

# Chapter 6

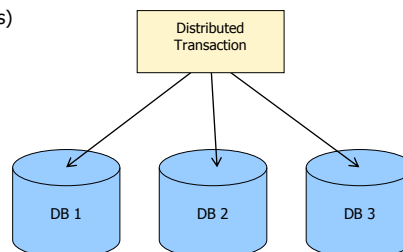
## Distributed Transactions

Commit Protocols  
X/OPEN-DTP  
Global Serializability



## TA-Mgmt in Distributed DBMS

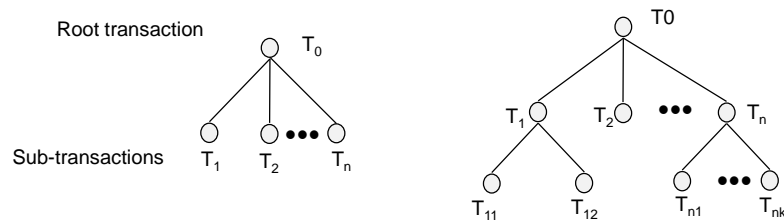
- **ACID properties must be ensured also in the distributed case**
- **Logging and Recovery**
  - global Commit Protocol
  - Robustness w.r.t. to partial failures, esp. Communication failures (network partitions)
- **Synchronization**
  - Global serializability
  - Global dependencies (e.g., global deadlocks)



## Transaction Structure

### Control Structure: Transaction Tree

- Represents invocation relations
- Single-level or multi-level
- No isolated rollback of sub-transaction: abort of sub-transactions leads to abort of overall transaction



## Commit Protocols

### Ensuring atomicity of distributed transaction by comprehensive Multi-Phase-Commit-Protocol

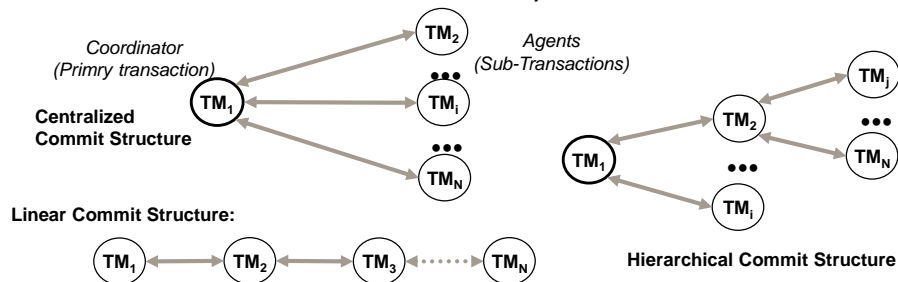
#### Requirements

- Correctness
- Low Overhead (#Messages, #Log-Writes)
- Low extension of response time
- Robustness against crashes and communication failures
- Node autonomy: each node has the possibility of an *unilateral abort* as long as possible



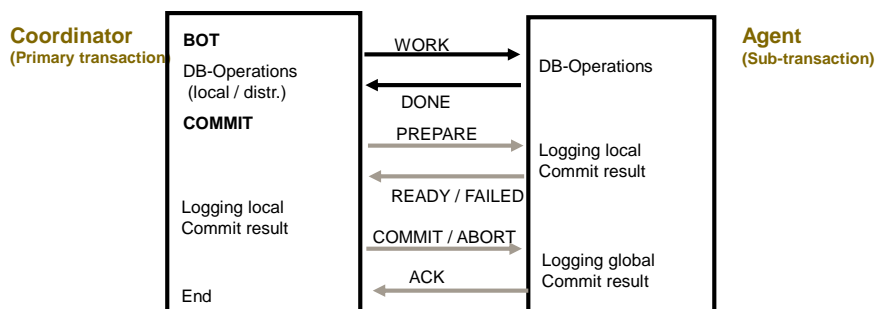
## Commit Protocols (2)

- **Transaction Manager (TM)** at each node  
(1 Coordinator + N-1 Agents)
- **Standard: 2-Phase-Commit**
- **Alternatives: 1-Phase-Commit, 3-Phase-Commit**
- **Communication structures: centralized, linear or hierarchical**



## Centralized 2-Phase-Commit (N=2)

- **Overhead**
  - Successful processing: 4 messages, 4 Log-Writes
  - ABORT messages only for sub-transactions which not voted with FAILED
- **Problem Coordinator drop out => Blocking**



