Distributed Algorithms

Distributed Transactions

Overview

Motivation for Transactions

Background

- ACID-Properties
- Locking-based Concurrency Control

Distributed Transactions

- Two-Phase Commit

Motivation for Transactions

Critical sections (mutual exclusion)

- used to achieve consistency in distributed systems
- manually applied by the developer
- complicated and error-prone (e.g., risk of deadlocks)

Rather needed: high-level concept automatically ensuring consistency even in the face of failures

→ **transactions**

Transactions

Important concept in databases and distributed systems

Atomic execution of a set of instructions

- e.g., bank transfer: debit source account and deposit destination account

Completing a transaction

- **Commit**: Transaction is successfully completed
  - Final state is stored ***persistently*** and then becomes visible outside of the transaction
- **Abort**: Transaction is aborted
  - **Rollback** to initial state, i.e., the effects of the transaction are undone

ACID-Properties

Atomicity	All-or-Nothing
Consistency	Transition from one *consistent* state to another *consistent* state
Isolation	Intermediate states are not visible outside the transaction's boundary
Durability	The final state is stored persistently and is not lost even in case of later failure

Concurrency Control and Recovery

The ACID-properties are endangered by

interfering concurrent transactions and by **faulty environments**

Atomicity

Consistency

Isolation

Durability

Concurrency Control

Recovery

Concurrency Control

- Supervision of simultaneously executing transactions
- Each TX is modeled as a sequence of *read* and *write* operations (called **schedule**) on individual *data items* followed by either *commit* or *abort*

- A schedule is
  - **serializable** iff it is equivalent to a serial schedule
  - **recoverable** iff any TX is only committed after all TXs from which the TX has *read uncommitted* data have been committed
  - **avoiding cascading aborts** iff no TX *reads uncommitted* data
  - **strict** iff no TX *reads or overwrites uncommitted* data

Concurrency Control

– ***Serializable*** iff it is equivalent to a serial schedule

$$A = \begin{bmatrix} T1 & T2 & T3 \\ R(X) & & \\ & R(Y) & \\ & & R(Z) \\ W(X) & & \\ & W(Y) & \\ & & W(Z) \\ Com. & & \\ & Com. & \\ & & Com. \end{bmatrix} \qquad B = \begin{bmatrix} T1 & T2 & T3 \\ R(X) & & \\ W(X) & & \\ Com. & & \\ & R(Y) & \\ & W(Y) & \\ & Com. & \\ & & R(Z) \\ & & W(Z) \\ & & Com. \end{bmatrix}$$

→ *A* is serializable, outcome is the same as a serial schedule *B*

→ Any reordering of *B* results in a serializable schedule

Concurrency Control

– ***Recoverable*** iff any TX is only committed after all TXs from which TX has read uncommitted data have been committed

$$C = \begin{bmatrix} T1 & T2 \\ R(X) & \\ W(X) & \\ & R(X) \\ & W(X) \\ Com. & \\ & Com. \end{bmatrix} \qquad D = \begin{bmatrix} T1 & T2 \\ R(X) & \\ W(X) & \\ & R(X) \\ & W(X) \\ & Com. \\ Abort & \end{bmatrix}$$

→ C is recoverable, $T2$ commits after $T1$

→ D is not recoverable, $T2$ has commited on invalid value X

Concurrency Control

– ***Avoid cascading aborts*** iff no TX reads uncommitted data

$$F = \begin{bmatrix} T1 & T2 \\ & R(X) \\ R(X) & \\ W(X) & \\ & W(X) \\ Abort & \\ & Com. \end{bmatrix} \qquad G = \begin{bmatrix} T1 & T2 \\ R(X) & \\ W(X) & \\ & R(X) \\ & W(X) \\ Abort & \\ & Abort \end{bmatrix}$$

