DBS

Universität
Karlsruhe (TH)

Chapter 10

Distributed Transactions: Synchronization

Conflict serializability

Global History

Assumption: Each global transaction affects each site.

Assumptions:
- Each site holds a partial database.
- These databases are mutually disjoint.

Definition 1(... Global History):

Let the hetero... ...eous federation consist of n sites, let $T_1, \ldots, T_n$ be sets of loca... ...ransactions at sites 1 , ... n, and let T be a set of global transact... ...ons. Finally, let $h_1, \ldots, h_n$ be local histories s.t. $T_i \subseteq trans(h_i)$ and $T \cap trans(h_i) \neq \emptyset$, $1 \leq i$ ... n.

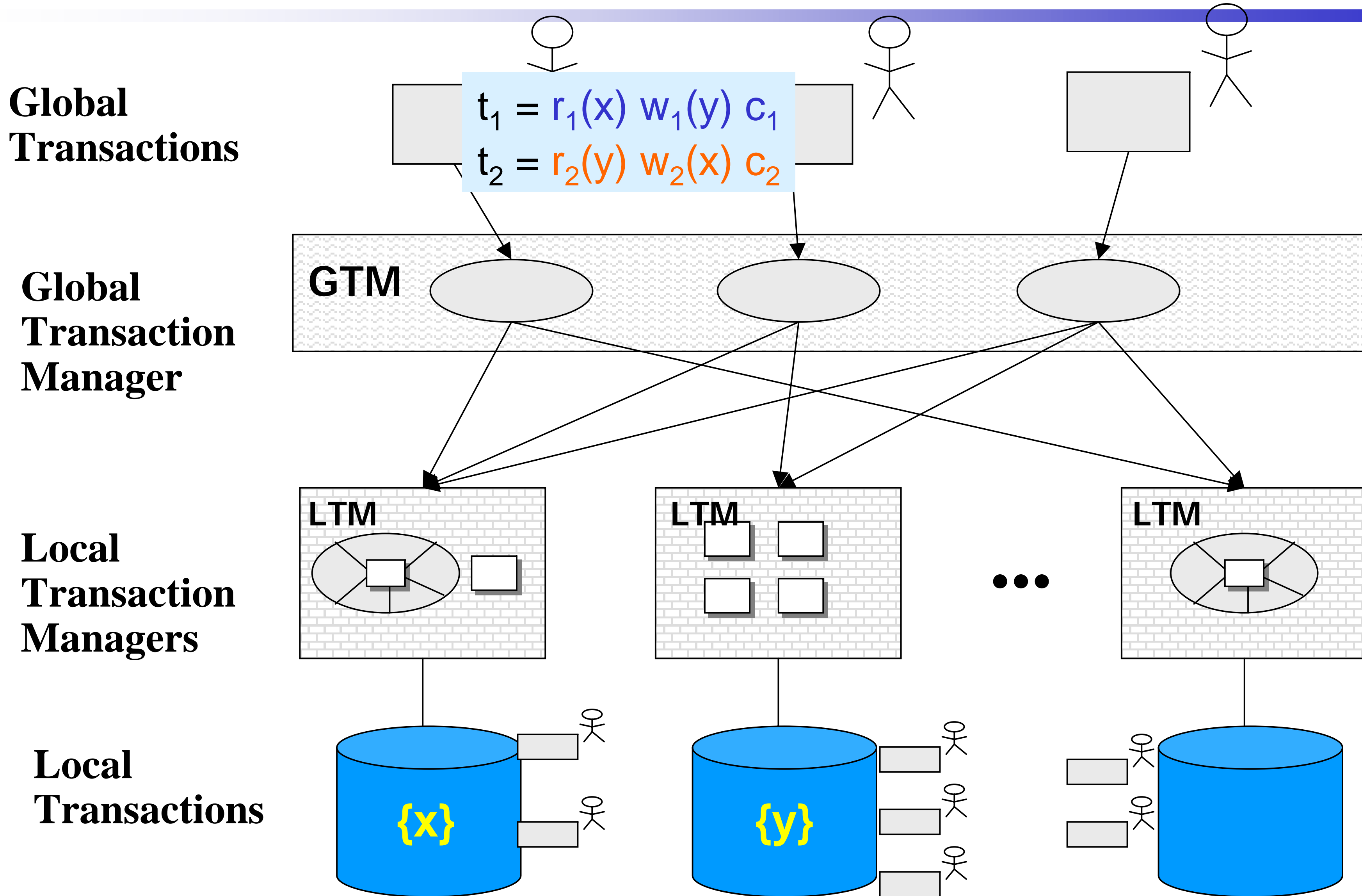
A (heterogeneous) **global history** for h_1 , ... h_n is a history h for all the T_i and T s.t. its local projection equ... ...he local history at each site, i.e.,

$$\Pi_i(h) = h_i \quad \text{for} \quad 1 \leq i \leq n$$

Projection on the operations in h that are executed at site i.

Local history contains the operations executed at that site.

Global vs. Local Serializability



Global vs. Local Serializability

Input sequence: $r_1(x)\ w_1(y)\ r_2(y)\ w_2(x)\ c_1\ c_2$

But: Delays and overtaking may occur!

h_1:

Server 1:	$r_1(x) \quad w_2(x)\ c_1 \quad c_2$	CSR: $t_1 < t_2$
Server 2:	$w_1(y) \quad c_1\ r_2(y)\ c_2$	CSR: $t_1 < t_2$

Global history: $h_1 = r_1(x)\ w_1(y)\ w_2(x)\ c_1\ r_2(y)\ c_2$ CSR: $t_1 < t_2$

h_2:

Server 1:	$r_1(x) \quad w_2(x)\ c_1 \quad c_2$	CSR: $t_1 < t_2$
Server 2:	$r_2(y) \quad c_2\ w_1(y)\ c_1$	CSR: $t_2 < t_1$

Global history: $h_2 = r_1(x)\ r_2(y)\ w_2(x)\ w_1(y)\ c_1\ c_2$ CSR: $t_1 < t_2 < t_1$