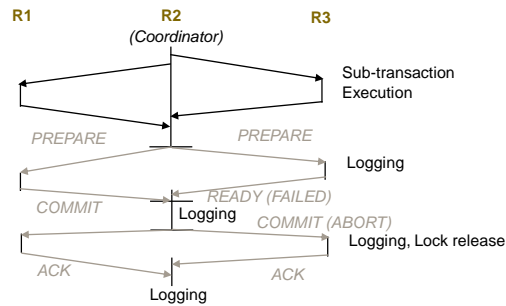
## Centralized 2-Phase-Commit (N=3)

- **Basic mechanism:**

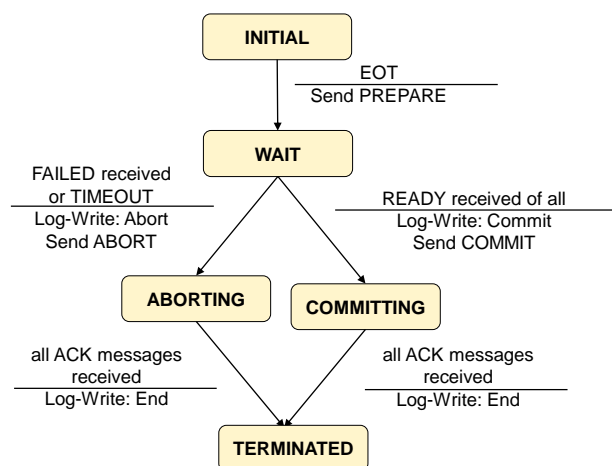
- 4 (N-1) Messages (N = no of nodes)
- 2 N Log-Writes

- **Optimization for read-only sub-transactions (M)**

- 4 (N-1) - 2M Messages for  $M < N$ , 2 (N-1) for  $M=N$
- 2 N - M Log-Writes



## 2PC State Transitions: Coordinator



## 2PC: Failure Management

### ■ Timeout Conditions Coordinator:

- WAIT => Abort Transaction; send ABORT Message
- ABORTING, COMMITTING => record Agents, which did not send ACK so far

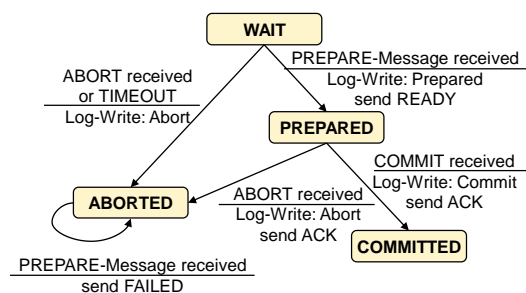
### ■ Coordinator Drop Out

- Log-State TERMINATED:
  - UNDO/REDO-Recovery depending on transaction end
  - No "open" sub-transactions
- Log-State ABORTING:
  - UNDO-Recovery
  - ABORT-Message to each node, which did not send ACK so far
- Log-State COMMITTING:
  - REDO-Recovery
  - COMMIT-Message to each node, which did not send ACK so far
- Otherwise: UNDO-Recovery



## 2PC: Failure Management (2)

### State Transitions Agent



### ■ Timeout-Conditions for Agents:

- WAIT => unilateral ABORT
- PREPARED => ask coordinator (or other node) about transaction state/end

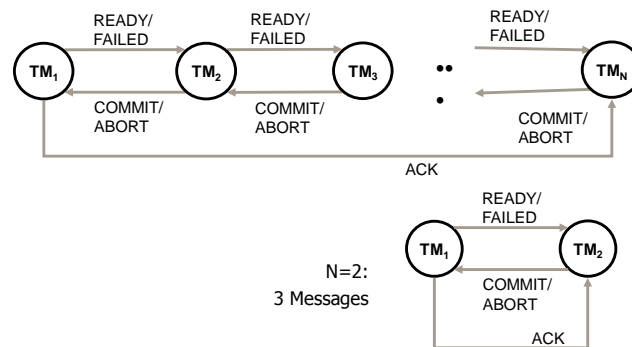
### ■ Agent Drop Out:

- Log-State COMMITTED: REDO-Recovery
- Log-State ABORTED or no 2PC-Log-Record: UNDO-Recovery
- Log-State PREPARED: ask coordinator about transaction state/end (coordinator keeps information, since no ACK so far)