→ *F* avoids cascading aborts

→ *G not:* Abort of *T1* forces *T2* to abort

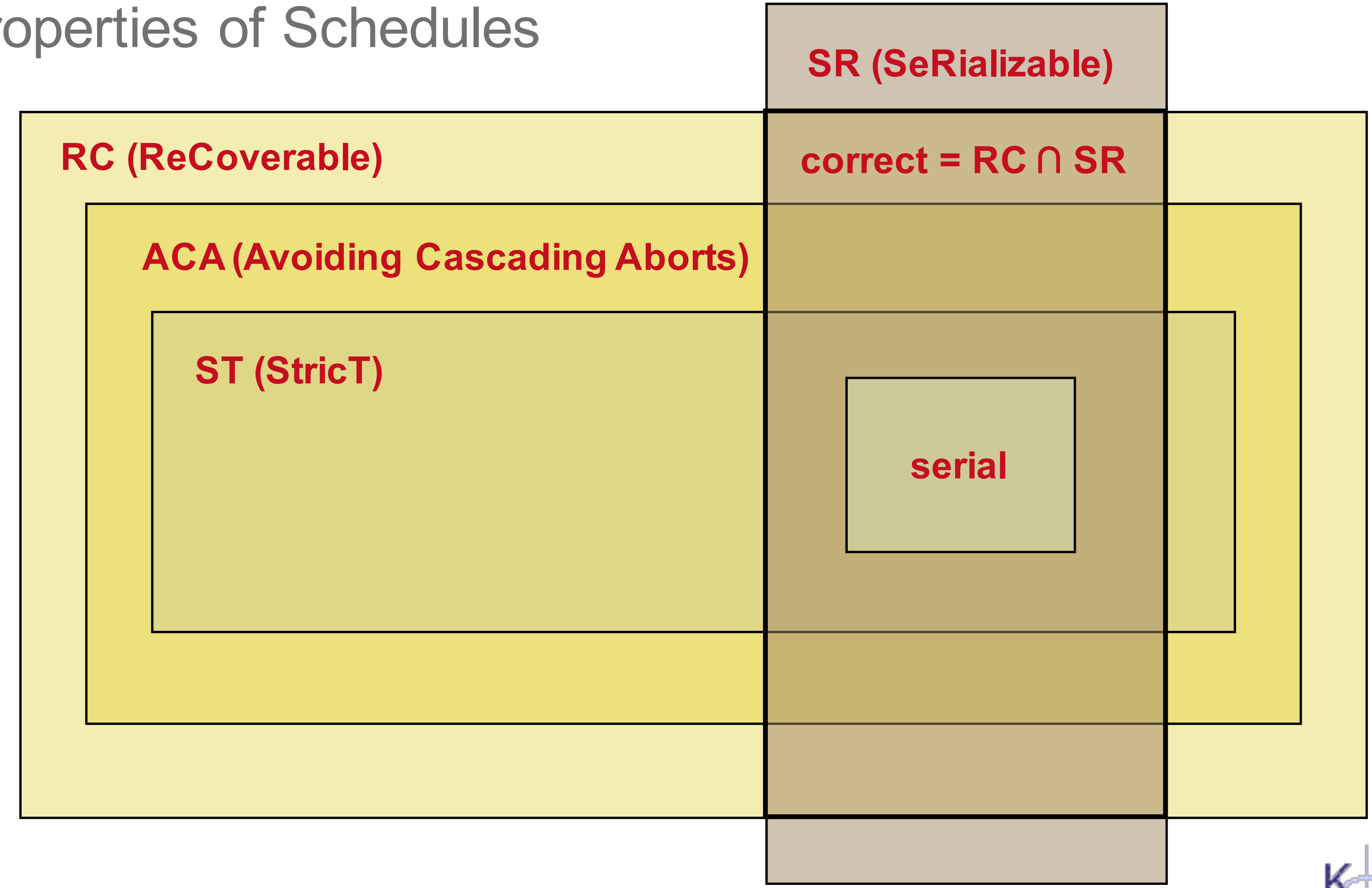
Concurrency Control

Strict iff no TX reads or overwrites uncommitted data

$$H = \begin{bmatrix} T1 & T2 \\ R(X) & \\ & R(Y) \\ W(X) & \\ & W(Y) \\ Com. & \\ & R(X) \\ & W(X) \\ & Com. \end{bmatrix}$$

→ *H* is strict

Properties of Schedules



Properties of Schedules

In practice, at least serializability and recoverability are used to ensure the ACID properties

→ **correct schedules**

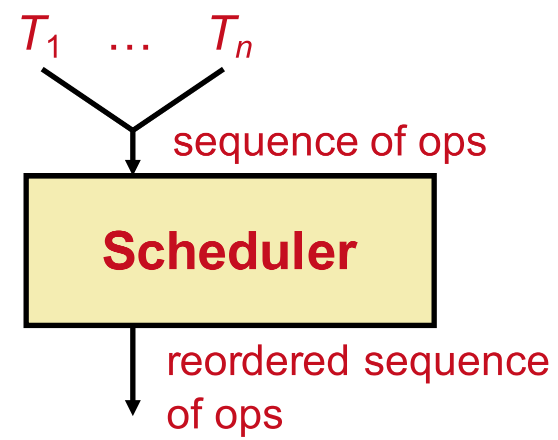
Two main variants of concurrency approaches

- **pessimistic** (locking)
- **optimistic** (non-locking)

Our focus: Locking

Scheduler

- How can correct schedules be enforced automatically?
- A **Scheduler** reorders the operations issued by the TXs
- For each operation there are three possibilities
  - immediate execution,
  - delaying the execution, and
  - rejecting the execution
    → respective TX is aborted
- But how must a scheduler reorder the operations to enforce correct schedules?