Local CSR on all involved servers does not imply global CSR

Global vs. Local Serializability

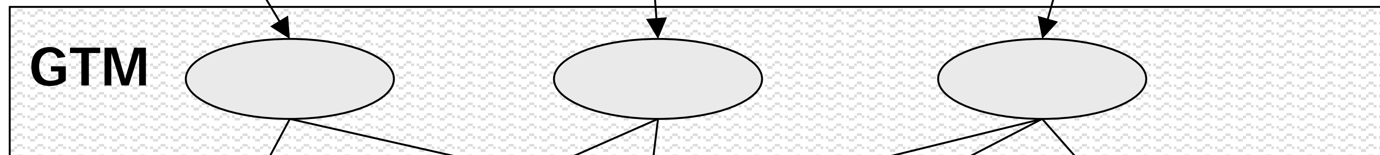
Global Transactions

2 global transactions t_1 = r(a) w(b) c

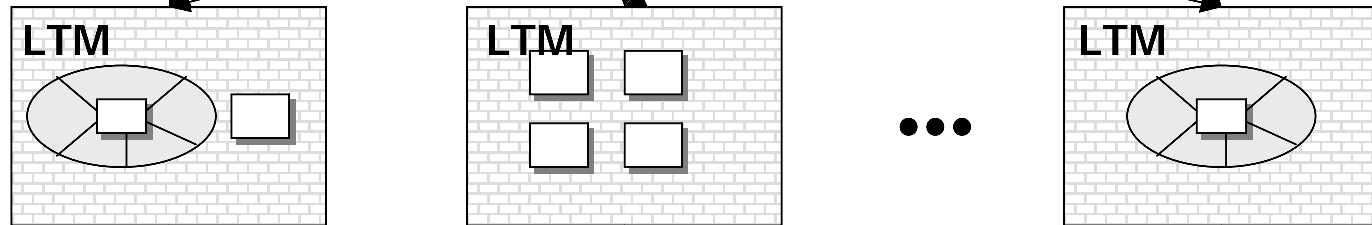
t_2 = w(a) r(c) c

1 local transaction t_3 = r(b) w(c) c

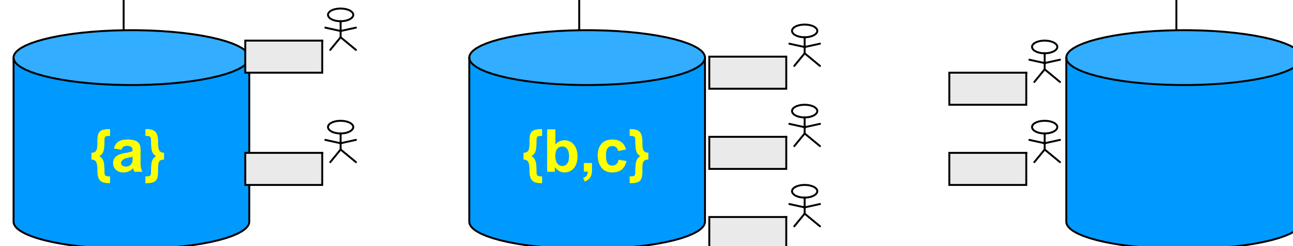
Global Transaction Manager



Local Transaction Managers



Local Transactions



Global vs. Local Serializability

The GTM decides to execute global transactions t_1 and t_2 serially.
CSR: $t_1 < t_2$

Server 1:	$r_1(a)\ c_1$	$w_2(a)\ c_2$		CSR: $t_1 < t_2$
Server 2:	$r_3(b)\ w_1(b)\ c_1$		$r_2(c)\ c_2\ w_3(c)\ c_3$	CSR: $t_2 < t_3 < t_1$

The global schedule is not CSR.
Problem: Local transactions may introduce further conflicts between global transactions.

© 2007 Univ.Karlsruhe, IPD, Prof. Lockemann/Prof. Böhm

Global vs. Local Serializability

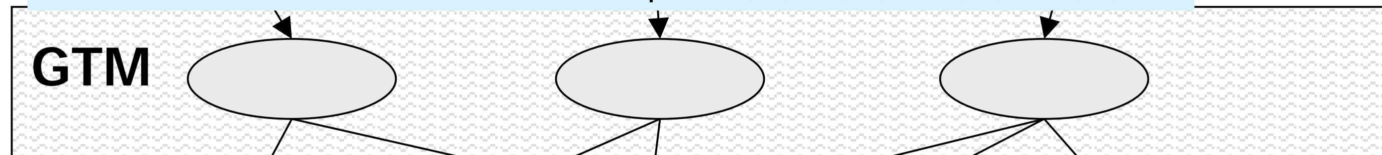
2 global transactions t_1 = r(a) r(d) c
t_2 = r(c) r(b) c

read-only, independent

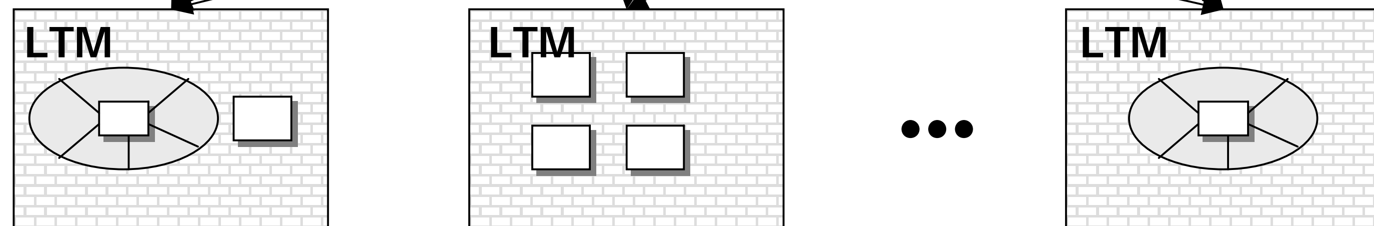
2 local transactions t_3 = r(a) r(b) w(a) w(b) c
t_4 = r(c) r(d) w(c) w(d) c

Global Transactions

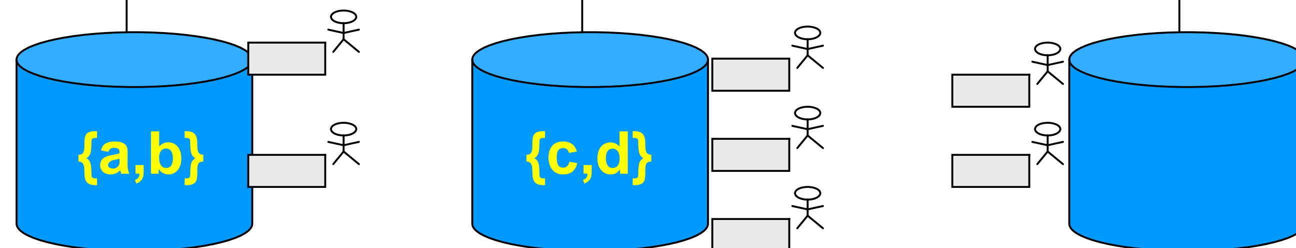
Global Transaction Manager



Local Transaction Managers



Local Transactions



© 2007 Univ.Karlsruhe, IPD, Prof. Lockemann/Prof. Böhm

Global vs. Local Serializability

Server 1: $r_1(a)$ $r_3(a)$ $r_3(b)$ $w_3(a)$ $w_3(b)$ $r_2(b)$ CSR: $t_1 < t_3 < t_2$

Server 2: $r_2(c)$ $r_4(c)$ $r_4(d)$ $w_4(c)$ $w_4(d)$ $r_1(d)$ CSR: $t_2 < t_4 < t_1$

Problem (as before):

Local transactions may introduce (further) conflicts between global transactions

even if the global transactions cannot be in conflict.

© 2007 Univ,Karlsruhe, IPD, Prof. Lockemann/Prof. Böhm

Indirect Conflicts

Definition 10.2 (Direct and Indirect Conflicts):

Let h_i be a local history, and let t, t‘ $\in$ trans(h_i), t $\neq$ t‘.

(i) t and t‘ are in **direct** conflict in h_i if there are two data operations p $\in$ t and q $\in$ t‘ in h_i that access the same data item and at least one of them is a write.

