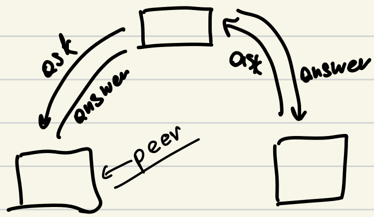
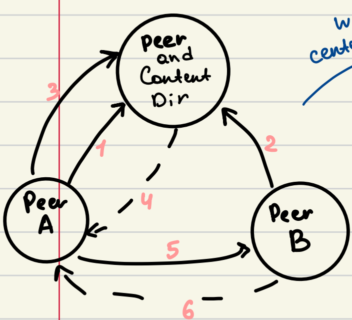
def P2P - architectural paradigm where control\data is shared among a set of equal peers!

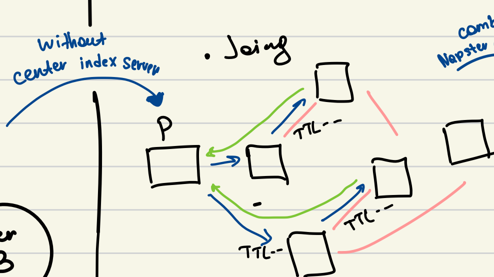


First generation of P2P

Napster	Gnutella	KaZaA



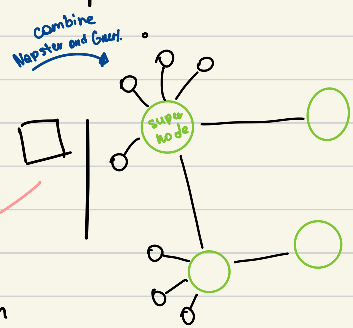
+ P2P
- Doesn't scale
+ Fast
- Single point of failure



- flooding algorithm
- TTL (time_to_live)

- High latency
- No garantee for search
- freeloaders

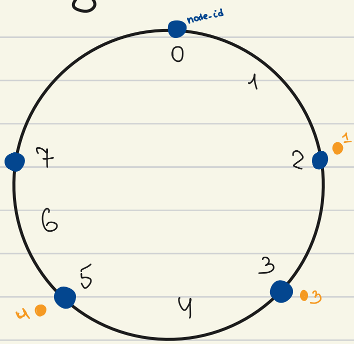
??? all peers are equals



(2.) DHT

1. P-Grid $O(\log n) \to$ prefix
2. CAN $O(d \cdot n^{\frac{1}{d}})$
3. CHORD $O(\log n)$ SHA-1

Ring modulo $2^m \cdot [0, 2^m]$



● - Peer node-id = hash (IP-address)

● - Data key = hash(data)

- log(n)
- Each node maintains a finger table

i	$n + 2^{i-1}$	node-id

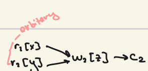
- Joining

  Stabilize ( )

- Recover from failure

(3) Databases

1. Transaction

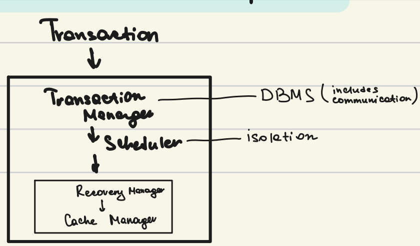


A *transaction* T_i is a partial order with ordering relation $<_i$ where

1. $T_i \subseteq \{r_i[x], w_i[x] \mid x \text{ is a data item}\} \cup \{a_i, c_i\}$;
2. $a_i \in T_i$ iff $c_i \notin T_i$;
3. if t is c_i or a_i (whichever is in T_i), for any other operation $p \in T_i$, $p <_i t$; and
4. if $r_i[x], w_i[x] \in T_i$, then either $r_i[x] <_i w_i[x]$ or $w_i[x] <_i r_i[x]$.

2. Transaction properties: ACID

3. DB System Model and Components



4. Serializability Theory (Math. tool to prove whether or not S works correctly)

- Correctness criteria for the correct execution of concurrent transactions.
- $J = \{T^1, T^2 \dots T^n\}$
- **Basic assumption**:

  A1. Each T^i correct, when execute in isolation

  A2: Each serial (sequential) execution of all n transact. (in any order) correct

need criteria, allows check whether a given parallel execut. is eq. to serial.