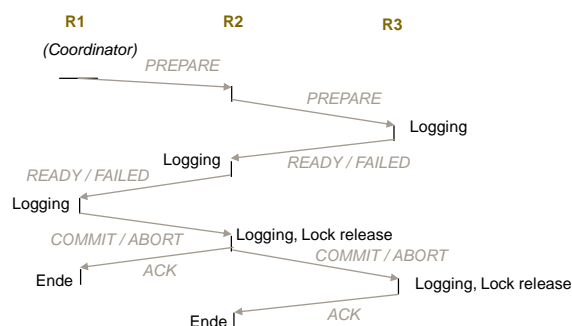
## Linear 2PC

- Sequential Commit Processing, #Messages:  $(N-1) + N = 2N-1$
- Transfer of Coordination Task to last Agent (*"Last Agent"-Optimization*)



## Hierarchical 2PC

- General model with arbitrary nesting**
  - Answering time increases with nesting depth (lower parallelization)



## 2PC Optimizations (1)

### ▪ ***Read-Only Sub-Transactions***

- Read-Only-Agent answers in phase 1 with READ-ONLY
- No logging necessary, lock release possible after phase 1
  - However: all other agents need to have their work finished, since locks are released
- Thus, phase-2-communication not needed for read-only participants



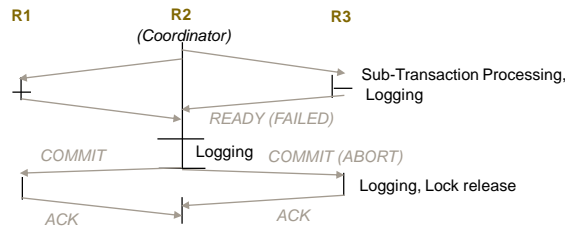
## 2PC Optimizations (2)

### ▪ ***Presumed Abort-Protocol***

- As soon as coordinator decides abort and has send the corresponding messages to agents, he forgets all transaction data
- Agents do not reply to abort message (no ack)
- If coordinator abort message does not reach an agent, this agent asks coordinator about final decision: if then coordinator does not find any information in its log, abort is assumed
- Advantages
  - Coordinator does not need to write abort record
  - Aborted transactions do not cause ack messages



# 1-Phase-Commit

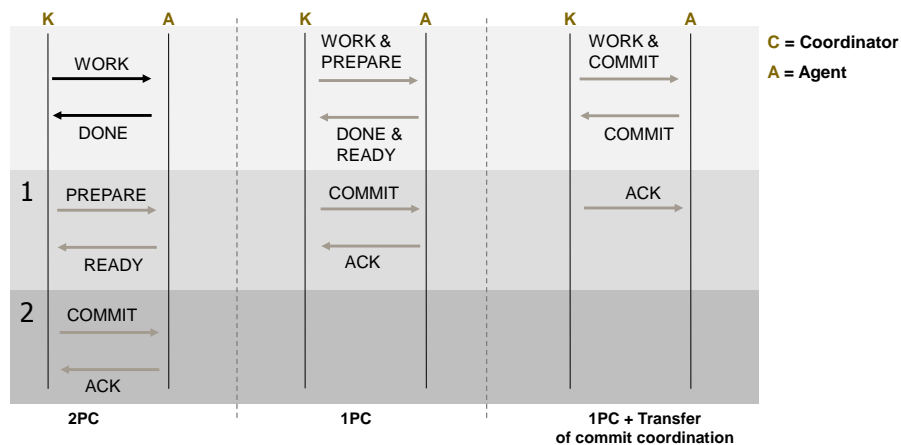


- **Sub-transactions already save their modifications before they pass back results to primary transaction**
  - After local Commit at coordinator node transaction success is given
- **2 (N-1) Messages**
  - Esp. advantageous for short (distributed) transactions
- **Disadvantages**
  - High dependency from coordinator, early relinquishment of unilateral abort
  - Higher probability of blocking through early prepared



## 1-Phase-Commit (2)

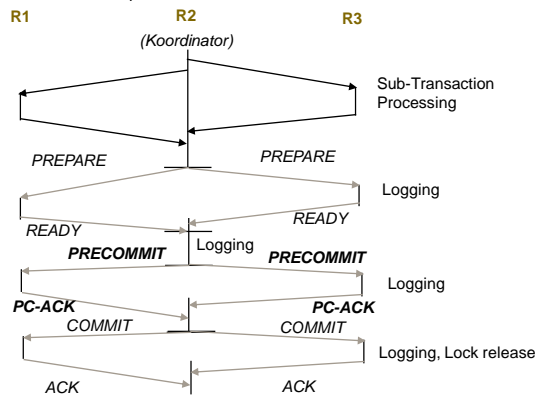
- For N=2 one more message can be saved by transfer of coordination





