8. Distributed Concurrency Control

General reference architecture.

Federated system
8.1: Preliminaries

Sites and subtransactions

- Let be given a fixed number of sites across which the data is distributed. The server at site $i$, $1 \leq i \leq n$ is responsible for a (finite) set $D_i$ of data items. The corresponding global database is given as $D = \bigcup_{i=1}^{n} D_i$.

- Data items are not replicated; thus $D_i \cap D_j = \emptyset$, $i \neq j$.

- Let $\mathcal{T} = \{T_1, \ldots, T_m\}$ be a set of transactions, where $T_i = (OP_i, <_i)$, $1 \leq i \leq m$.

- Transaction $T_i$ is called *global*, if its actions are running at more than one server; otherwise it is called *local*.

- The part of a transaction $T_i$ being executed at a certain site $j$ is called *subtransaction* and is denoted by $T_{ij}$.
Parallelism as prerequisite for distributed execution

A transaction $T$ is a partial order $\prec$ of actions in $OP$, $T = (OP, \prec)$, where $OP$ is a finite set of $T$’s actions $RX$ and $WX$, where $X$ is a data item.

Moreover, $\prec \subseteq OP \times OP$ is a partial order on $OP$ which fulfills the following properties:

- Each data item is read and written by $T$ at most once.
- If $p$ is a read action and $q$ is a write actions of $T$ and both access the same data item, then $p < q$.

Complete transaction

We call a transaction complete, if its first action is begin $b$ and its last action either is commit $c$ or abort $a$.

---

1A binary relation is a partial order, if it is reflexive, antisymmetric and transitive.
Histories and schedules

Let \( \mathcal{T} = \{T_1, \ldots, T_n\} \) be a (finite) set of complete transactions, where for each \( T_i \) we have \( T_i = (OP_i, <_i) \).

A history of \( \mathcal{T} \) is a pair \( S = (OP_S, <_S) \), where

1. \( OP_S = \bigcup_{i=1}^{n} OP_i \) and \( <_S \) a partial order on \( OP_S \) such that \( <_S \supseteq \bigcup_{i=1}^{n} <_i \).
2. Let \( p, q \in OP_S \), where \( p \) and \( q \) belong to distinct transactions, however access the same data object. If \( p \) or \( q \) is a write action, then either \( p <_S q \) or \( q <_S p \); we say, \( p \) and \( q \) are in conflict; if \( p <_S q \) and \( p \) and \( q \) are in conflict, we write \( (p, q) \in \text{conf}(S) \).

A schedule of \( \mathcal{T} \) is a prefix of a history.\(^2\)

Conflict graph

The conflict graph of a schedule \( S \) is given as \( G(S) = (V, E) \), where \( V \) is the set of transactions in \( S \) and the set of edges \( E \) is given by the conflicts in \( S \): \( T_i \rightarrow T_j \in E \), iff there are conflicting actions \( p \in OP_i, q \in OP_j \) and \( p <_S q \).

\(^2\)A partial order \( L' = (A', <') \) is a prefix of a partial order \( L = (A, <) \), if \( A' \subseteq A \), \( <' \subseteq < \), for all \( a, b \in A' \): \( a <' b \) if \( a < b \), and for all \( p \in A, q \in A' \): \( p < q \Rightarrow p <' q \).
A parallel debit/credit transaction. \textit{b}: BEGIN; \textit{c}: COMMIT.

When transactions are depicted as directed graphs, we omit transitive edges.

Two parallel debit/credit transactions, each prepared for parallel execution.

\[ \Rightarrow \text{Definition of a schedule? Definition of serializability?} \]
Two parallel debit/credit transactions, each prepared for parallel execution.

Transaction $T_1$

Transaction $T_2$

Locally observable schedules of the two transactions when executed in parallel by CPU PA and CPU PB.

(i) $PA : R_1 A W_1 A R_2 A W_2 A$
$PB : R_1 B W_1 B R_2 B W_2 B$

(ii) $PA : R_1 A W_1 A R_2 A W_2 A$
$PB : R_2 B W_2 B R_1 B W_1 B$

On each CPU in both cases the local schedules are serializable - however, globally, in the second case the transactions are not executed in a serializable manner!
A schedule/history of the two parallel debit/credit transactions.