Scheduler (Actions: Execute | reject | delay) (• a program or collection of program that controls the concurrent execution of transaction)

- **History** (indicates the order in which the oper. of the trans. were executed relatively to each other). Complet history:

Formally, let $T = \{T_1, T_2, \dots, T_n\}$ be a set of transactions. A *complete history* H over T is a partial order with ordering relation $<_H$ where:

1. $H = \cup_{i=1}^{n} T_i$;
2. $<_H \supseteq \cup_{i=1}^{n} <_i$; and
3. for any two conflicting operations $p, q \in H$, either $p <_H q$ or $q <_H p$.

so history is prefix of complet history.

- **Committed projection** $C(H)$ → delete all operations that do not belong to transactions commited in H.

1. Serializable histories

- **Equivalent histories:** (conflict equivalence)

We define two histories H and H' to be *equivalent* ($\equiv$) if

1. they are defined over the same set of transactions and have the same operations; and
2. they order conflicting operations of nonaborted transactions in the same way; that is, for any conflicting operations p_i and q_j belonging to transactions T_i and T_j (respectively) where $a_i, a_j \notin H$, if $p_i <_H q_j$ then $p_i <_{H'} q_j$.[6]

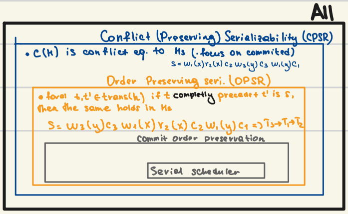
Idea: outcome of a concurrent execution of Ts depends only on the relative ordering of conflicting operations.

- Serial history (1,2,3,4)
- **A H is serializable if C(H) is equivalent to Hs.**

- SG (dag)

  A H is serializable iff SG(H) is acyclic.

- Correctness Criteria



5. Atomicity of Distributed Transactions

1. Failure Classes

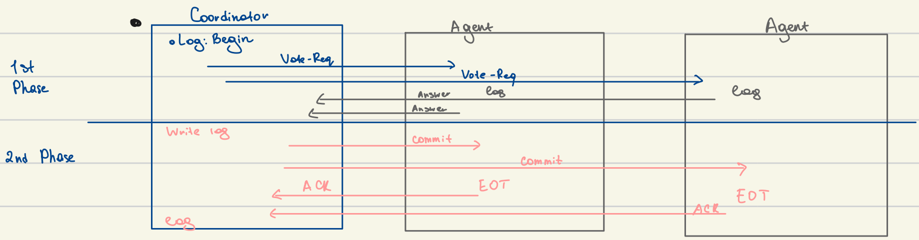
a) Failure in the application or violation of constraints → Abort of single transaction

2. – 5 → Crash Recovery (Redo / Undo)

→ Disaster Recovery.

2. 2PC (2 Phase Commit Protocol)

- All-or-nothing commited Protocol
- Safety (If one commits, no one aborts)
  - if one aborts, no one commits)

  Liveness (If no failures and A and B can commit, action commits
  - if failures, reach a conclusion ASAP)
- 

- Locks all sub-transactions since coord. might roll-back

Variants:

1. Presumed Abort

- if nothing known about a transaction, assume ABORT
- Recovery coord: no logs? abort
- Agent drop during 1st phase: write log → abort (no notification)

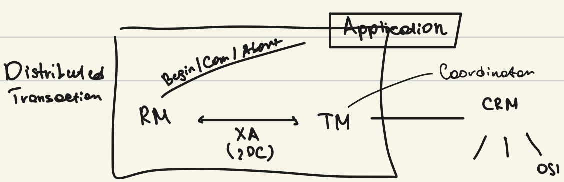
2. Transfer of Coordination

- Tranfer 3 times

3. Tree 2PC (Nested 2PC)

3. 3PC

4. Distributed Transaction: X / Open DTP



5. Data Consistency (Serializability)

- MDBMS (all T are global, m cooperating and homogeneous DS)
- Global T^i consist of a set of sub-transaction t^i_k, one per component DBMS
- Schedulers:

a) Aggressive (schedule immediatly)
b) Conservative (delay operations)

1. 2 Phase locking (2PL)

1. When it receives an operation $p_i[x]$ from the TM, the scheduler tests if $pl_i[x]$ conflicts with some $ql_j[x]$ that is already set. If so, it delays $p_i[x]$, forcing T_i to wait until it can set the lock it needs. If not, then the scheduler sets $pl_i[x]$, and then sends $p_i[x]$ to the DM.[2] → no two locks at the same time
2. Once the scheduler has set a lock for T_i, say $pl_i[x]$, it may not release that lock *at least* until after the DM acknowledges that it has processed the lock's corresponding operation, $p_i[x]$. → DM process in the same order
3. Once the scheduler has released a lock for a transaction, it may not subsequently obtain *any* more locks for that transaction (on *any* data item). → all pairs of conflict operations of 2 transactions are scheduled in the same order.