Locking

- Similar to critical sections (mutual exclusion)
- However, locks are not granted manually by the programmer but *automatically* by the scheduler

- Decreases concurrency
  (TXs may have to wait until required locks are granted)
- Usually two types of locks are used to increase concurrency
  - **Read lock**: TX can read data item after lock was granted
  - **Write lock**: TX can read and write data item after lock was granted

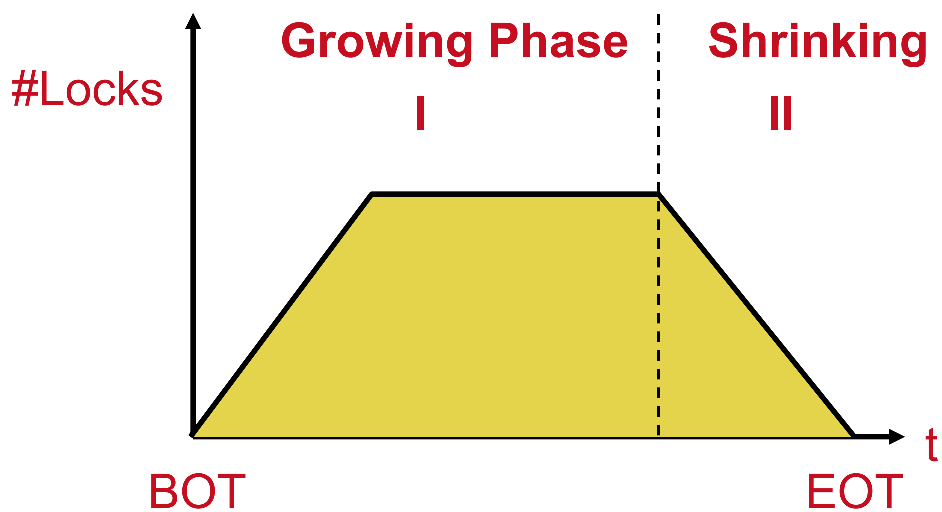
Lock compatibility

	Read	Write
Read	+	–
Write	–	–

- After a read lock was granted, only further read locks can be granted but no write locks
  → shared lock
- After a write lock was granted, no further locks (neither read nor write locks) can be granted
  → exclusive lock
- A read lock can be upgraded to a write lock provided that no further read locks have been granted
- A write lock can be downgraded to a read lock if the TX has not yet written the data item. After a downgrade, further read locks can be granted

2PL (Two Phase Locking)

- After a TX has released a lock, it cannot request a new lock
  → a TX must hold all locks until it needs no further lock
  → reduces concurrency because locks may be held longer
- At the end of the TX, all remaining locks are released
- Ensures serializability, but not recoverability, deadlock freeness, and avoiding cascading aborts

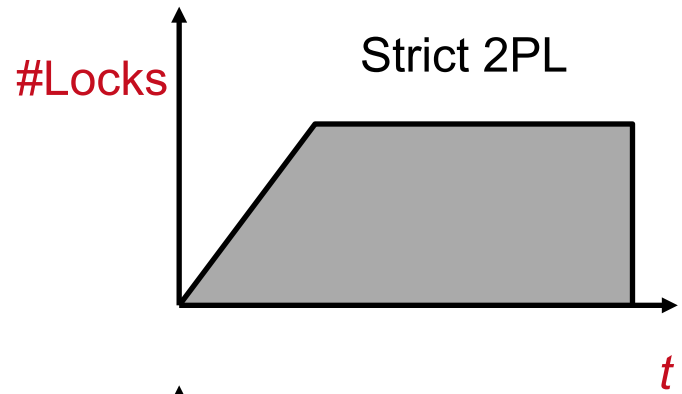


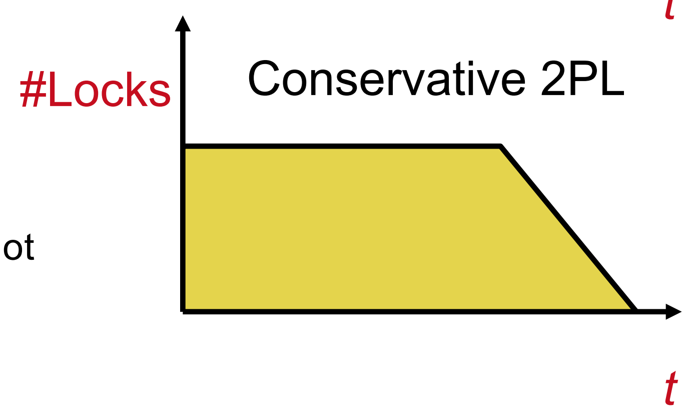
More stringent variants of 2PL

- **Strict 2PL**

  (all write-locks are held until TX end)
  - refers only to the *release* of locks
  - ensures strict schedules (→ RC & ACA) in addition to serializability
  - can still result in deadlock(s) (risk even higher!)
- **Conservative 2PL**

  (all locks are acquired at TX start)
  - refers only to the *acquisition* of locks
  - ensures deadlock freeness in addition to serializability
- **Combination** of strict and conservative 2PL is (often) not used due to degraded concurrency!
- Strict 2PL **most used** in practice!