![Schedule Diagram]

The schedule is not serializable as its conflict graph is cyclic.

### Serializability

- A schedule $S = (OP_S, <_S)$ is *serial*, if for any two transactions $T_1, T_2$ appearing in $S$, $<_S$ orders all actions of $T_1$ before all actions of $T_2$, or vice versa.
- A schedule is called (conflict-)serializable, if there exists a (conflict-)equivalent serial schedule over the same set of transactions.
- A schedule $S = (OP_S, <_S)$ is serializable, iff its conflict graph is acyclic.

---

We consider only conflict-serializability and therefore talk about serializability in the sequel, for short.
Local and global schedules

We are interested in deciding whether or not the execution of a set of transactions is serializable, or not.

- At the local sites we can observe an evolving sequence of the respective transactions’ actions.
- We would like to decide whether or not all these locally observable sequences imply a (globally) serializable schedule.
- However, on the global level we cannot observe an evolving sequence, as there does not exist a notion of global physical time.
Example

Schedule:

Observed local schedules:

Site 1 (PA) : $R_1 A W_1 A R_2 A W_2 A$
Site 2 (PB) : $R_2 B W_2 B R_1 B W_1 B$

Can schedules be represented as action sequences, as well?

... yes, we call them *global schedules*. 
From now on local and global schedules are sequences of actions!

Let $\mathcal{T} = \{T_1, \ldots, T_m\}$ be a set of transactions being executed at $n$ sites. Let $S_1, \ldots, S_n$ be the corresponding local schedules.

A global schedule of $\mathcal{T}$ with respect to $S_1, \ldots, S_n$ is any sequence $S$ of the actions of the transactions in $\mathcal{T}$, such that its projection onto the local sites equals the corresponding local schedules $S_1, \ldots, S_n$.

Example

Consider local schedules $S_1 = R_1 A W_2 A$ and $S_2 = W_1 B R_2 B$.

Global schedules: $S : R_1 A W_1 B W_2 A R_2 B$,
$S' : R_1 A W_1 B R_2 B W_2 A$

Not a global schedule: $S'' : R_1 A R_2 B W_1 B W_2 A$
Examples where there does not exist a serializable global schedule

- $T_1 = R_1A W_1B$, $T_2 = R_2C W_2A$ are global transactions and $T_3 = R_3B W_3C$ is a local transaction.

  $S_1: R_1A \ W_2A$
  $S_2: R_3B \ W_1B \ R_2C \ W_3C$

  Note, in $S_2$ subtransactions $T_{12}$ and $T_{22}$ have no conflicting actions!

- $T_1 = RA RD$ und $T_2 = RB RC$ are global transactions, while $T_3 = RA RB WA WB$ and $T_4 = RD WD RC WC$ are local transactions.

  $S_1: R_1A \ R_3A \ R_3B \ W_3A \ W_3B \ R_2B$
  $S_2: R_4D \ W_4D \ R_1D \ R_2C \ R_4C \ W_4C$

  Note, both global transactions are only reading and, in particular, disjoint data sets!

In both examples the local schedules are serializable, however no serializable global schedule exists.
Serializability of global schedules

- As we do not have replication of data items, whenever there is a conflict in a global schedule, the same conflict must be part of exactly one local schedule.

- Consequently, the conflict graph of a global schedule is given as the union of the conflict graphs of the respective local schedules.

- In particular, given a set of local schedules, either all or none corresponding global schedule is serializable.
8. Distributed Concurrency Control

8.1. Preliminaries

Examples

\[ S_1 : R_1A \quad W_1A \quad R_2A \quad W_2A \]
\[ S_2 : R_2B \quad W_2B \quad R_1B \quad W_1B \]

\[ S_1 : R_1A \quad W_2A \]
\[ S_2 : R_3B \quad W_1B \quad R_2C \quad W_3C \]

\[ S_1 : R_1A \quad R_3A \quad R_3B \quad W_3A \quad W_3B \quad R_2B \]
\[ S_2 : R_4D \quad W_4D \quad R_1D \quad R_2C \quad R_4C \quad W_4C \]
Types of federation

- **homogeneous federation:**
  
  Same services and protocols at all servers. Characterized by distribution transparency: the federation is perceived by the outside world as if it were not distributed at all.