T_1: $r_1[x] \rightarrow w_1[y] \rightarrow c_1$ $\quad$ T_2: $w_2[x] \rightarrow w_2[y] \rightarrow c_2$

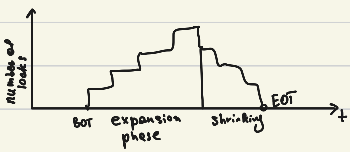
and suppose they execute as follows:

$$H_1 = rl_1[x]\ r_1[x]\ ru_1[x]\ wl_2[x]\ w_2[x]\ wl_2[y]\ w_2[y]\ wu_2[x]\ wu_2[y]\ c_2\ wl_1[y]\ w_1[y]\ wu_1[y]\ c_1$$

Since $r_1[x] < w_2[x]$ and $w_2[y] < w_1[y]$, SG(H_1) consists of the cycle $T_1 \rightarrow T_2 \rightarrow T_1$. Thus, H_1 is not SR.

The problem in H_1 is that T_1 released a lock ($ru_1[x]$) and subsequently set a lock ($wl_1[y]$), in violation of the two phase rule. Between $ru_1[x]$ and $wl_1[y]$, another transaction T_2 wrote into both x and y, thereby appearing to follow T_1 with respect to x and precede it with respect to y. Had T_1 obeyed the two phase rule, this "window" between $ru_1[x]$ and $wl_1[y]$ would not have opened, and T_2 could not have executed as it did in H_1. For example, T_1 and T_2 might have

- A 2PL is CPSR, the resulting schedules are CPSR



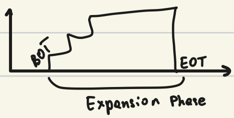
2. Dead locks

1. Timeout (Think smth wrong → abort)
2. Waits-for graph (WFG) → (Check cyclics)

3. Variants of 2PL

1. Conservative 2PL (Never aborts transaction)
   - Waiting queue

2. Strict Two-Phase Locking (S_2PL) Protocol

- All locks have to be kept until commit time.
- 

- COPSR

4. Distributed Dead locks

1. Timestamp-based Deadlock Prevention

$P(T^i) = \frac{1}{ts(T^i)}$ – old transact. → higher priority.

Strategies:

a) Wait-die

- wait only if P is higher
- if not → abort

b) Wound-Wait

- younger waits
- Older get, young aborted

2. Ordered Shared Locks (OSL)

- + additional lock mode (OS Ordered Shared) parallelism is possible, but constrained
- Rules: 1. All transactions respect the two-phase nature of 2PL

2. A dependent transaction does not release any locks. (if T^k wants to commit, all T^i on which T^k depends must be committed.

- P_8 is OPSR, P_{8+} S_2PL is COPSR (all possible COPSR schedules)

5. Non-Locking Schedulers

- MDBMS (Local S_2PL, Globally – nothing)
- Ticket system

1. Timestamp Ordering (TO)

2. Strict Timestamp Ordering (STO)

- Read only from commited T

4 Replicas

- Where:
    - Primary Copy
    - Everywhere

  and

  When
    - Eager Replication
    - Lazy Replication

- Quorum

- Regarding the location *where* updates can be executed, a distinction is made between
  – Primary Copy: one dedicated replica node in the system hosts a „leading“ copy. All updates have to be applied at this copy and are then propagated to the other replicas
  – Update Everywhere: updates can be applied at each replica in the system
- Regarding *when* updates are propagated, the following approaches exist:
  – Eager Replication: the update operation and all subsequent update propagation is done within the same transaction
  – Lazy Replication: the original update operation and all updates that happen at the other replicas are logically decoupled, i.e., they are executed in different transactions