(ii) t and t‘ are in **indirect** conflict in h_i if there exists a sequence t_1 , ... , t_r of transactions in trans(h_i) s.t. t is in h_i in direct conflict with t_1, t_j is in h_i in direct conflict with t_{j+1}, $1 \leq j \leq r-1$, and t_r is in h_i in direct conflict with t‘.

(iii) t and t‘ are in **conflict** in h_i if they are in direct or indirect conflict.

Note: “Conflict” from now on means “direct or indirect conflict.”

Global vs. Local Serializability

Previous example: The GTM decides to execute global transactions t_1 and t_2 serially. CSR: $t_1 < t_2$

Server 1: $r_1(a)$ c_1 $w_2(a)$ c_2 CSR: $t_1 < t_2$
Server 2: $r_3(b)$ $w_1(b)$ c_1 $r_2(c)$ c_2 $w_3(c)$ c_3 CSR: $t_2 < t_3 < t_1$

t_1 and t_2 do **commute** in Site 2, but that leaves their indirect conflict unchanged.

t_1 and t_2 are in indirect conflict.

t_1 and t_2 are in direct conflict.

Global Conflict Graph

Definition 10.3 (Global Conflict Graph):
Let h be a global history for local histories h_1 , ... , h_n; let $G(h_i)$ denote the conflict graph of h_i, $1 \leq i \leq n$.

The **global conflict graph** of h is the graph union of all $G(h_i)$.

Theorem 10.4:
Let h_1 , ... , h_n be local histories s.t. each $G(h_i)$ is acyclic. Let h be a global history for the h_i.

Then h is **globally conflict serializable** iff G(h) is acyclic.

Global Conflict Serializability

Theorem 10.5:
Let h be a global history with local histories h_1 , ... , h_n involving a set T of transactions s.t. each h_i is conflict serializable.
Then h is globally conflict serializable iff there exists a total order "<" on T that is consistent with each local serialization order of the transactions.

Thus, the crucial point in all protocols is to make sure that such a total ordering among the transactions can be established.

Distributed 2-Phase Locking

Distributed 2PL

Perform standard 2PL locally at each site:

For each step the scheduler **requests a lock** on behalf of the step‘s transaction.
Each lock is requested in a specific **mode (read or write)**.
If the data item is not yet locked in an **incompatible mode** the lock is granted;
otherwise there is a **lock conflict** and the transaction becomes **blocked** (suffers a **lock wait**) until the current lock holder **releases the lock**.

Also recall:

A locking protocol is **two-phase (2PL)** if for every output schedule s and every transaction $t_i \in trans(s)$ no ql_i step follows the first ou_i step ($q, o \in \{r, w\}$).

© 2007 Univ.Karlsruhe, IPD, Prof. Lockemann/Prof. Böhm

Consistent Global Ordering

All subtransactions of a given transaction must have identical equivalence times. ⇒ **Enforce global equivalence time**

- Local equivalence time can be controlled from outside:
  - Local equivalence time = commit time ⇒ all global transactions keep their locks up to the local end ⇒ all local histories must be RG.
- GTM must ensure that all local commits of a global transaction take place simultaneously.
  - Requires commit synchronization.

© 2007 Univ.Karlsruhe, IPD, Prof. Lockemann/Prof. Böhm

Synchronization

Definition 10.6 (commit-deferred)

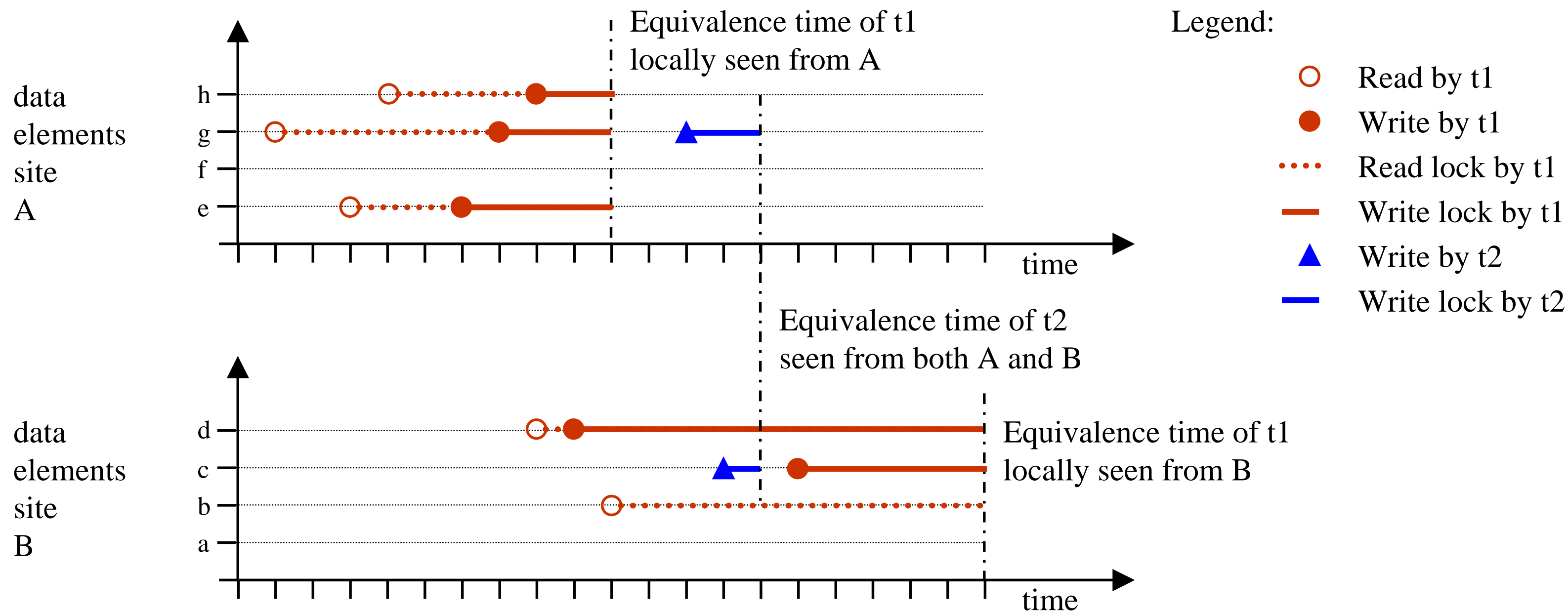
A global transaction t is **commit-deferred** if its commit is only communicated by GTM to all LTM after the local completion of all data operations of t was acknowledged.

Theorem 10.7

Let h be a global history for $h_1,\ldots,h_n$. If $h_i \in RG$, $1 \leq i \leq n$, and all global transactions are commit-deferred, then h is **global conflict serializable**.