- **heterogenous federation:**

  Servers are autonomous and independent of each other; no uniformity of services and protocols across the federation.

Interface to recovery

Every global transactions runs the 2-phase-commit protocol. By that protocol the subtransactions of a global transaction synchronize such that either all subtransactions commit, or none of them, i.e. all abort. Details are given in Chapter 10.
8.2: Homogeneous Concurrency Control

Serializability by distributed 2-Phase Locking (2PL)

A transactions entry into the unlock-phase has to be synchronized among all sites the transaction is being executed.

**Primary Site 2PL:**

- One site is selected at which lock maintenance is performed exclusively.
- This site thus has global knowledge and enforcing the 2PL rule for global and local transactions is possible.
- The lock manager simply has to refuse any further locking of a subtransaction $T_{ij}$ whenever a subtransaction $T_{ik}$ has started unlocking already.
- Much communication is resulting which may create a bottleneck at the primary site.

**Example**

$S_1 : R_1A \quad W_1A \quad R_2A \quad W_2A$

$S_2 : R_2B \quad W_2B \quad R_1B \quad W_1B$
**Distributed 2PL:**

- When a server wants to start unlocking data items on behalf of a transaction, it communicates with all other servers regarding the lock point of the other respective subtransaction.
- The server has to receive a *locking completed*-message from each of these servers.
- This implies extra communication between servers.

**Example**

$$S_1 : \ R_1 A \quad W_1 A \quad R_2 A \quad W_2 A$$

$$S_2 : \ R_2 B \quad W_2 B \quad R_1 B \quad W_1 B$$
Distributed Strong 2PL:

- Every subtransaction of a global transaction and every local transaction holds locks until commit.
- Then by the 2-phase-commit protocol the 2PL-rule is enforced as a side-effect.

Applying strong 2PL the global 2PL-property is self-guaranteed without any explicit measures!
Locking protocols are prone to deadlocks!

**Global deadlock**

Diagram showing a global deadlock among three servers: A, B, and C. Each server contains a transaction with two operations: waiting for a lock and waiting for a message. The diagram illustrates the deadlock situation where each transaction is waiting for a lock held by another transaction.
Global deadlock detection is difficult. Detection strategies:

- **Centralized detection**: Each site maintains its local wait-for graph. One distinguished site is selected to which all local wait-for graphs are send periodically. The selected site computes the union of all local wait-for graphs and checks for deadlocks.

- **Time-out based detection**: Whenever during a wait a time-out occurs, the respective transaction decides for a deadlock and aborts itself.

- **Edge chasing**: Whenever a transaction $T$ waits for a transaction $T'$, it sends its identification to $T'$. Whenever a transaction $T'$ receives such a message, it sends the identification of such $T$ to all transactions it is waiting for. If a transaction receives its own identification, it decides for a deadlock and it aborts itself.

- **Path pushing**:
  
  (i) Each server that has a waits-for path from transaction $t_i$ to transaction $t_j$ such that $T_i$ has an incoming waits-for-message edge and $T_j$ has an outgoing waits-for-message edge sends that path to the server along the outgoing edge.

  (ii) Upon receiving a path the server concatenates this with the local paths that already exist, and forwards the result along its outgoing edges again. If there exists a cycle among $k$ servers, at least one of them will detect the cycle in at most $k$ rounds.
Global deadlock detection is difficult. Detection strategies:

- **Centralized detection**: Each site maintains its local wait-for graph. One distinguished site is selected to which all local wait-for graphs are send periodically. The selected site computes the union of all local wait-for graphs and checks for deadlocks.

- **Time-out based detection**: Whenever during a wait a *time-out* occurs, the respective transaction decides for a deadlock and aborts itself.

- **Edge chasing**: Whenever a transaction $T$ waits for a transaction $T'$, it sends its identification to $T'$. Whenever a transaction $T'$ receives such a message, it sends the identification of such $T$ to all transactions it is waiting for. If a transaction receives its own identification, it decides for a deadlock and it aborts itself.

- **Path pushing**:
  
  (i) Each server that has a waits-for path from transaction $t_i$ to transaction $t_j$ such that $T_i$ has an incoming waits-for-message edge and $T_j$ has an outgoing waits-for-message edge sends that path to the server along the outgoing edge.