## 3-Phase-Commit

- **Non-blocking Technique**
- **Preconditions:**
  - No network partitions
  - At most  $K < N$  nodes fail simultaneously



## 3-Phase-Commit (2)

- **ABORT Processing like 2PC**
- **New intermediate phase, if sub-transactions finish phase 1 with READY**
  - Coordinator gets to state PRECOMMIT and tells all the sub-transactions about this decision
  - After k receipts (PC-ACK) COMMIT decision is taken
  - Now it is clear that transaction 'will survive', not earlier
- **Coordinator drop out: selection of new coordinator**
  - Requesting transaction state of not finally processed transactions at 'surviving' nodes
    - Commit / Abort (or no Information, resp.): notification
    - Precommit at least at one surviving node:  
Commit protocol is continued by new coordinator by sending precommit messages
- **Resolves blocking problem in state Prepared**
  - Negative coordinator decision: no sub-transaction in Precommit
  - Positive coordinator decision: at least 1 node must be in Precommit
  - Even if coordinator in Precommit, abort still possible!



# Messages Overhead

N: #Nodes

M: #Read-Only Sub-Transactions

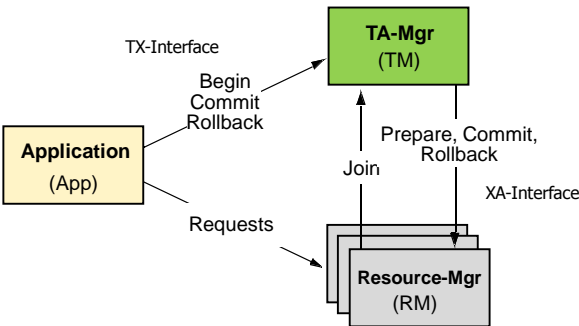
	General	Example 1 (N=2, M=0)	Example 2 (N=10, M=5)
1-Phase-Commit	$2*(N-1)$	2	18
Linear 2PC	$2*N-1$	3	19
centralized/hierarchical 2PC	$4*(N-1)-2M$	4	26
3-Phase-Commit	$6*(N-1)-4M$	6	34



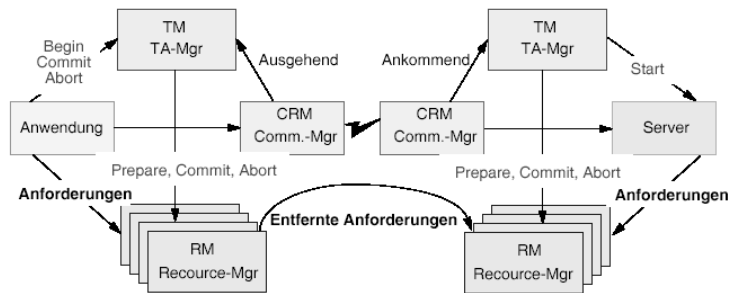
# TA-Mgmt in Open Systems

## X/OPEN DTP

- Independent TA-Mgr
- Resource Manager
  - recoverable
  - XA-compliant



## TA-Mgmt in Open Systems (2)



### TA-Ablauf

- AW startet TA, die vom lokalen TA-Mgr verwaltet wird
- Wenn die AW oder der RM, der für die AW eine Anforderung bearbeitet, eine entfernte Anforderung durchführen, informieren die CRMs an jedem Knoten ihre lokalen TA-Mgr über die ankommende oder ausgehende TA
- TA-Mgr verwalten an jedem Knoten jeweils die TA-Arbeit am betreffenden Knoten
- Wenn die AW COMMIT oder ROLLBACK durchführt oder scheitert, kooperieren alle beteiligten TA-Mgr, um ein atomares und dauerhaftes Commit zu erzielen.