Granularity of Data Items / Locks

- Granularity of data items and locks
  - depends on the application scenario and
  - varies from individual data items to sets of files
- Determines concurrency and locking overhead

  (→ tradeoff)
  - **Fine-grained locking**
    - high concurrency but also high locking overhead
  - **Coarse-grained locking**
    - low locking overhead but also low concurrency
- **Lock escalation**
  - TXs starts locking items of fine granularity
  - If a TX acquires too many locks, the granularity of locks is increased

DISTRIBUTED TRANSACTIONS

Distributed Transactions

Often local transactions are not sufficient

- E.g., booking a journey requires **atomic** booking of a flight, a hotel, and a rental car at the airline, the hotel, and the car rental service

Distributed transactions allow transactions to span multiple independent participants on different nodes

- Commit and abort of distributed TXs have to be coordinated among the participants

Distributed Transactions

- More complicated than centralized transactions (e.g., in a database) due to the nature of distributed systems
  - Arbitrary communication delays
  - Distinct execution speeds
  - Link failures (→ network partitions)
  - Node failures (→ process crashes)
  - Partial failures instead of total failures
  - …

Distributed 2PL

- Each data item is stored at exactly one node
- Each node has a scheduler managing its local data items
- The schedulers at all nodes, taken together, constitute a **distributed scheduler**
- Granting a lock on data item *x* only depends on the locks currently active on *x* → decision can be taken locally
- The schedulers must agree on the beginning of the shrinking phase, i.e., on the *first* release of a lock
- Commit or abort operation is sent to *all* nodes where the TX has accessed data items → **atomic commit protocol (ACP)** needed!
- If strict 2PL is used, there is not need for the schedulers to agree on the beginning of the shrinking phase; they simply release all locks of a TX when they receive the commit or the abort command

Two-Phase Commit (2PC)

- Prevalent commit protocol for distributed transactions
- Achieves only **atomicity**
  (other ACID properties neglected!)
- Participants go through **two phases** which are needed to allow **unilateral aborts** of participants
  - **Prepare (to commit)** (aka. Voting Phase)
    - Each participant votes to commit or to abort the TX
    - Once a participant has voted to commit, it can no longer abort the TX unilaterally
  - **Commit** (aka. Completion Phase)
    - Participants actually commit, after **consensus** has been reached that **all** participants are prepared. Otherwise all participants abort

Two-Phase Commit (contd.)

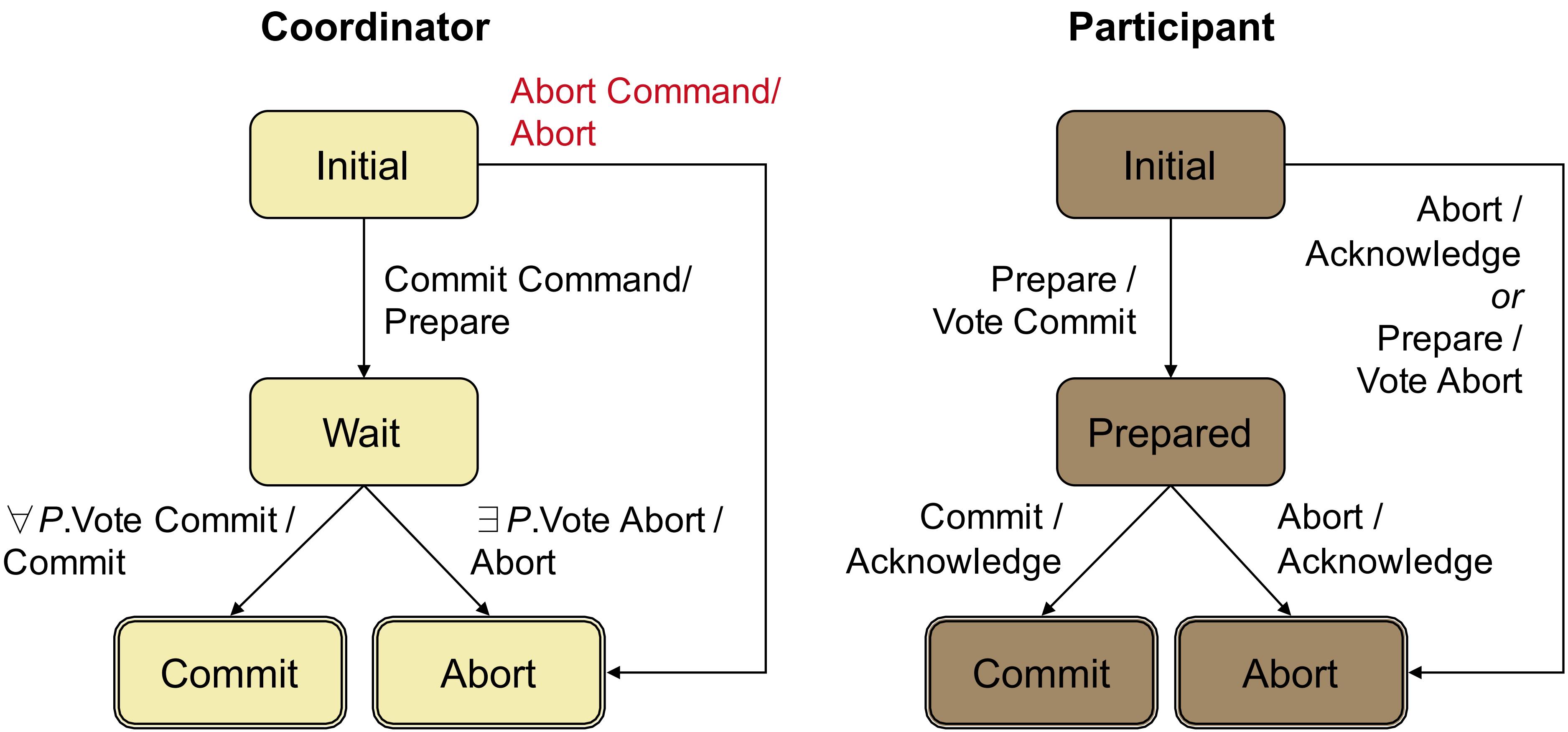
- Any *participant* can initiate to commit or to abort the transaction (i.e., issue **commit** or **abort command**)
  - In client/server systems, usually the client initiates commit
  - In messaging system, any participant can initiate commit