  (ii) Upon receiving a path the server concatenates this with the local paths that already exist, and forwards the result along its outgoing edges again. If there exists a cycle among $k$ servers, at least one of them will detect the cycle in at most $k$ rounds.
Global deadlock detection is difficult. Detection strategies:

- **Centralized detection**: Each site maintains its local wait-for graph. One distinguished site is selected to which all local wait-for graphs are send periodically. The selected site computes the union of all local wait-for graphs and checks for deadlocks.

- **Time-out based detection**: Whenever during a wait a *time-out* occurs, the respective transaction decides for a deadlock and aborts itself.

- **Edge chasing**: Whenever a transaction $T$ waits for a transaction $T'$, it sends its identification to $T'$. Whenever a transaction $T'$ receives such a message, it sends the identification of such $T$ to all transactions it is waiting for. If a transaction receives its own identification, it decides for a deadlock and it aborts itself.

- **Path pushing**:
  
  (i) Each server that has a waits-for path from transaction $t_i$ to transaction $t_j$ such that $T_i$ has an incoming waits-for-message edge and $T_j$ has an outgoing waits-for-message edge sends that path to the server along the outgoing edge.

  (ii) Upon receiving a path the server concatenates this with the local paths that already exist, and forwards the result along its outgoing edges again. If there exists a cycle among $k$ servers, at least one of them will detect the cycle in at most $k$ rounds.
Global deadlock detection is difficult. Detection strategies:

- **Centralized detection**: Each site maintains its local wait-for graph. One distinguished site is selected to which all local wait-for graphs are send periodically. The selected site computes the union of all local wait-for graphs and checks for deadlocks.

- **Time-out based detection**: Whenever during a wait a *time-out* occurs, the respective transaction decides for a deadlock and aborts itself.

- **Edge chasing**: Whenever a transaction \( T \) waits for a transaction \( T' \), it sends its identification to \( T' \). Whenever a transaction \( T' \) receives such a message, it sends the identification of such \( T \) to all transactions it is waiting for. If a transaction receives its own identification, it decides for a deadlock and it aborts itself.

- **Path pushing**:
  
  (i) Each server that has a waits-for path from transaction \( t_i \) to transaction \( t_j \) such that \( T_i \) has an incoming waits-for-message edge and \( T_j \) has an outgoing waits-for-message edge sends that path to the server along the outgoing edge.

  (ii) Upon receiving a path the server concatenats this with the local paths that already exist, and forwards the result along its outgoing edges again. If there exists a cycle among \( k \) servers, at least one of them will detect the cycle in at most \( k \) rounds.
Serializability by assigning timestamps to transactions

- Global and local transactions are timestamped; all subtransactions of a transaction obtain the same timestamp.
- Timestamps must be system-wide unique and based on synchronized clocks.
- To be system-wide unique, timestamps are values of local clocks concatenated with the site ID.

Time Stamp Protocol TS

- To each transaction $T$ it is assigned a unique timestamp $Z(T)$ when it is started.
- A transaction $T$ must not write an object which has been read by any $T'$ where $Z(T') > Z(T)$.
- A transaction $T$ must not write an object which has been written by any $T'$ where $Z(T') > Z(T)$.
- A transaction $T$ must not read an object which has been written by any $T'$ where $Z(T') > Z(T)$.
The TS-protocol guarantees serializability of schedules.

Let $S$ be a global schedule of a set of transactions $T = \{T_1, \ldots, T_n\}$, which all apply TS.

Assume, $S$ is not serializable, i.e. the conflict graph $G(S)$ is cyclic, where w.l.o.g. $T_1 \rightarrow T_2 \rightarrow \cdots \rightarrow T_k \rightarrow T_1$.

- Each edge $T \rightarrow T'$ implies $T$ and $T'$ have conflicting actions, where the action of $T$ preceds the one of $T'$.
- Because of TS we know $Z(T) < Z(T')$. This implies the following:

$$Z(T_1) < Z(T_2) < \ldots < Z(T_n) < Z(T_1),$$

a contradiction. Therefore $S$ is serializable.
The TS-protocol guarantees serializability of schedules.

Let $S$ be a global schedule of a set of transactions $T = \{T_1, \ldots, T_n\}$, which all apply TS.