## Concurrency Control in Distributed DBS

### Centralized locking techniques unacceptable

- Decrease of node autonomy
- High communication overhead

### Distributed locking techniques

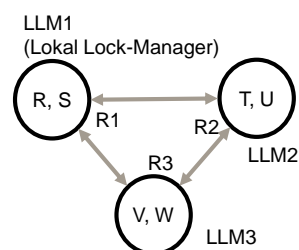
- Each node manages locks for local data
- Lock requests do not require messages
- Locks release within commit protocol
- Most relevant approach

### Timestamp ordering

- Transactions get unique timestamp at BOT
- Data accesses in timestamp order
- No deadlocks, however many aborts

### Optimistic concurrency control

- Validation at transaction end
- Many aborts and starvation of transactions



## Homogeneous Federations (1)

### ▪ Homogeneous Federation

- Data distributed on  $n$  *Sites*
- $D = \bigcup_{i=1}^n D_i, 1 \leq i \leq n$
- No replication
- global TA only

### ▪ Definition *Global History*

- Given: federation with  $n$  *Sites*.  
 $T = \{t_1, \dots, t_m\}$  set of (global) TA.  
 $s_1, \dots, s_n$  local histories
- A Global History for  $T$  and  $s_1, \dots, s_n$  is a History  $s$  for  $T$ , so that:  
 $\Pi_i(s) = s_i$  for all  $i, 1 \leq i \leq n$ .



## Homogeneous Federations (2)

### ▪ Sub-Transaction

- Projection of a (global) Transaction on *Site*  $i$

### ▪ Example

- Given: Federation with 2 *Sites*,  $D1 = \{x\}$  und  $D2 = \{y\}$
- $s_1 = r_1(x) w_2(x)$  and  $s_2 = w_1(y) r_2(y)$  local schedules
- $s = r_1(x) w_1(y) w_2(x) c_1 r_2(y) c_2$  global history
- $\Pi_1(s) = s_1$  and  $\Pi_2(s) = s_2$  (without considering commit operations)
- Alternative notation for global Histories:  
Server 1:  $r_1(x) \quad w_2(x)$   
Server 2:  $w_1(y) \quad r_2(y)$



## Homogeneous Federations (3)

### ▪ Definition *Conflict Serializability*

- A global (local) History  $s$  is *globally (locally) conflict serializable*, if there is a conflict-equivalent serial history over the global (local) (sub-) transactions

### ▪ Example

- History  $s$ 

Server 1:	$r_1(x)$	$w_2(x)$
Server 2:	$r_2(y)$	$w_1(y)$
- Scheduler at *Site 1*:  $t_1 < t_2$
- Scheduler at *Site 2*:  $t_2 < t_1$
- Conflict graph would be cyclic,  $s$  not conflict serializable



## Homogeneous Federations (4)

### ▪ Theorem

- Let  $s$  be global History with local Histories  $s_1, \dots, s_n$  over a set  $T$  of Transactions, so that each  $s_i$ ,  $1 \leq i \leq n$ , conflict serializable. Then it is true:
  - $s$  globally conflict serializable, if and only if there is a total order „<“ on  $T$ , which is consistent with the local serialization orders of the Transactions, i.e.,
  - $(\forall t, t' \in T, t \neq t') t < t' \Rightarrow (\forall s_i, 1 \leq i \leq n, t, t' \in \text{trans}(s_i)) (s_i \text{ serial, } s_i \approx_c s'_i) t <_{s'_i} t'$



## Homogeneous Federations (5)

### ▪ Exploitation of 2PL

- Local exploitation of 2PL: Problem
  - global decision, when locks can be released
- Solution 1: *Primary Site 2PL*
  - Locks are managed only at *Primary Site*
    - Drawback: *Primary Site* as bottleneck
- Solution 2: *Distributed 2PL (D2PL)*
  - Managing states of all local schedules at all sites
    - Drawback: high communication overhead
  - Before entering unlock-phase local server asks all others
    - Drawback: Overhead still (too) high



## Homogeneous Federations (6)

### ▪ Exploitation of 2PL (contd.)

- Remark
  - If all local servers apply SS2PL, then the resulting global history is not only conflict serializable but also strict