- A **Coordinator** is used to achieve consensus
  - All participants must have registered at the coordinator
  - Coordinator requests all participants to vote
  - Decides to commit if *all* participants have voted to commit
  - Decides to abort if *any* participant has voted to abort

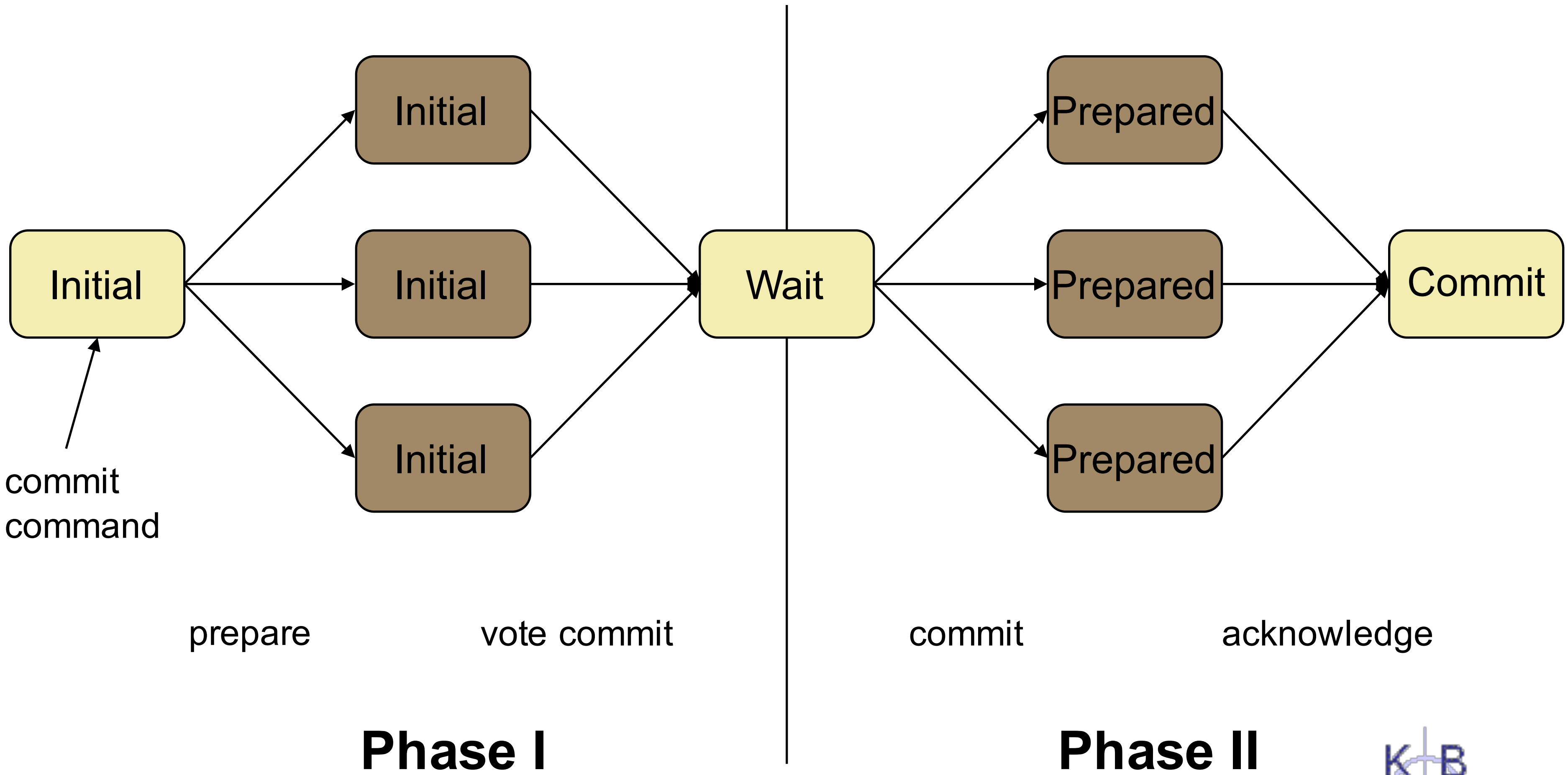
- Also cooperative decentralized implementations possible

