Distributed Systems
Principles and Paradigms

Chapter 05

(version 12th October 2001)

Maarten van Steen
Vrije Universiteit Amsterdam, Faculty of Science
Dept. Mathematics and Computer Science
Room R4.20. Tel: (020) 444 7784
E-mail: steen@cs.vu.nl, URL: www.cs.vu.nl/∼steen

Clock Synchronization

- Physical clocks
- Logical clocks
- Vector clocks

Physical Clocks (1/3)

Problem: Sometimes we simply need the exact time, not just an ordering.

Solution: Universal Coordinated Time (UTC):

- Based on the number of transitions per second of the cesium 133 atom (pretty accurate).
- At present, the real time is taken as the average of some 50 cesium-clocks around the world.
- Introduces a leap second from time to time to compensate that days are getting longer.

UTC is **broadcast** through short wave radio and satellite. Satellites can give an accuracy of about ± 0.5 ms.

Question: Does this solve all our problems? Don't we now have some global timing mechanism?

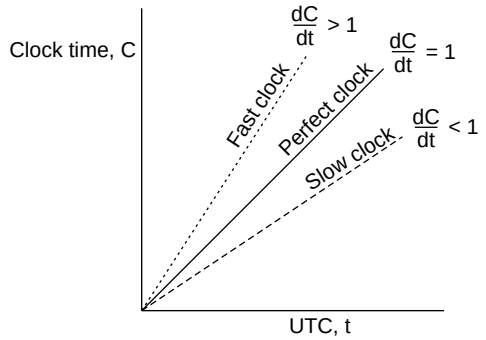
Physical Clocks (2/3)

Problem: Suppose we have a distributed system with a UTC-receiver somewhere in it $\Rightarrow$ we still have to distribute its time to each machine.

Basic principle:

- Every machine has a timer that generates an interrupt H times per second.
- There is a clock in machine p that **ticks** on each timer interrupt. Denote the value of that clock by $C_p(t)$, where t is UTC time.
- Ideally, we have that for each machine p, $C_p(t) = t$, or, in other words, $dC/dt = 1$

Physical Clocks (3/3)



In practice: $1 - \rho \leq \frac{dC}{dt} \leq 1 + \rho$.

Goal: Never let two clocks in any system differ by more than δ time units $\Rightarrow$ synchronize at least every $\delta/(2\rho)$ seconds.

Clock Synchronization Principles

Principle I: Every machine asks a **time server** for the accurate time at least once every $\delta/(2\rho)$ seconds.

Okay, but you need an accurate measure of round trip delay, including interrupt handling and processing incoming messages.

Principle II: Let the time server scan all machines periodically, calculate an average, and inform each machine how it should adjust its time relative to its present time.

Okay, you'll probably get every machine in sync. Note: you don't even need to propagate UTC time (why not?)

Fundamental problem: You'll have to take into account that setting the time back is **never** allowed $\Rightarrow$ smooth adjustments.

The Happened-Before Relationship

Problem: We first need to introduce a notion of ordering before we can order anything.

The **happened-before** relation on the set of events in a distributed system is the smallest relation satisfying:

- If a and b are two events in the same process, and a comes before b, then $a \rightarrow b$.
- If a is the sending of a message, and b is the receipt of that message, then $a \rightarrow b$.
- If $a \rightarrow b$ and $b \rightarrow c$, then $a \rightarrow c$.

Note: this introduces a partial ordering of events in a system with concurrently operating processes.

Logical Clocks (1/2)

Problem: How do we maintain a global view on the system's behavior that is consistent with the happened-before relation?

Solution: attach a timestamp $C(e)$ to each event e, satisfying the following properties:

P1: If a and b are two events in the same process, and $a \rightarrow b$, then we demand that $C(a) < C(b)$.

P2: If a corresponds to sending a message m, and b to the receipt of that message, then also $C(a) < C(b)$.

Problem: How to attach a timestamp to an event when there's no global clock $\Rightarrow$ maintain a **consistent** set of logical clocks, one per process.

Logical Clocks (2/2)

Each process P_i maintains a **local** counter C_i and adjusts this counter according to the following rules:

1: For any two successive events that take place within P_i, C_i is incremented by 1.