### ▪ Also available

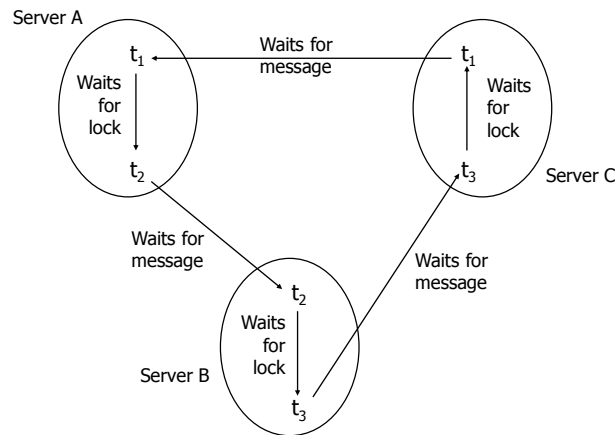
(Weikum, Vossen: Transactional Information Systems, pp. 680)

- Distributed TO
- Distributed SGT
- Distributed optimistic protocols



## Distributed Deadlock Detection (1)

- **Example of a distributed deadlocks**



## Distributed Deadlock Detection (2)

- **Solution 1: *Centralized Detection***

- Exploiting (kind of) *centralized Monitor*
  - Collecting and analyzing 'Wait-for-Information' of local servers
  - After deadlock detection selection of 'victim' by appropriate communication with local servers (taking rollback costs into account)
- Drawback
  - Bottleneck
  - (Communication-) Overhead
- Adequacy
  - Only in the case of fast and highly available communication connections
  - Not for communication over the Internet



## Distributed Deadlock Detection (3)

### ▪ **Solution 1: *Centralized Detection (contd.)***

- Further Problem: *False Deadlocks*
  - Immediately after cycle is closed by last edge, one of the corresponding TAs is locally aborted
  - Monitor does not notice and aborts second TA
  - Does not happen, if all local schedulers use 2PL and there are no spontaneous aborts („TA-suicide“)
- Timeouts can be used



## Distributed Deadlock Detection (4)

### ▪ **Solution 2: *Decentralized Approaches***

- *Edge chasing*
- *Path pushing*
- ***Edge chasing***
  - Blocked TA sends *Probe*-Message with own TA-ID to blocking TA
  - Each TA, which gets *Probe*-Message, sends it further to blocking TA
  - If a TA gets a message containing own TA-ID, deadlock is detected
  - Solution can be own abort





## Distributed Deadlock Detection (5)

### ▪ **Path pushing**

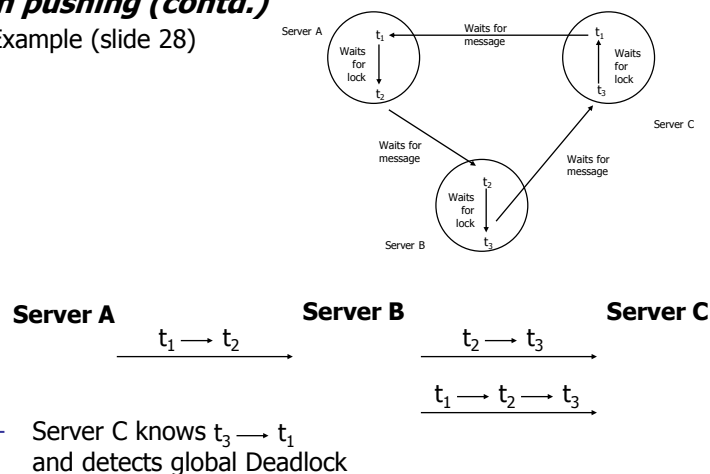
- Idea: circulating paths instead of single TA-IDs
- Algorithm
  1. Each server, which has a *waits-for path* from  $t_i$  to  $t_j$ , with  $t_i$  has an incoming and  $t_j$  an outgoing *waits-for message*, sends this path along the outgoing edge, providing that identifier of  $t_i$  is smaller than the one of  $t_j$ .
  2. After receipt of a path the server concatenates this path with its local paths and passes result further again. If there is a cycle among  $n$  servers, at least one of them detects the cycle in at most  $n$  steps.