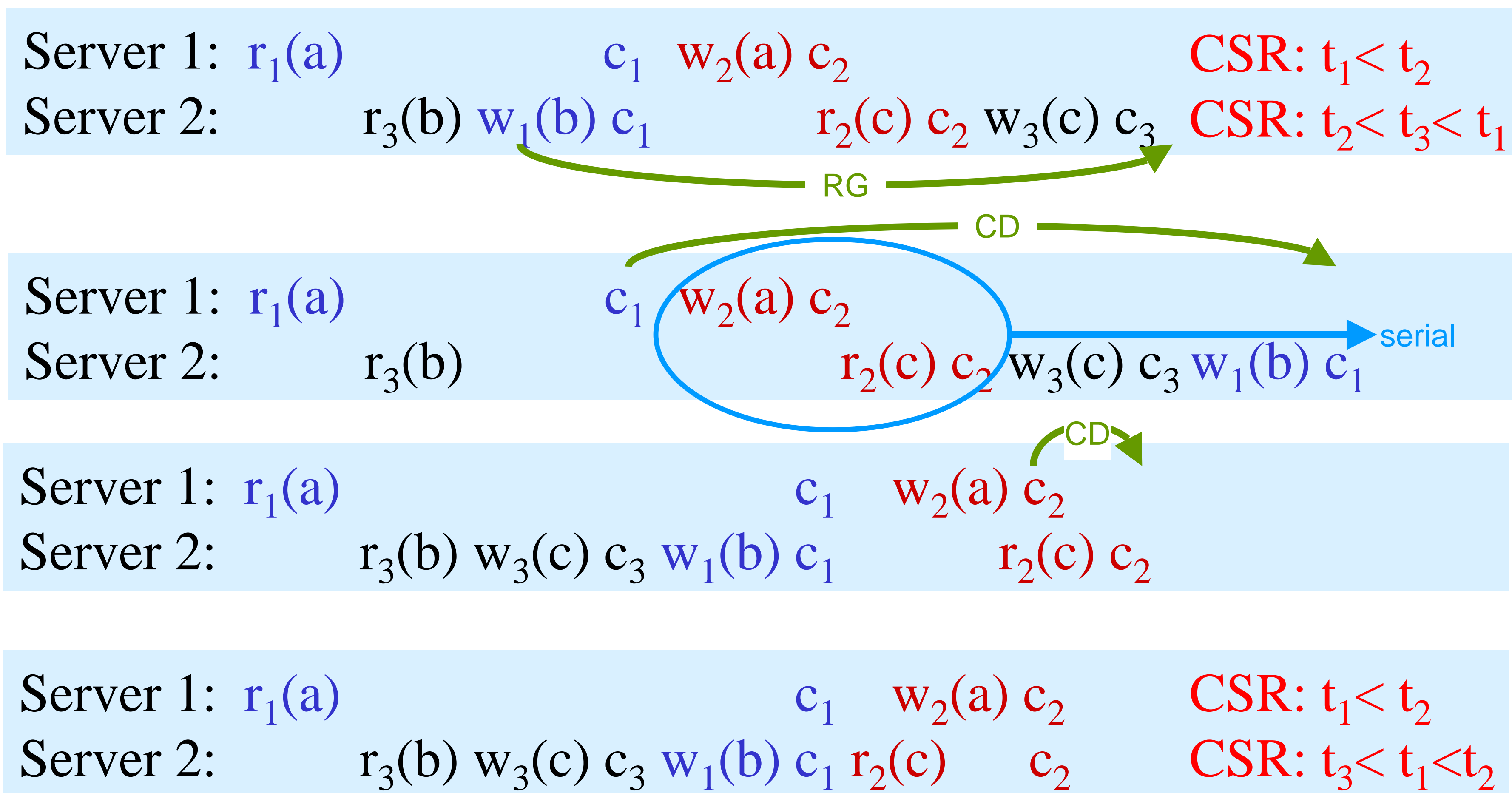
Need for commit-deferred

- Consider transactions t1 und t2 that locally follow 2PL, but are not globally serializable:



Global vs. Local Serializability

Previous example: The GTM decides to execute global transactions t_1 and t_2 serially. CSR: $t_1 < t_2$

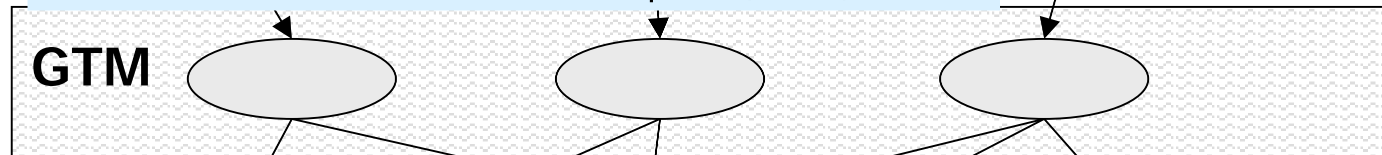


Need for RG (1)

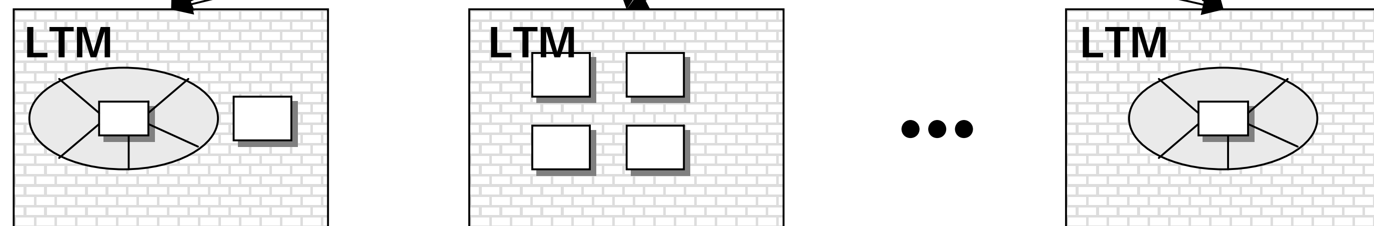
2 global transactions t_1 = w(a) w(d) c
t_2 = w(c) w(b) c

2 local transactions t_3 = r(a) r(b) c
t_4 = r(c) r(d) c

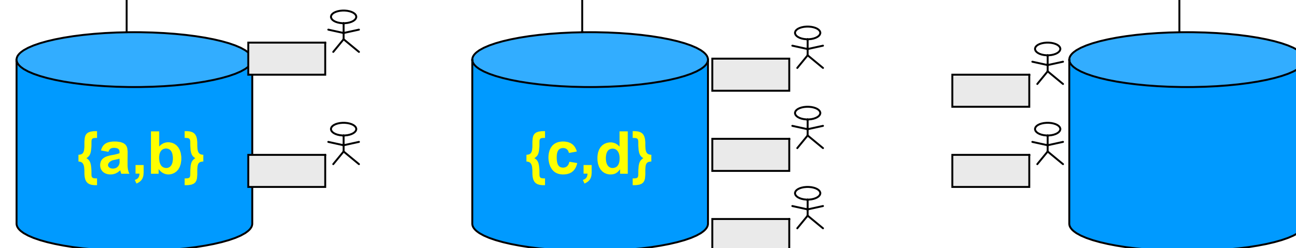
Global Transactions



Global Transaction Manager



Local Transaction Managers



Local Transactions

Need for RG (2)

The GTM decides to execute global transactions t_1 and t_2 in the order $w_1(a)$ $w_1(d)$ $w_2(c)$ $w_2(b)$ c_1 c_2. CSR: $t_1 < t_2$

But there may be different communication delays:

Server 1: $h_1 = w_1(a)\ r_3(a)\ r_3(b)\ c_3\ w_2(b)$

Server 2: $h_2 = w_2(c)\ r_4(c)\ r_4(d)\ c_4\ w_1(d)$

GTM issues c_1 c_2:

Server 1: $h_1 = w_1(a)\ r_3(a)\ r_3(b)\ c_3\ w_2(b)\ c_1\ c_2$ CSR: $t_1 < t_3 < t_2$

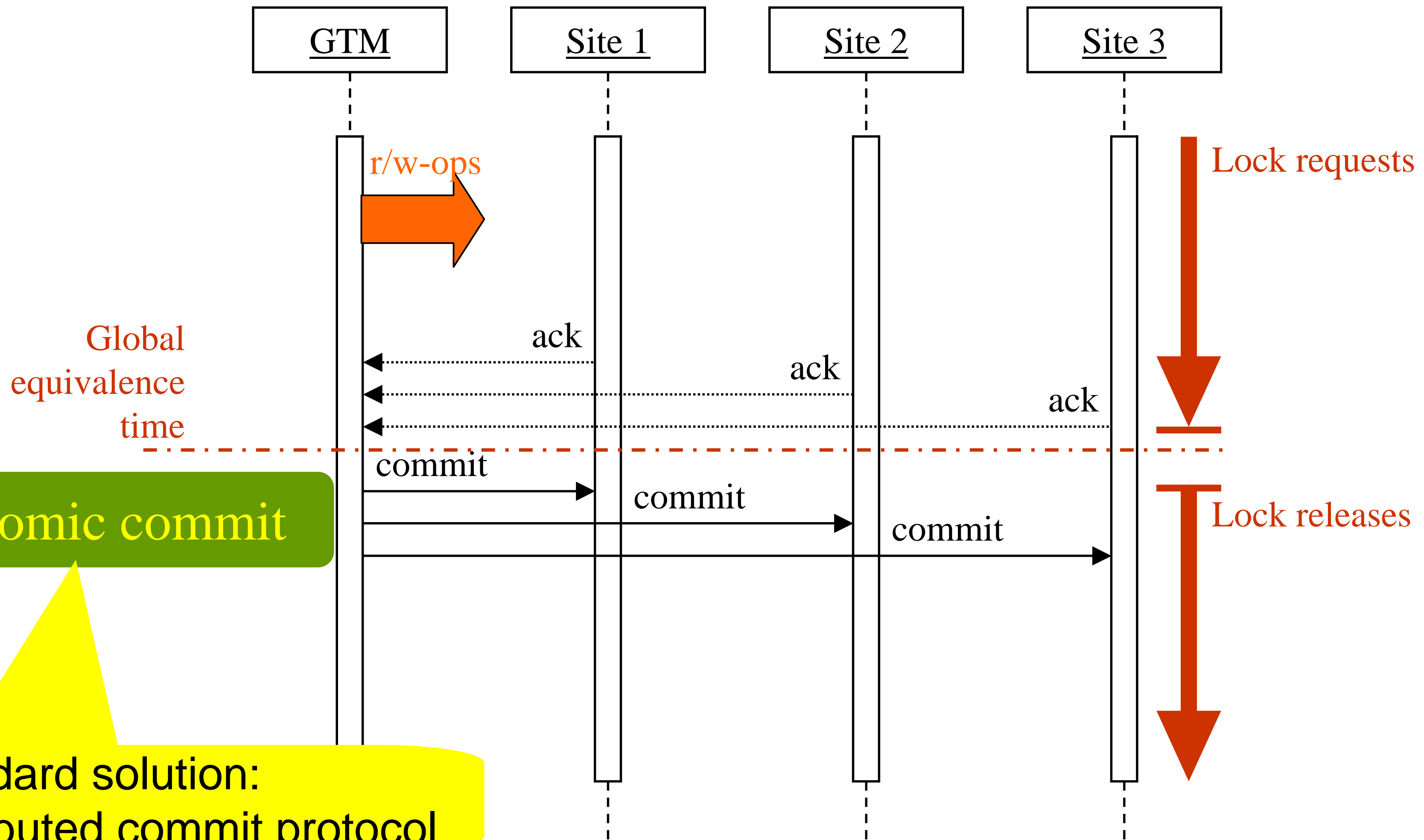
Server 2: $h_2 = w_2(c)\ r_4(c)\ r_4(d)\ c_4\ w_1(d)\ c_1\ c_2$ CSR: $t_2 < t_4 < t_1$

Permitted under (simple) 2PL.

Under RG: $w_2(c)\ w_1(d)\ c_1\ c_2\ r_4(c)\ r_4(d)\ c_4$ CSR: $t_2 < t_1 < t_4$ or $t_1 < t_2 < t_4$

and similarly for Server 1.

Rigorous / commit-deferred



Standard solution:
distributed commit protocol

Solves distributed recovery problem as well, see there!

Light-weight solution: Tickets (1)

- Idea: Avoid different ordering of serially submitted global transactions by forcing direct conflicts no matter whether there are indirect conflicts.
- Only minimal LTM requirements: CSR.
- Additional data object in each database („ticket“).
- Each global transaction
  - reads ticket,
  - writes back incremented value.

„Take-a-ticket“ operation.

Light-weight solution: Tickets (2)