2PC State Transitions

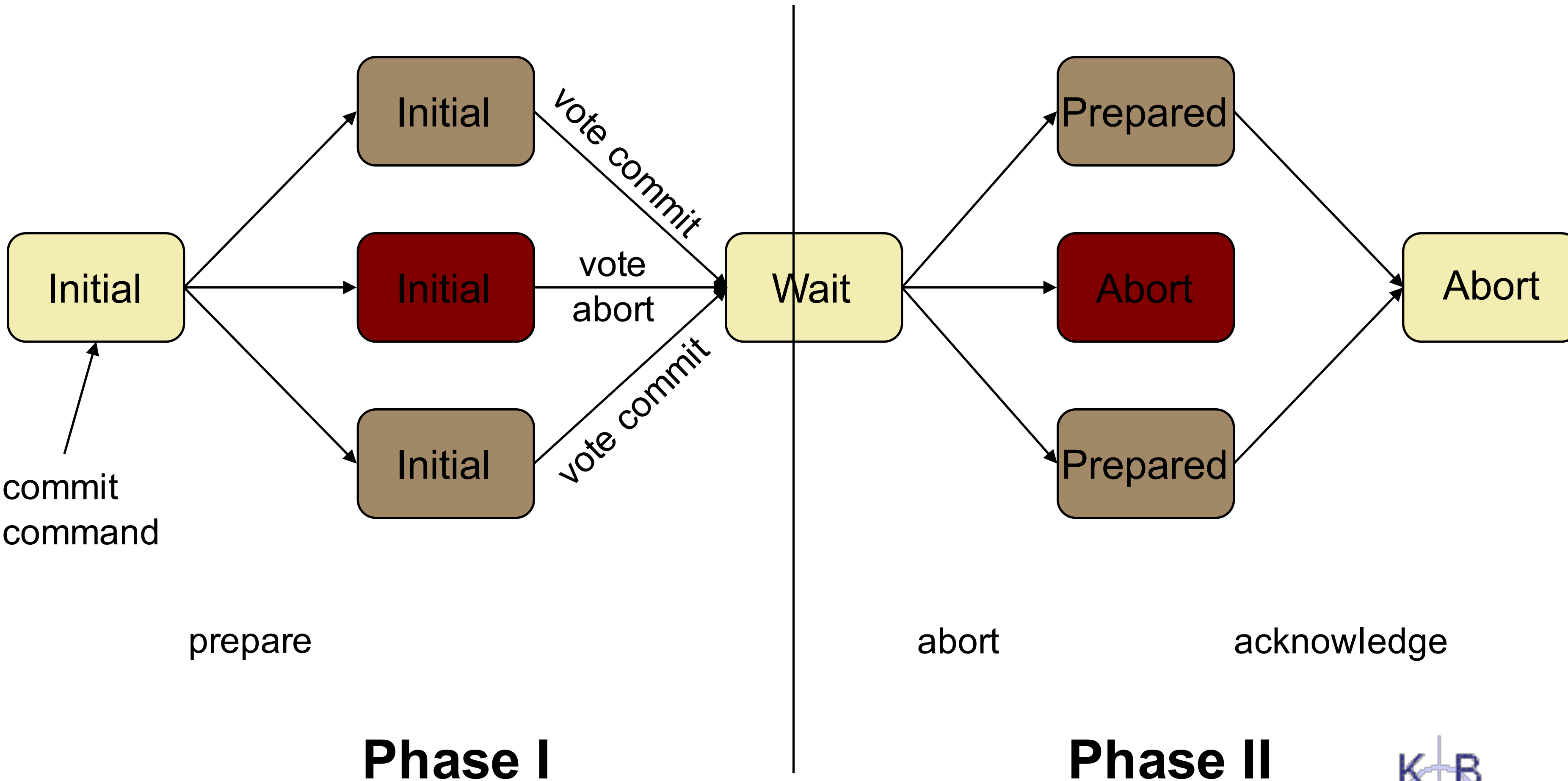
Message received/Message send in turn



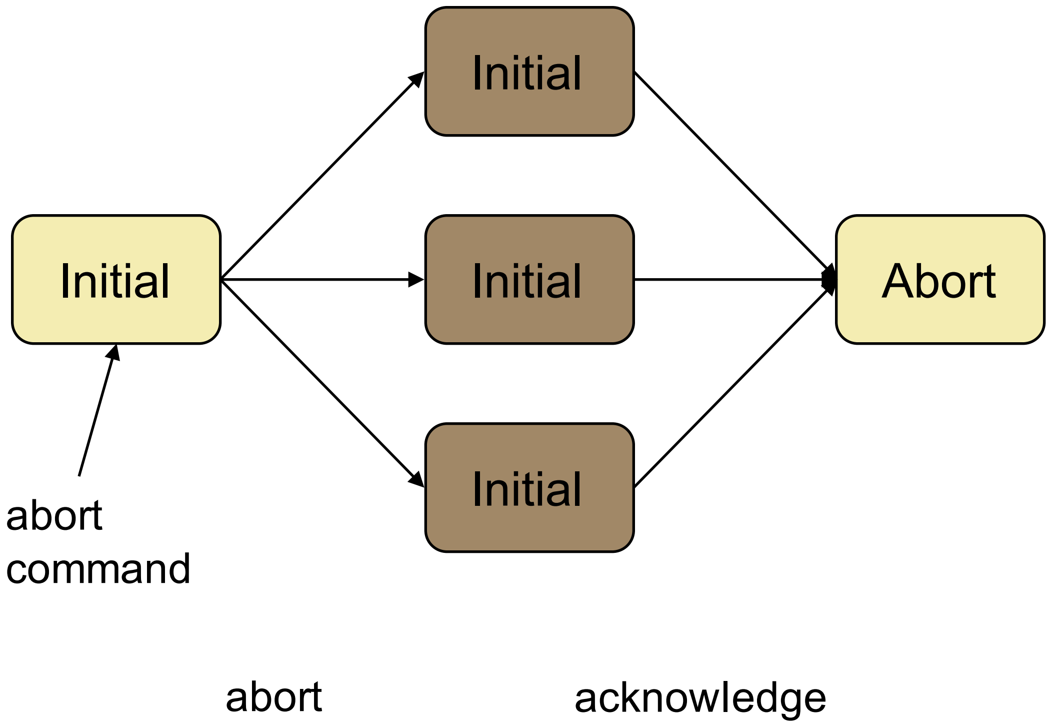