## Distributed Deadlock Detection (6)

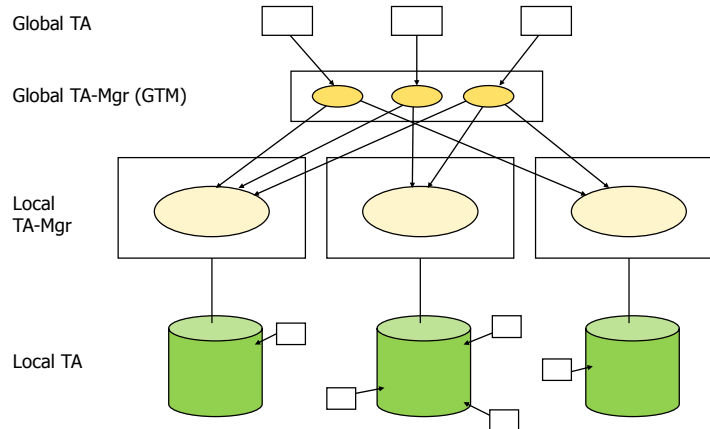
### ▪ **Path pushing (contd.)**

- Example (slide 28)



## Heterogeneous Federations (1)

### ■ Illustration



## Heterogeneous Federations (2)

### ■ Histories in heterogeneous Federations

- Example
  - $D_1 = \{a, b\}$ ,  $D_2 = \{c, d, e\}$
  - $D = \{a, b, c, d, e\}$
  - local TAs:  $t_1 = r(a) w(b)$ ,  $t_2 = w(d) r(e)$
  - global TAs:  $t_3 = w(a) r(d)$ ,  $t_4 = w(b) r(c) w(e)$
  - local Histories:
    - $s_1 = r_1(a) w_3(a) c_3 w_1(b) c_1 w_4(b) c_4$
    - $s_2 = r_4(c) w_2(d) r_3(d) c_3 r_2(e) c_2 w_4(e) c_4$



## Heterogeneous Federations (3)

### ▪ Histories in heterogeneous Federations (contd.)

- Definition **Global History** (revisited)
  - Considered heterogeneous Federation is supposed to have  $n$  Sites
  - $T_1, \dots, T_n$  local TA at Sites 1, ...,  $n$
  - $T$  set of global TA
  - $s_1, \dots, s_n$  local Histories with  
 $T_i \subseteq \text{trans}(s_i)$  and  $T \cap \text{trans}(s_i) \neq \emptyset$  for  $1 \leq i \leq n$ .
  - A (heterogeneous) global History (for  $s_1, \dots, s_n$ ) is a History  $s$  for  
 $\bigcup_{i=1}^n T_i \cup T$ , so that local projection equals local history, i.e.  
 $\Pi_i(s) = s_i$  for all  $i$ ,  $1 \leq i \leq n$ .



## Heterogeneous Federations (4)

### ▪ Histories in heterogeneous Federations (contd.)

- Example
  - $D_1 = \{a\}$ ,  $D_2 = \{b, c\}$
  - global TAs:  $t_1 = r(a) w(b)$ ,  $t_2 = w(a) r(c)$
  - local TA:  $t_3 = r(b) w(c)$
  - Assumption: GTM decides to process  $t_1$  first
  - Resulting local Histories:
    - Server 1:  $s_1 = r_1(a) \quad w_2(a)$
    - Server 2:  $s_2 = \quad r_3(b) w_1(b) \quad r_2(c) \quad w_3(c)$
  - note: both global TAs are processed serially at both Sites
  - Global History:  $s = r_1(a) r_3(b) w_1(b) c_1 w_2(a) r_2(c) c_2 w_3(c) c_3$
  - Obviously  $s_1, s_2 \in \text{CSR}$ , but  $s_1 \approx_c t_1 t_2$  and  $s_2 \approx_c t_2 t_3 t_1$
  - Thus, conflict graph of  $s$  cyclic processing order as chosen by GTM not acceptable



## Heterogeneous Federations (5)

### ■ Histories in heterogeneous Federations (contd.)

- Example (contd.)
  - Reasons
    - Direct Conflict between global Transactions in  $s_1$
    - *indirect Conflict* in  $s_2$ , since
      - global TA  $t_2$  in direct Conflict with local TA  $t_3$  and
      - local TA  $t_3$  in direct Conflict with global TA  $t_1$
- Indirect Conflicts can also occur, if there are no direct conflicts between global TA

(Weikum, Vossen: Transactional Information Systems, p. 693)