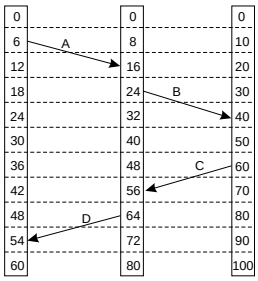
2: Each time a message m is sent by process P_i, the message receives a timestamp $T_m = C_i$.

3: Whenever a message m is received by a process P_j, P_j adjusts its local counter C_j:

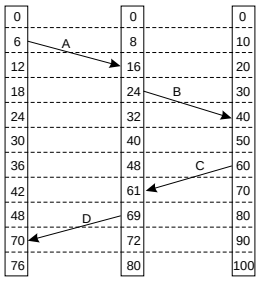
$$C_j \leftarrow \max\{C_j + 1, T_m + 1\}.$$

Property **P1** is satisfied by (1); Property **P2** by (2) and (3).

Logical Clocks – Example



(a)



(b)

Total Ordering with Logical Clocks

Problem: it can still occur that two events happen at the same time. Avoid this by attaching a process number to an event:

P_i timestamps event e with $C_i(e).i$

Then: $C_i(a).i$ before $C_j(b).j$ if and only if:

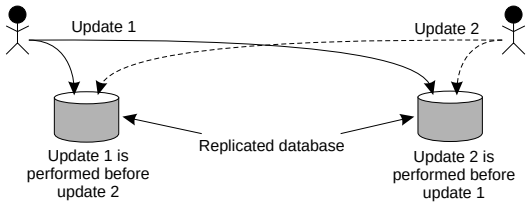
1: $C_i(a) < C_j(a)$; or

2: $C_i(a) = C_j(b)$ and $i < j$

Example: Totally-Ordered Multicast (1/2)

Problem: We sometimes need to guarantee that concurrent updates on a replicated database are seen in the same order everywhere:

- Process P_1 adds \$100 to an account (initial value: \$1000)
- Process P_2 increments account by 1%
- There are two replicas



Outcome: in absence of proper synchronization, replica #1 will end up with \$1111, while replica #2 ends up with \$1110.

Example: Totally-Ordered Multicast (2/2)

- Process P_i sends timestamped message msg_i to all others. The message itself is put in a local queue $queue_i$.
- Any incoming message at P_j is queued in $queue_j$, according to its timestamp.
- P_j passes a message msg_i to its application if:

  (1) msg_i is at the head of $queue_j$
  (2) for each process P_k, there is a message msg_k in $queue_j$ with a larger timestamp.

  Note: We are assuming that communication is reliable and FIFO ordered.

Extension to Multicasting: Vector Timestamps (1/2)

Observation: Lamport timestamps do not guarantee that if $C(a) < C(b)$ that a indeed happened before b. We need **vector timestamps** for that.

- Each process P_i has an array $V_i[1..n]$, where $V_i[j]$ denotes the number of events that process P_i knows have taken place at process P_j.
- When P_i sends a message m, it adds 1 to $V_i[i]$, and sends V_i along with m as **vector timestamp** $vt(m)$. Result: upon arrival, each other process knows P_i's timestamp.

Question: What does $V_i[j] = k$ mean in terms of messages sent and received?

Extension to Multicasting: Vector Timestamps (2/2)

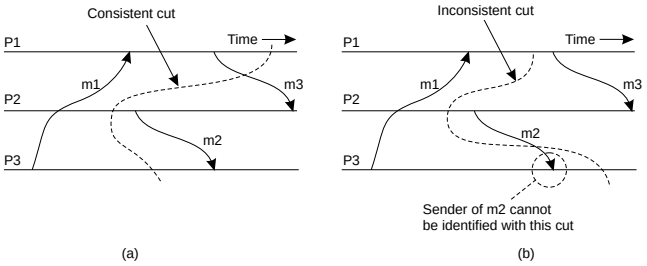
- When a process P_j receives a message m from P_i with vector timestamp $vt(m)$, it (1) updates each $V_j[k]$ to $\max\{V_j[k], V(m)[k]\}$, and (2) increments $V_j[j]$ by 1. **NOTE:** Book is wrong!
- To support causal delivery of messages, assume you increment your own component only when sending a message. Then, P_j postpones delivery of m until:
  - $vt(m)[i] = V_j[i] + 1$.
  - $vt(m)[k] \leq V_j[k]$ for $k \neq i$.