Assume, $S$ is not serializable, i.e. the conflict graph $G(S)$ is cyclic, where w.l.o.g. $T_1 \rightarrow T_2 \rightarrow \cdots \rightarrow T_k \rightarrow T_1$.

- Each edge $T \rightarrow T'$ implies $T$ and $T'$ have conflicting actions, where the action of $T$ preceds the one of $T'$.

- Because of TS we know $Z(T) < Z(T')$. This implies the following:

\[ Z(T_1) < Z(T_2) < \ldots < Z(T_n) < Z(T_1), \]

a contradiction. Therefore $S$ is serializable.
The TS-protocol guarantees serializability of schedules.

Let $S$ be a global schedule of a set of transactions $T = \{T_1, \ldots, T_n\}$, which all apply TS.

Assume, $S$ is not serializable, i.e. the conflict graph $G(S)$ is cyclic, where w.l.o.g. $T_1 \rightarrow T_2 \rightarrow \cdots \rightarrow T_k \rightarrow T_1$.

- Each edge $T \rightarrow T'$ implies $T$ and $T'$ have conflicting actions, where the action of $T$ preceds the one of $T'$.
- Because of TS we know $Z(T) < Z(T')$. This implies the following:

\[ Z(T_1) < Z(T_2) < \ldots < Z(T_n) < Z(T_1), \]

a contradiction. Therefore $S$ is serializable.
Local and global transaction managers

- Each server runs its own *local* transaction manager which guarantees local serializability, i.e. the serializable execution of its local transactions and subtransactions.

- To guarantee global serializability a *global* transaction manager controls the execution of the global transactions. This could either be based on ordering the commit of the transaction, or by introducing artificial data objects called *tickets* which have to be accessed by the subtransactions.
Global serializability through local guarantees: rigorous local schedules

Rigorous schedules

A local schedule $S = (OP_S, <_S)$ of a set of complete transactions is *rigorous* if for all involved transactions (local and subtransactions) $T_i, T_j$ there holds:

Let $p_j \in OP_j, q_i \in OP_i, i \neq j$ such that $(p_j, q_i) \in conf(S)$. Then either $a_j <_S q_i$ or $c_j <_S q_i$.

Commit-deferred transaction

A global transaction $T$ is *commit-deferred* if its commit action is sent by the global transaction manager to the local sites of $T$ only *after* the local executions of all subtransactions of $T$ at that sites have been acknowledged.

Commit-deferment is achieved as a side-effect of the 2-phase-commit protocol.
Examples

Consider two servers where $D_1 = \{A, B\}$ and $D_2 = \{C, D\}$. We have the following transactions:

- **global:** $T_1 = WA WD$  
  $T_2 = WC WB$
- **local:** $T_3 = RA RB$  
  $T_4 = RC RD$

We have the following local schedules:

- $S_1 : \quad W_1A \quad c_1 \quad R_3A \quad R_3B \quad c_3 \quad W_2B \quad c_2$
- $S_2 : \quad W_2C \quad c_2 \quad R_4C \quad R_4D \quad c_4 \quad W_1D \quad c_1$

Even though the local schedules are serializable, the two global transactions are not executed in a serializable manner. The local schedules are rigorous, however not commit-deferred.
Lemma

A schedule is serializable, whenever it is rigorous.

Sketch of proof: Assume the contrary. Then there exists a history which has a cyclic conflict graph, though rigorousness holds. As a commit is the final action of a transaction, rigorousness makes such a cycle impossible.

Theorem

Let $S$ be a global history for local histories $S_1, \ldots, S_n$. If $S_i$ rigorous, $1 \leq i \leq n$ and all global transactions are commit-deferred, then $S$ is globally serializable.

Sketch of proof: Assume the contrary. Then there exists a history which has a cyclic conflict graph, though rigorousness and commit-deferment hold. As rigorousness guarantees local serializability, such a cycle must involve at least two sites. As a commit is the final action of a transaction, commit-deferment makes such a cycle impossible.

Because of the 2-phase-commit protocol, under rigorousness global serializability practically comes for free!
Global serializability through explicit measures: tickets

Ticket-based concurrency control

- Each server guarantees serializable local schedules in a way unknown for the global transactions.