Centralized 2PC (successful completion)



Centralized 2PC (unsuccessful completion)

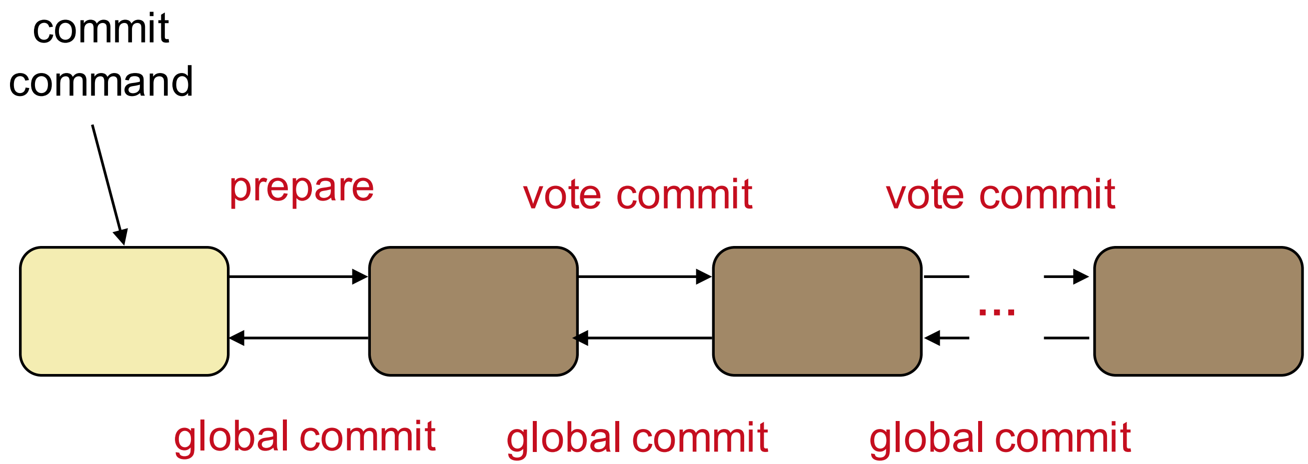


Centralized 2PC (unsuccessful completion)