Example: Take $V_3 = [0,2,2]$, $vt(m) = [1,3,0]$. What information does P_3 have, and what will it do when receiving m (from P_1)?

Global State (1/3)

Basic Idea: Sometimes you want to collect the current state of a distributed computation, called a **distributed snapshot**. It consists of all local states and messages in transit.

Important: A distributed snapshot should reflect a **consistent** state:

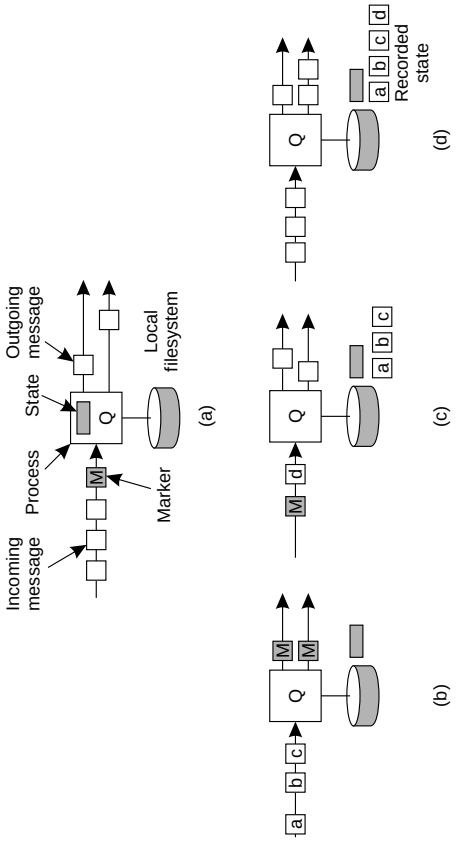


Global State (2/3)

Note: any process P can initiate taking a distributed snapshot

- P starts by recording its own local state
- P subsequently sends a marker along each of its outgoing channels
- When Q receives a marker through channel C, its action depends on whether it had already recorded its local state:
  - Not yet recorded: it records its local state, and sends the marker along each of its outgoing channels
  - Already recorded: the marker on C indicates that the channel's state should be recorded: all messages received before this marker and the time Q recorded its own state.
- Q is finished when it has received a marker along each of its incoming channels

Global State (3/3)



Election Algorithms

Principle: An algorithm requires that some process acts as a coordinator. The question is how to select this special process **dynamically**.

Note: In many systems the coordinator is chosen by hand (e.g. file servers). This leads to centralized solutions $\Rightarrow$ single point of failure.

Question: If a coordinator is chosen dynamically, to what extent can we speak about a centralized or distributed solution?

Question: Is a fully distributed solution, i.e. one without a coordinator, always more robust than any centralized/coordinated solution?

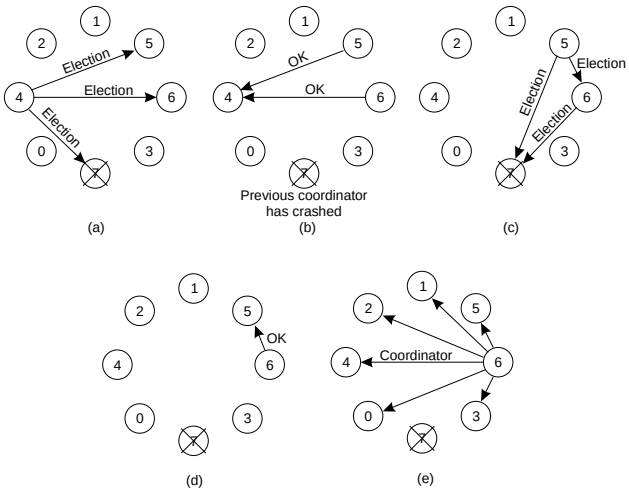
Election by Bullying (1/2)

Principle: Each process has an associated priority (weight). The process with the highest priority should always be elected as the coordinator.