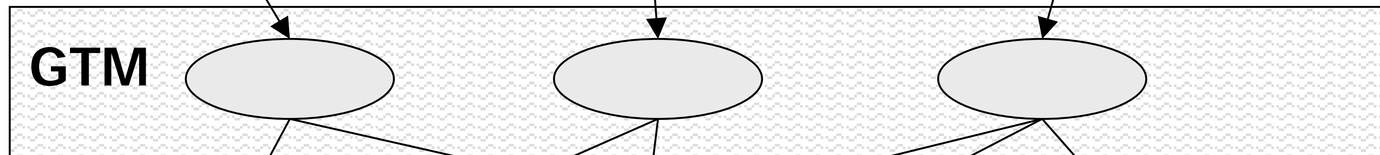
Global Transactions

2 global transactions t_1 = r(a) w(b) c

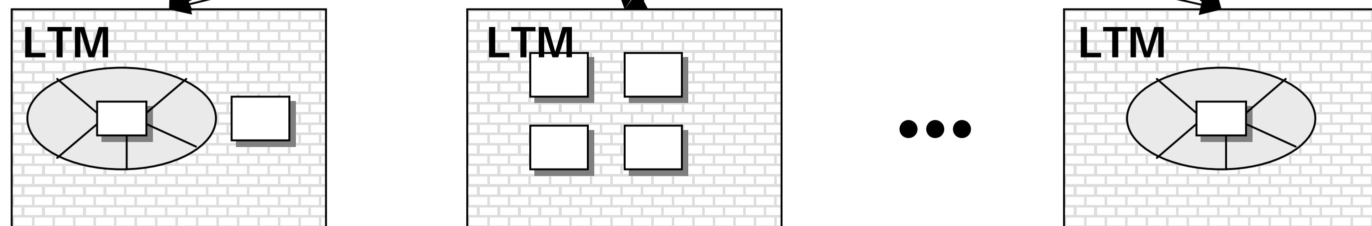
t_2 = w(a) r(c) c

1 local transaction t_3 = r(b) w(c) c

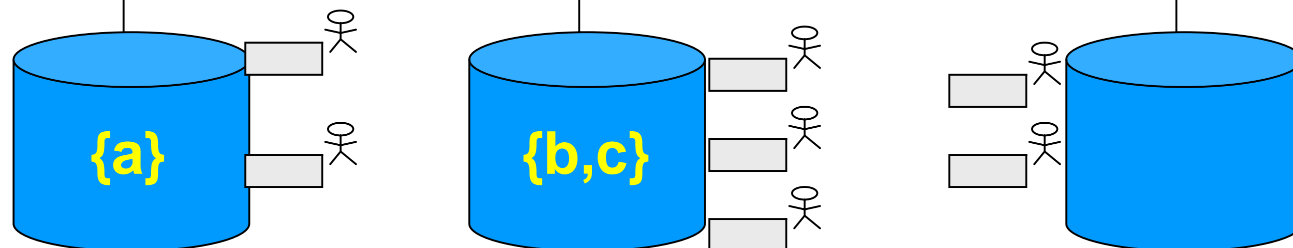
Global Transaction Manager



Local Transaction Managers



Local Transactions



© 2007 Univ.Karlsruhe, IPD, Prof. Lockemann/Prof. Böhm

Light-weight solution: Tickets (3)

- Assume t_1 t_2.
- Possible submissions:
  - $h_1 = r_1(a)\ c_1\ w_2(a)\ c_2$ — CSR: $t_1 < t_2$
  - $h_2 = r_3(b)\ w_1(b)\ c_1\ r_2(c)\ c_2\ w_3(c)\ c_3$ — CSR: $t_2 < t_3 < t_1$
- 2PL without tickets:
  - $h_1 = r_1(a)\ c_1\ w_2(a)\ c_2$ — CSR: $t_1 < t_2$
  - $h_2 = r_3(b)\ r_2(c)\ c_2\ w_3(c)\ c_3\ w_1(b)\ c_1$ — CSR: $t_2 < t_3 < t_1$
- Use of tickets:
  - $h_1 = r_1(tc_1)\ w_1(tc_1+1)\ r_1(a)\ c_1\ r_2(tc_1)\ w_2(tc_1+1)\ w_2(a)\ c_2$
  - $h_2 = r_3(b)\ r_1(tc_2)\ w_1(tc_2+1)\ w_3(c)\ c_3\ w_1(b)\ c_1\ r_2(tc_2)\ w_2(tc_2+1)\ r_2(c)\ c_2$ — CSR: $t_3 < t_1 < t_2$

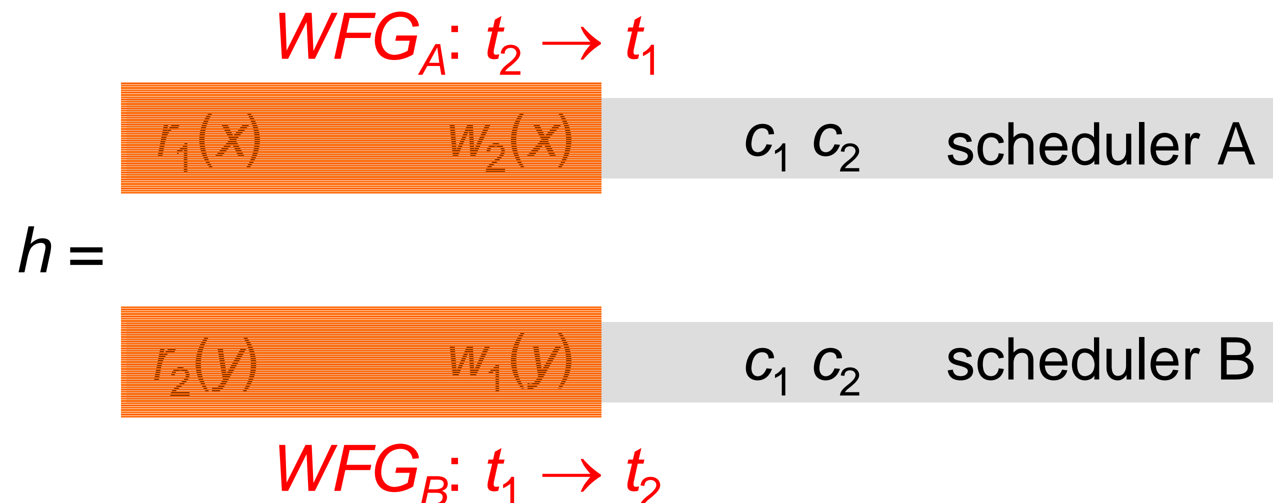
Distributed Deadlocks

How they occur

- Since nodes are autonomous, each local scheduler S_i maintains its own local waits-for graph WFG_i.
- Local WFGs do not suffice!
- Example:

$t_1 = r_1(x)\ w_1(y)\ c_1$

$t_2 = r_2(y)\ w_2(x)\ c_2$



Centralized deadlock detection

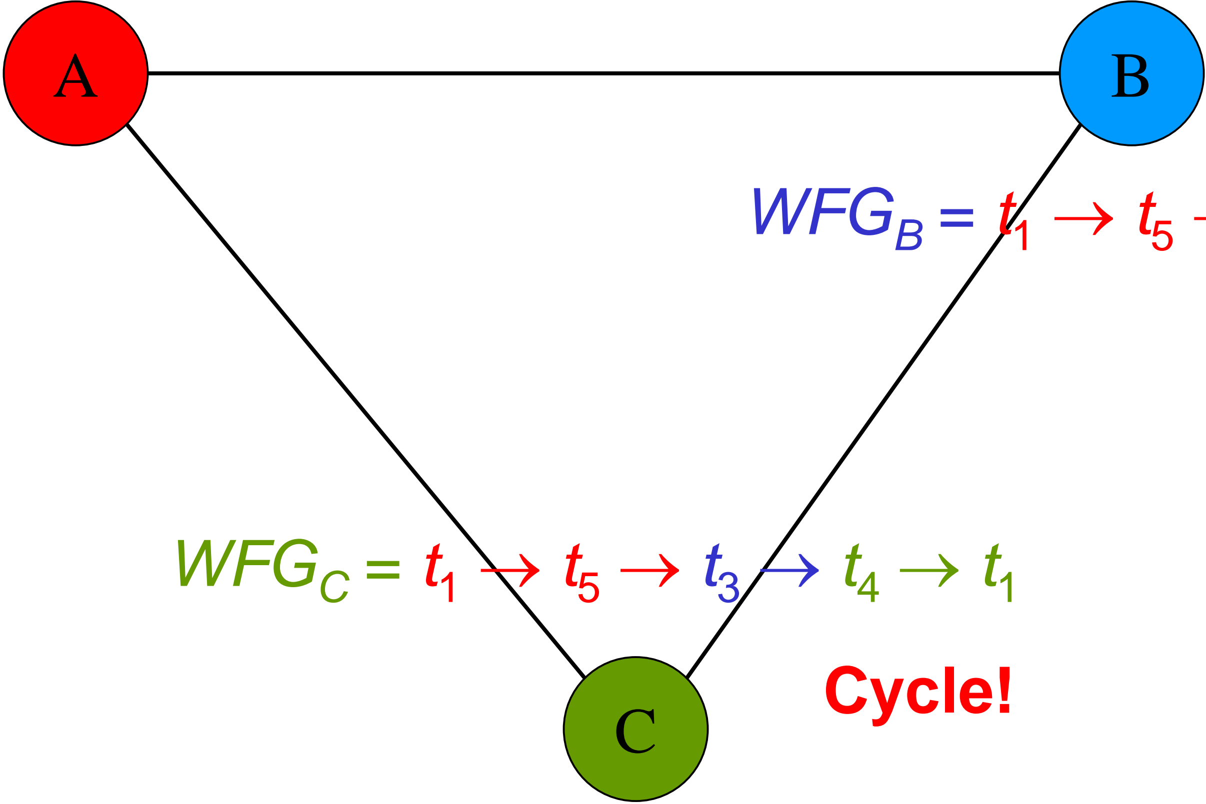
Centralized detection of global deadlocks:

- A singular site maintains the global WFG as the union of all local WFG_i. Schedulers S_i send (parts of) their WFG_i to the central site.
- Clearly: Deadlocks are guaranteed to be detected.
- Price:
  - communication overhead,
  - central site as a single point of failure, performance bottleneck
  - delayed detection due to only periodic transmission of the WFG_i,
  - false positives (phantom deadlocks) due to delayed communication of aborts.

Distributed deadlock detection (1)

- If cycle is detected in local WFG, abort one TA in the cycle $\Rightarrow$ global abort necessary (see “Recovery”).
- Otherwise **Path Pushing**:
  - Suppose WFG_i includes path $t_i \rightarrow \ldots \rightarrow t_j$.
  - Send the path to each node suspected of blocking t_j.
  - Each node adds the path to its own WFG.
  - If no deadlock but a new suspicious path, push the new path.
  - Each deadlock will ultimately be detected.

Distributed deadlock detection (2)



Distributed Timestamp Ordering

Distributed TO

- Local TO as known before.
- Each scheduler assigns to each transaction arriving at its site a time stamp.
  - Global effect requires that time stamps are globally unique.
    $\Rightarrow$ Only then can each scheduler apply the TO protocol locally and independently. $\Rightarrow$ Distribution does not raise new problems. $\Rightarrow$ No communication needed among the schedulers.
- Distributed TO is particularly simple and requires little overhead. $\Rightarrow$ We pay the usual price of a small subset of *CSR*.

Uniqueness of time stamps (1)

Required: Method for assigning globally unique time stamps without the need for a central authority or for communication among the sites.

Solution:

- Within a node achieve uniqueness by simply running a counter and assigning its value to the next (local) time stamp (*relative time*).
- Compute the global time stamp of transaction t from the relative time z_t, the total number k of nodes and the unique node ID (e.g., a number i) of the affected site K_i:

$$ts(t) = z_t * k + i$$

Uniqueness of time stamps (2)

Example:

t_1 starts at K_1 and t_2 at K_2, and let $z_{t_1} = z_{t_2} = 0$ (i.e., t_1 and t_2 are the first transactions executed at each node). Then

$$ts(t_1) = 0 * 2 + 1 = 1$$

$$ts(t_2) = 0 * 2 + 2 = 2$$

Remarks:

- In general, order of time stamps does not correspond to order of arrival.

  $\Rightarrow$ Method is in general not fair.

  - Strict fairness only if nodes communicate.

Weaker serializability

Resume

- Rigorous/commit-deferred:
  - Minus: Harmful to autonomy because enforcement of *RG* and potential waits for global commit.
  - Plus: Natural combination with recovery.
  - But: Local transactions participate in the Minus without profiting from the Plus!

Relax serializability?

- Tickets and Distributed TO:
  - Plus: Autonomy practically preserved.
  - Minus: Separate solution for recovery needed, small *CSR* subset.

Quasi-Serialisierbarkeit (1)

Definition 10.8

Eine Menge $\{h_1,\ldots,h_n\}$ lokaler Historien heißt **quasi-seriell**, falls gilt

- $h_k \in CSR$, $1 \leq k \leq n$, und
- es gibt eine Totalordnung < auf der Menge T globaler Transaktionen, so dass $t_i < t_j$ für $t_i, t_j \in T$, $i \neq j$ impliziert, dass in jedem h_k, $1 \leq k \leq n$, die t_i-Subtransaktion *vollständig* vor der t_j-Subtransaktion steht (*falls beide in h_k vorkommen*).

Quasi-Serialisierbarkeit (2)

Definition 10.9

Eine Menge $\{h_1,\ldots,h_n\}$ lokaler Historien heißt **quasi-serialisierbar**, falls es eine quasi-serielle Menge $\{h'_1,\ldots,h'_n\}$ lokaler Historien gibt, so dass $h_i \equiv_C h'_i$ für $1 \leq i \leq n$.

Definition 10.10

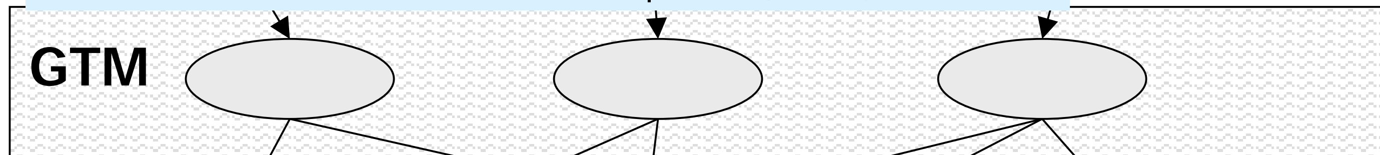
Eine globale Historie heißt **quasi-serialisierbar**, falls die Menge ihrer Lokalprojektionen quasi-serialisierbar ist.

Definition quasi-seriell betrachtet nur globale Transaktionen
$\Rightarrow$ Quasi-Serialisierbarkeit ignoriert lokale Transaktionen

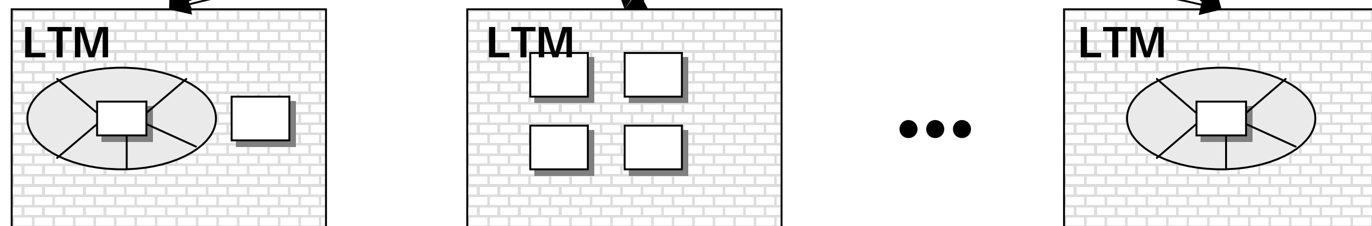
Quasi-Serialisierbarkeit (3)