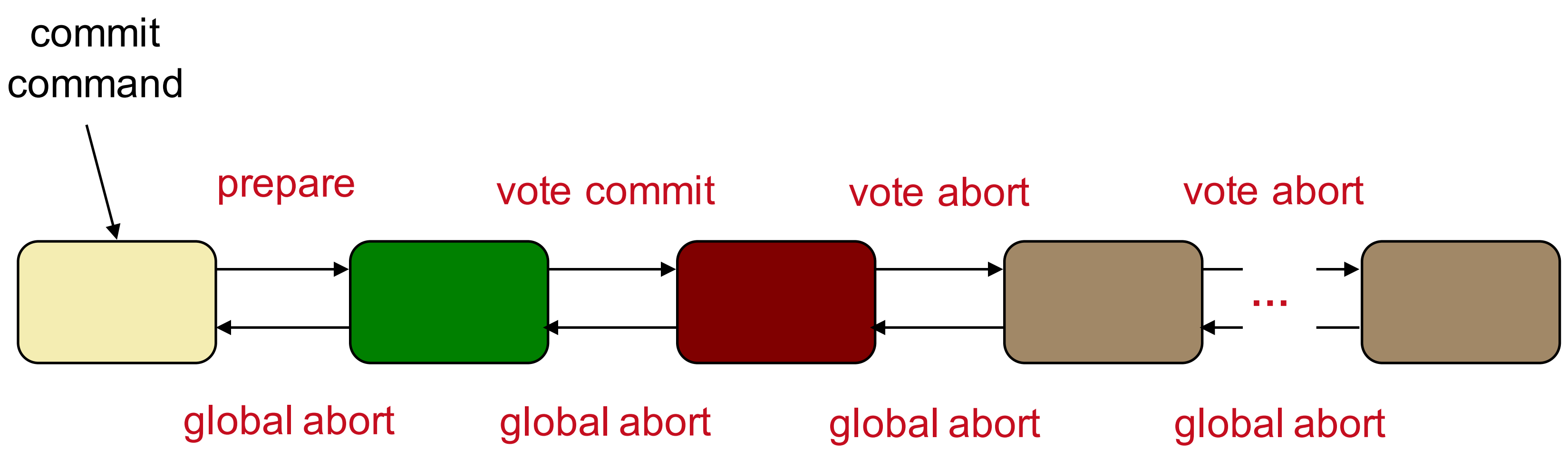
Phase I

Linear 2PC (successful completion)



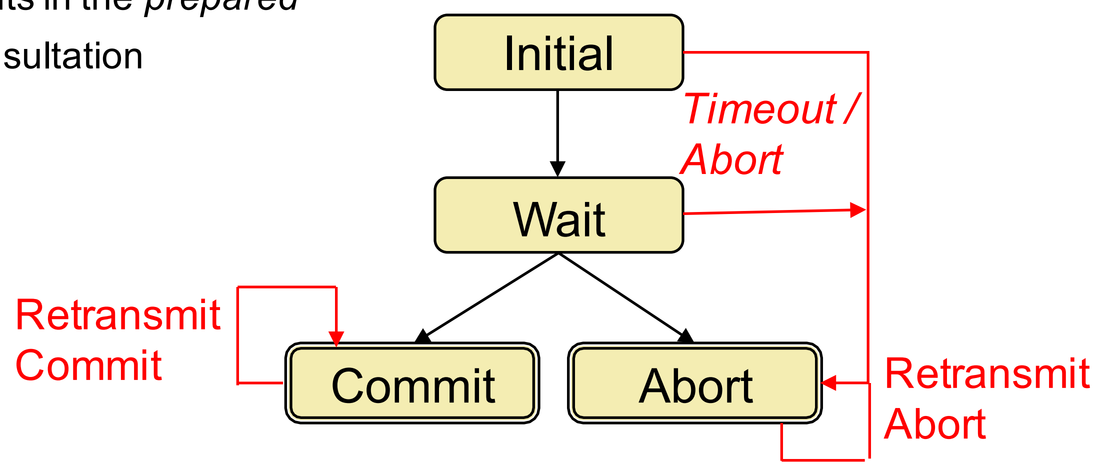
- Requires knowledge of next node
  - can be transmitted along with messages
- Fewer messages but no parallelism

Linear 2PC (unsuccessful completion)



Coping with Failures

- Timeouts are used to cope with failures such as lost messages; however, timeout values are difficult to choose
- Timeout actions not requiring consultation of other parties
  - Coordinator aborts TX when it timeouts in *initial* or *wait* state
  - Coordinator retransmits commit/abort message to participants when it timeouts in *commit*/*abort* state
  - A participant aborts TX when it timeouts in *initial* state
- Participant timeouts in the *prepared* state requires consultation



Coping with Failures

- The period from the moment a participant has voted to *commit* to the moment it knows the global decision is called **uncertainty period**

- An uncertain participant is blocked until it becomes certain
  - It cannot unilaterally abort because it cannot revoke its vote
  - It can also not unilaterally commit because the global decision may be to abort
  - It can try to contact other participants to find one which is certain (that either voted abort or that already knows the global decision)
  - If it can only contact uncertain participants, it is blocked
    (reason may be communication failure or failure of all other participants)

The Downsides of Distributed Transactions

- Not all resources may support distributed transactions
- Long-running transactions may block resources due to locking resulting in degraded throughput
- Distributed transactions introduce a large overhead due to necessary coordination

Java Transaction API (JTA)

- Specification developed by Sun
- Enables distributed transactions across multiple XA resources
- XA (ext. Architecture) is a standard defined by the Open Group
  - Global transaction manager (TP monitor)
    - Coordinates distributed transactions
    - Uses 2PC protocol
  - Local resource manager at each XA resource
    - Interacts directly with the resource (e.g., database)
    - Uses e.g., JMS, JDBC (offers local transactions)
- JTA Implementations
  - JBossTS
  - Atomikos TransactionsEssentials
  - Bitronix JTA

Bibliography

1. P. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, Reading, MA, USA, 1987. http://research.microsoft.com/pubs/ccontrol/
2. M. T. Özsu, *Notes On Database System Reliability*, http://www.cs.mcgill.ca/~cs577/lectures/Reliability.pdf
3. P. Bernstein and E. Newcomer. *Principles of Transaction Processing*. Morgan Kaufmann Publishers, 1997.
4. G. Coulouris, J. Dollimore, and T. Kindberg. *Distributed Systems: Concepts and Design*. Addison-Wesley, 2001. pp. 519--523