Issue: How do we find the heaviest process?

- Any process can just start an election by sending an election message to all other processes (assuming you don't know the weights of the others).
- If a process P_{heavy} receives an election message from a lighter process P_{light}, it sends a take-over message to P_{light}. P_{light} is out of the race.
- If a process doesn't get a take-over message back, it wins, and sends a victory message to all other processes.

Election by Bullying (2/2)



Question: We're assuming something very important here – what?

Election in a Ring

Principle: Process priority is obtained by organizing processes into a (logical) ring. Process with the highest priority should be elected as coordinator.

- Any process can start an election by sending an election message to its successor. If a successor is down, the message is passed on to the next successor.
- If a message is passed on, the sender adds itself to the list. When it gets back to the initiator, everyone had a chance to make its presence known.
- The initiator sends a coordinator message around the ring containing a list of all living processes. The one with the highest priority is elected as coordinator.

Question: Does it matter if two processes initiate an election?

Question: What happens if a process crashes *during* the election?

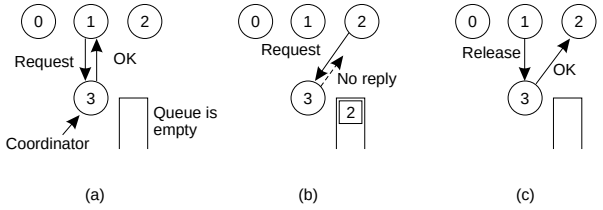
Mutual Exclusion

Problem: A number of processes in a distributed system want exclusive access to some resource.

Basic solutions:

- Via a centralized server.
- Completely distributed, with no topology imposed.
- Completely distributed, making use of a (logical) ring.

Centralized: Really simple:

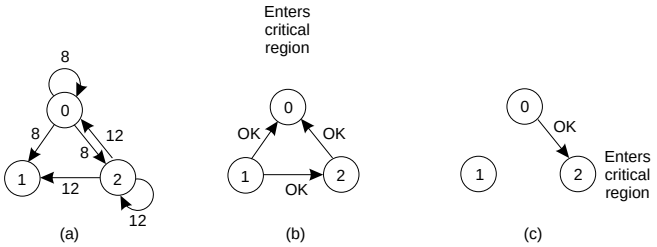


Mutual Exclusion: Ricart & Agrawala

Principle: The same as Lamport except that acknowledgments aren't sent. Instead, replies (i.e. grants) are sent only when:

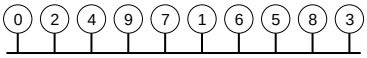
- The receiving process has no interest in the shared resource; or
- The receiving process is waiting for the resource, but has lower priority (known through comparison of timestamps).

In all other cases, reply is **deferred**, implying some more local administration.

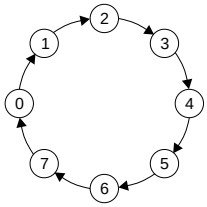


Mutual Exclusion: Token Ring Algorithm

Essence: Organize processes in a *logical* ring, and let a token be passed between them. The one that holds the token is allowed to enter the critical region (if it wants to)



(a)



(b)

Comparison:

Algorithm	# msgs	Delay	Problems
Centralized	3	2	Coordinator crash
Distributed	2 (n – 1)	2 (n – 1)	Crash of any process
Token ring	1 to ∞	0 to n – 1	Lost token, process crash

Distributed Transactions

- The transaction model
- Classification of transactions
- Concurrency control

Transactions

```
BEGIN_TRANSACTION(server, transaction);
READ(transaction, file-1, data);
WRITE(transaction, file-2, data);
newData := MODIFIED(data);
IF WRONG(newData) THEN
  ABORT_TRANSACTION(transaction);
ELSE
  WRITE(transaction, file-2, newData);
  END_TRANSACTION(transaction);
END IF;
```

Essential: All READ and WRITE operations are executed, i.e. their effects are made permanent at the execution of END_TRANSACTION.

Observation: Transactions form an **atomic** operation.