2 global transactions	t_1 = w(a) r(d) c
	t_2 = r(b) r(c) w(e) c
2 local transactions	t_3 = r(a) w(b) c
	t_4 = w(d) r(e) c

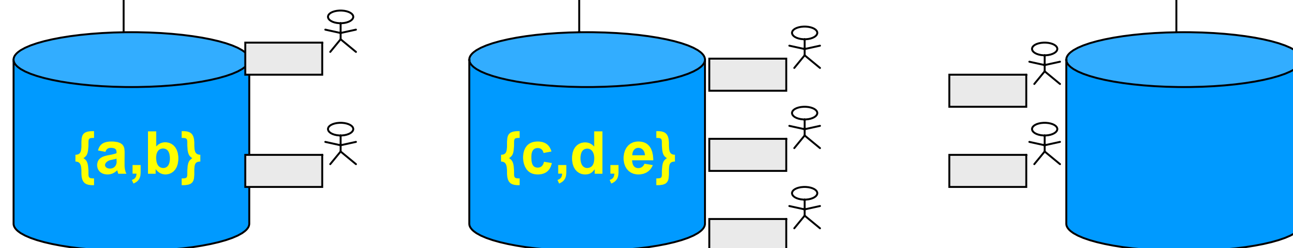
Global Transactions



Global Transaction Manager



Local Transaction Managers



Local Transactions

Quasi-Serialisierbarkeit (4)

Server 1: $h_1 = w_1(a)\ c_1\ r_3(a)\ w_3(b)\ c_3\ r_2(b)\ c_2$ CSR: $t_1 < t_3 < t_2$

Server 2: $h_2 = r_2(c)\ w_4(d)\ r_1(d)\ c_1\ w_2(e)\ c_2\ r_4(e)\ c_4$ CSR: $t_2 < t_4 < t_1$

$\equiv_C w_4(d)\ r_1(d)\ c_1\ r_2(c)\ w_2(e)\ c_2\ r_4(e)\ c_4$

seriell: $t_1 < t_3 < t_2$

quasi-seriell (in beiden t_1 vor t_2)

h_1 und h_2 sind Projektionen der (somit quasi-serialisierbaren) Historie

$h = w_1(a)\ r_3(a)\ r_2(c)\ w_4(d)\ r_1(d)\ c_1\ w_3(b)\ c_3\ r_2(b)\ w_2(e)\ c_2\ r_4(e)\ c_4$

Zweistufige Serialisierbarkeit (1)

Weitere Abschwächung der Quasi-Serialisierbarkeit:

Der GTM kann die Serialisierbarkeit globaler Historien garantieren, die LTMs können die Serialisierbarkeit lokaler Schedules gewährleisten. Daher:

Definition 10.11

Eine globale Historie *h* heißt **zweistufig serialisierbar**, falls gilt:

- $\forall i, 1 \leq i \leq n \;\; \Pi_i(h) = h_i \in CSR$
- Ist *T* die Menge der in *h* vorkommenden globalen Transaktionen, so gilt $\Pi_T(h) \in CSR$

Anmerkung:

Jede global serialisierbare Historie ist zweistufig serialisierbar. Die Umkehrung dieses Satzes gilt nicht.

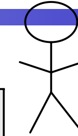
Zweistufige Serialisierbarkeit (2)

Global Transactions

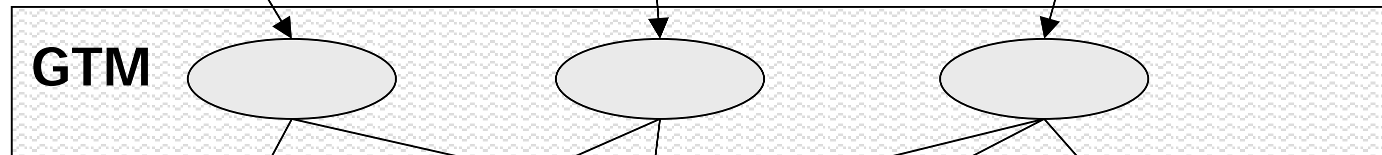
2 global transactions t_1 = r(a) w(c) w(d) c

t_2 = r(a) w(b) w(d) c

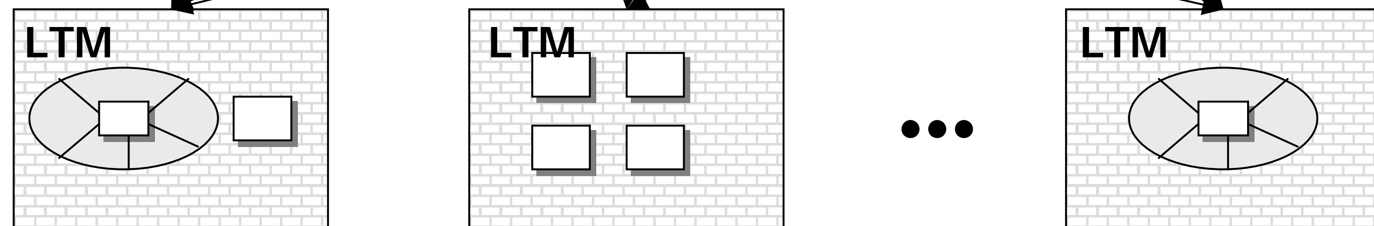
1 local transaction t_3 = w(a) c



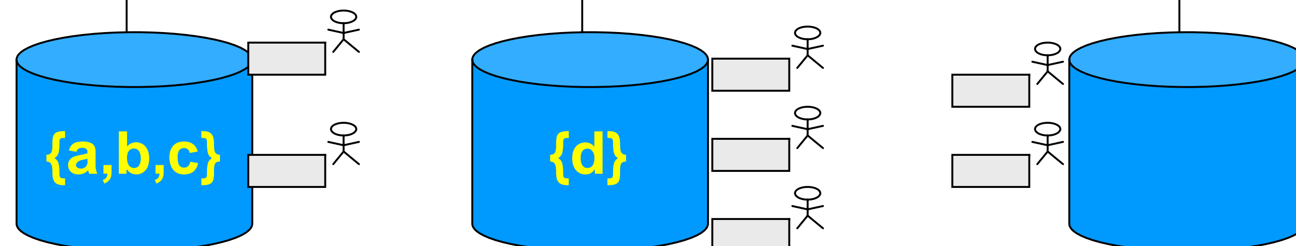
Global Transaction Manager



Local Transaction Managers



Local Transactions



Zweistufige Serialisierbarkeit (3)

Server 1: $h_1 = r_1(a)\ w_1(c)\ c_1\ w_3(a)\ c_3\ r_2(a)\ w_2(b)\ c_2$ CSR: $t_1 < t_3 < t_2$

Server 2: $h_2 = w_2(d)\ c_2\ w_1(d)\ c_1$ CSR: $t_2 < t_1$

h_1, h_2 seriell

h_1 und h_2 sind Projektionen der Historie $\Pi_T(h) \in$ CSR: $t_2 < t_1$

$h = r_1(a)\ w_1(c)\ w_2(d)\ w_3(a)\ c_3\ w_1(d)\ c_1\ r_2(a)\ w_2(b)\ c_2$

h nicht *CSR*!