## Heterogeneous Federations (6)

### ■ Global Serializability

- Definition ***Direct and Indirect Conflicts***
  - $s_i$  local History and  $t$  and  $t'$  Transactions of  $\text{trans}(s_i)$ ,  $t \neq t'$ 
    1.  $t$  and  $t'$  are in Direct Conflict in  $s_i$  if
$$(\exists p \in t) (\exists q \in t') (p, q) \in \text{conf}(s_i) \quad (\text{cf. Chapter 3})$$
    2.  $t$  and  $t'$  in Indirect Conflict in  $s_i$ , if there is a sequence  $t_1, \dots, t_r$  of Transactions in  $\text{trans}(s_i)$ , so that  $t$  in  $s_i$  in direct Conflict with  $t_1$ ,  $t_j$  in  $s_i$  in direct Conflict with  $t_{j+1}$ ,  $1 \leq j \leq r-1$  and  $t_r$  in  $s_i$  in direct Conflict with  $t'$
    3.  $t$  and  $t'$  are in  $s_i$  in Conflict, if they are in  $s_i$  in direct or indirect Conflict



## Heterogeneous Federations (7)

### ▪ Global Serializability

- Definition **Global Conflict Graph**

- Let  $s$  be global History for local Histories  $s_1, \dots, s_n$
- Let  $G(s_i)$  be Conflict Graph of  $s_i$ ,  $1 \leq i \leq n$ , which considers direct as well as indirect Conflicts
- The **Global Conflict Graph** of  $s$  is defined as the Union of all  $G(s_i)$ ,  $1 \leq i \leq n$ , i.e.  
$$G(s) := \bigcup_{i=1}^n G(s_i)$$

- **(Multidatabase Serializability) Theorem**

- Given local Histories  $s_1, \dots, s_n$ ,  
with each  $G(s_i)$ ,  $1 \leq i \leq n$ , acyclic (i.e.,  $s_i \in \text{CSR}$ )
- $s$  global History for  $s_i$ ,  $1 \leq i \leq n$
- then:  $s$  is globally conflict serializable if and only if  $G(s)$  acyclic



## Heterogeneous Federations (8)

### ▪ Global Serializability through local Guaranties

- Exploitation of Commitment Ordering (cf. Chapter 3)

- One of several possibilities  
(see Weikum, Vossen: Transactional Information Systems, pp. 698, for more)
- Theorem
  - $s$  global History for  $s_1, \dots, s_n$
  - If  $s_i \in \text{COCSR}$ ,  $1 \leq i \leq n$ , and all global TA process their Commits strictly sequentially, then  $s$  globally serializable



## Heterogeneous Federations (9)

### ▪ Global Serializability through local Guaranties (contd.)

- Exploitation of Commitment Ordering (contd.)
  - Example
    - $s_1 = r_1(a) \ c_1 \ w_3(a) \ w_3(b) \ c_3 \ r_2(b) \ c_2$
    - $s_2 = w_4(c) \ r_1(c) \ r_2(d) \ r_4(e) \ c_1 \ c_2 \ [w_4(d) \ c_4]$
    - $t_1, t_2$  global;  $t_3, t_4$  local
    - note: Commit-Operations ordered equally in both Histories
    - Assume  $s_2$  processed until squared bracket
    - note, global TAs concurrently at *Site 2*



## Heterogeneous Federations (10)

### ▪ Global Serializability through local Guaranties (contd.)

- Exploitation of Commitment Ordering (contd.)
  - Example (contd.)
    - If Server 2 realizes COCSR, then  $s_2$  cannot be continued, because the indirect conflict between  $t_2$  and  $t_1$ , which would be caused by  $t_4$ , would require that TA process their Commit operations in the order given by that conflict; but this is not possible any more
    - A COCSR-Scheduler would abort  $t_4$ !

### ▪ Global Serializability without local Guaranties, e.g. by: *Ticket-Based CC*

(see Weikum, Vossen: Transactional Information Systems, pp. 698)



# Conclusion

- **ACID Properties for Distributed Transactions**
- **Synchronization**
  - Ensuring global Serializability
  - Distributed locking techniques preferred (less communication overhead, fewer rollbacks than timestamp and optimistic techniques)
- **Global Deadlock Management**
  - Simplest Solution: Timeout
  - Deadlock Detection (e.g. Wound/Wait) avoids Communication, but results in unnecessary rollbacks
  - Distributed Deadlock Detection: high Overhead, but fewer rollbacks
- **Distributed Commit Protocols**
  - Atomicity and Durability of distributed modifications
  - Standard: hierarchical 2PC
  - Variants with better performance/availability (1PC, 3PC ...)
  - Comparably high overhead
- **XOPEN/DTP (2PC) and local ACID generally do not ensure global Serializability!**