ACID Properties

Model: A transaction is a collection of operations on the state of an object (database, object composition, etc.) that satisfies the following properties:

Atomicity: All operations either succeed, or all of them fail. When the transaction fails, the state of the object will remain unaffected by the transaction.

Consistency: A transaction establishes a valid state transition. This does not exclude the possibility of invalid, intermediate states during the transaction's execution.

Isolation: Concurrent transactions do not interfere with each other. It appears to each transaction T that other transactions occur either *before* T, or *after* T, but never both.

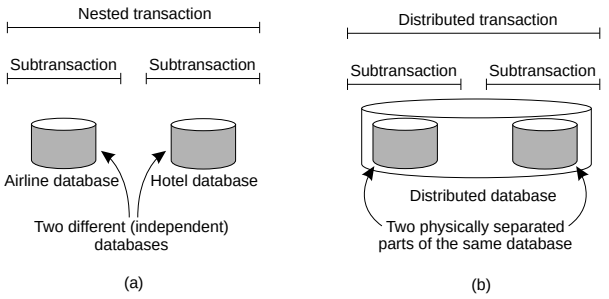
Durability: After the execution of a transaction, its effects are made permanent: changes to the state survive failures.

Transaction Classification

Flat transactions: The most familiar one: a sequence of operations that satisfies the ACID properties.

Nested transactions: A *hierarchy* of transactions that allows (1) concurrent processing of subtransactions, and (2) recovery per subtransaction.

Distributed transactions: A (flat) transaction that is executed on distributed data ⇒ often implemented as a two-level nested transaction with one subtransaction per node.



Flat Transactions: Limitations

Problem: Flat transactions constitute a very simple and clean model for dealing with a sequence of operations that satisfies the ACID properties. However, after a series of successful operations *all* changes should be undone in the case of failure. Sometimes unnecessary:

Trip planning. Plan a intercontinental trip where all flights have been reserved, but filling in the last part requires some "experimentation." The first reservations are *known* to be in order, but cannot yet be committed.

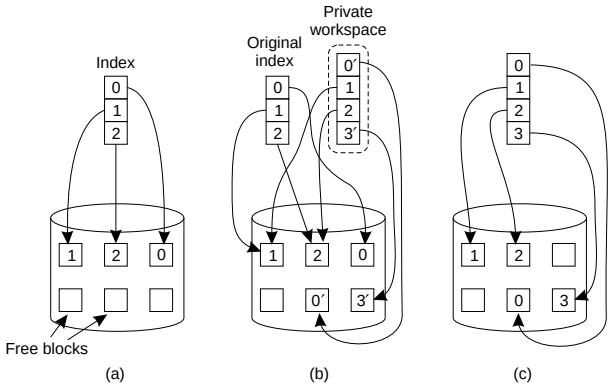
Bulk updates. When updating bank accounts for monthly interests we have to lock the entire database (every account should be updated exactly once: it is a transaction over the entire database.)

Better: each update is immediately committed. However, in the case of failure, we'll have to be able to continue where we left off.

Implementing Transactions (1/2)

Solution 1: Use a **private workspace**, by which the client gets its own copy of the (part of the) database. When things go wrong delete copy, otherwise commit the changes to the original.

Optimization: don't get everything:



Implementing Transactions (2/2)

Solution 2: Use a writeahead log in which changes are recorded allowing you to **roll back** when things go wrong:

```
x = 0;
y = 0;
BEGIN_TRANSACTION;
    x = x + 1;
    y = y + 2;
    x = y * y;
END_TRANSACTION;
```

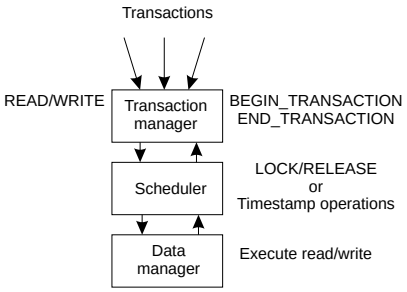
Log	Log	Log
[x = 0/1]	[x = 0/1]	[x = 0/1]
	[y = 0/2]	[y = 0/2]
		[x = 1/4]

Question: Where do **distributed** transactions fit in?

