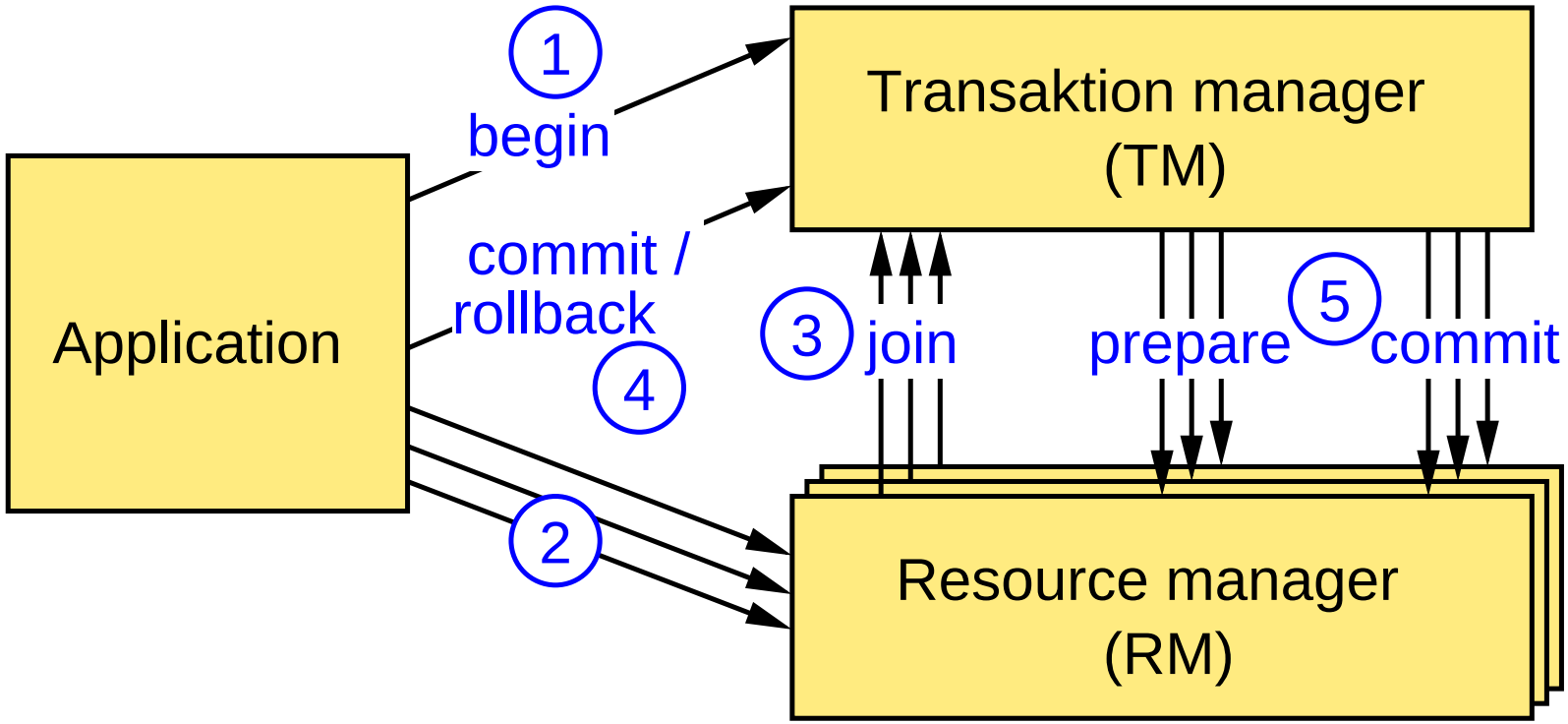
4. Aplication requires to commit or abort the transaction.

Model for Managing Distributed Transactions



5. TM requires RM to commit the changes:
   2−phase commit

Model for Managing Distributed Transactions



5. TM requires RM to commit the changes:
   2−phase commit

2-Phase Commit

- Phase 1 (voting phase)
  - TM asks all involved RM, if the commit would be successful (“*prepare*”)
  - each RM that answers “yes” prepares for the commit
- Phase 2 (finalization)
  - if all RMs answered with “yes”:
    - TM sends *commit* command to all RMs
    - RM ultimately commits the data and sends an acknowlegement to TM
  - else:
    - TM sends an *abort* command to all RMs