- Each server maintains a special counter as database object, which is called ticket. Each subtransaction of a global transaction being executed at that server increments (reads and writes) the ticket (take-a-ticket-Operation). Doing so we introduce explicit conflicts between global transactions running at the same server.

- The global transaction manager guarantees that the order in which the tickets are accessed by the subtransactions will imply a linear order on the global transactions.
Global serializability through explicit measures: tickets

Ticket-based concurrency control

- Each server guarantees serializable local schedules in a way unknown for the global transactions.
- Each server maintains a special counter as database object, which is called ticket. Each subtransaction of a global transaction being executed at that server increments (reads and writes) the ticket (take-a-ticket-Operation). Doing so we introduce explicit conflicts between global transactions running at the same server.
- The global transaction manager guarantees that the order in which the tickets are accessed by the subtransactions will imply a linear order on the global transactions.
Global serializability through explicit measures: tickets

**Ticket-based concurrency control**

- Each server guarantees serializable local schedules in a way unknown for the global transactions.

- Each server maintains a special counter as database object, which is called *ticket*. Each subtransaction of a global transaction being executed at that server increments (reads and writes) the ticket (*take-a-ticket*-Operation). Doing so we introduce explicit conflicts between global transactions running at the same server.

- The global transaction manager guarantees that the order in which the tickets are accessed by the subtransactions will imply a linear order on the global transactions.
Applying ticketing by examples

By $I_j$ we denote the ticket at server $j$.

- Let $T_1 = R_1 A \ R_1 D$ and $T_2 = R_2 B \ R_2 C$ be global transactions and let $T_3 = R_3 A \ R_3 B \ W_3 A \ W_3 B$ and $T_4 = R_4 D \ W_4 D \ R_4 C \ W_4 C$ be local transactions.
  
  $S_1 : \ R_1(I_1) \ W_1(I_1) \ R_1 A \ R_3 A \ R_3 B \ W_3 A \ W_3 B \ R_2(I_1) \ W_2(I_1) \ R_2 B$

  $S_2 : \ R_4 D \ W_4 D \ R_1(I_2) \ W_1(I_2) \ R_1 D \ R_2(I_2) \ W_2(I_2) \ R_2 C \ R_4 C \ W_4 C$

  Not serializable - could be detected at server 2.

- Let $T_1 = R_1 A \ W_1 B$ and $T_2 = R_2 B \ W_2 A$ be global transactions.
  
  $S_1 : \ R_1(I_1) \ W_1(I_1) \ R_1 A \ R_2(I_1) \ W_2(I_1) \ W_2 A$

  $S_2 : \ R_2(I_2) \ W_2(I_2) \ R_1(I_2) \ W_1(I_2) \ W_1 B$

  Not serializable, could not be detected neither at server 1 nor at server 2, however the order of take-a-ticket operations does not imply a linear order on the global transactions.
Applying ticketing by examples

By \( I_j \) we denote the ticket at server \( j \).

- Let \( T_1 = R_1 A R_1 D \) and \( T_2 = R_2 B R_2 C \) be global transactions and let
  \( T_3 = R_3 A R_3 B W_3 A W_3 B \) and \( T_4 = R_4 D W_4 D R_4 C W_4 C \) be local transactions.
  
  \( S_1 : \ R_1(I_1) \ W_1(I_1) \ R_1 A R_3 A R_3 B W_3 A W_3 B R_2(I_1) \ W_2(I_1) \ R_2 B \)
  
  \( S_2 : \ R_4 D W_4 D R_1(I_2) \ W_1(I_2) \ R_1 D R_2(I_2) \ W_2(I_2) \ R_2 C R_4 C W_4 C \)

Not serializable - could be detected at server 2.

- Let \( T_1 = R_1 A W_1 B \) and \( T_2 = R_2 B W_2 A \) be global transactions.
  
  \( S_1 : \ R_1(I_1) \ W_1(I_1) \ R_1 A R_2(I_1) \ W_2(I_1) \ W_2 A \)
  
  \( S_2 : \ R_2(I_2) \ W_2(I_2) \ R_2 B R_1(I_2) \ W_1(I_2) \ W_1 B \)

Not serializable, could not be detected neither at server 1 nor at server 2, however the order of take-a-ticket operations does not imply a linear order on the global transactions.