Transactions: Concurrency Control (1/2)

Problem: Increase efficiency by allowing several transactions to execute at the same time.

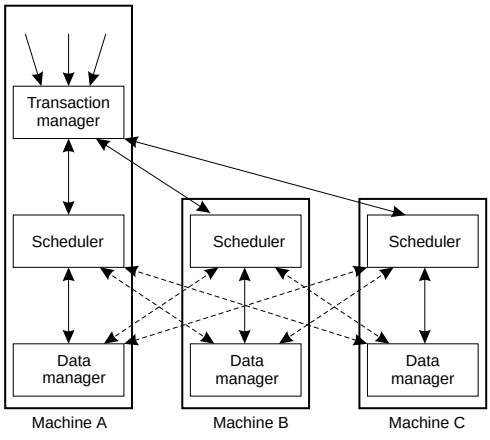
Constraint: Effect should be the same as if the transactions were executed in some serial order.



Question: Does it actually make sense to allow concurrent transactions on a single server?

Transactions: Concurrency Control (2/2)

Distributed transactions:



Question: What about a distributed transaction manager?

Serializability (1/2)

Consider a collection E of transactions $T_1, \ldots, T_n$. Goal is to conduct a **serializable execution** of E:

- Transactions in E are possibly concurrently executed according to some schedule S.
- Schedule S is equivalent to some *totally ordered* execution of $T_1, \ldots, T_n$.

```
BEGIN_TRANS     BEGIN_TRANS     BEGIN_TRANS
  x = 0;          x = 0;          x = 0;
  x = x + 1;      x = x + 2;      x = x + 3;
END_TRANS       END_TRANS       END_TRANS
```

Schedule 1	Schedule 2	Schedule 3
x = 0	x = 0	x = 0
x = x + 1	x = 0	x = 0
x = 0	x = x + 1	x = x + 1
x = x + 2	x = x + 2	x = 0
x = 0	x = 0	x = x + 2
x = x + 3	x = x + 3	x = x + 3
OK	OK	Illegal

Serializability (2/2)

Note: Because we're not concerned with the computations of each transaction, a transaction can be modeled as a *log* of **read** and **write** operations.

Two operations $\mathsf{OPER}(T_i, x)$ and $\mathsf{OPER}(T_j, x)$ on the same data item x, and from a set of logs may **conflict** at a data manager:

read-write conflict (rw): One is a read operation while the other is a write operation on x.

write-write conflict (ww): Both are write operations on x.

Basic Scheduling Theorem

Let $\mathbf{T} = \{T_1, \ldots, T_n\}$ be a set of transactions and let E be an execution of these transactions modeled by logs $\{L_1, \ldots, L_n\}$. E is serializable if there exists a total ordering of $\mathbf{T}$ such that for each pair of conflicting operations O_i and O_j from distinct transactions T_i and T_j (respectively), O_i precedes O_j in any log $L_1, \ldots, L_n$, if and only if T_i precedes T_j in the total ordering.

Note: The important thing is that we process conflicting reads and writes in certain relative orders. This is what concurrency control is all about.

Note: It turns out that read-write and write-write conflicts can be synchronized *independently*, as long as we stick to a total ordering of transactions that is consistent with both types of conflicts.

Synchronization Techniques

Two-phase locking: Before reading or writing a data item, a lock must be obtained. After a lock is given up, the transaction is not allowed to acquire any more locks.

Timestamp ordering: Operations in a transaction are timestamped, and data managers are forced to handle operations in timestamp order.

Optimistic control: Don't prevent things from going wrong, but correct the situation if conflicts actually did happen. Basic assumption: you can pull it off in most cases.

Two-phase Locking (1/3)

- Clients do only READ and WRITE operations within transactions.
- Locks are **granted** and **released** only by scheduler.
- Locking policy is to avoid **conflicts** between operations

Two-phase Locking (2/3)

- **Rule 1:** When client submits OPER(T_i,x), scheduler tests whether it conflicts with an operation OPER(T_j,x) from some other client. If no conflict then grant LOCK(T_i,x), otherwise delay execution of OPER(T_i,x).

  Conflicting operations are executed in the same order as that locks are granted.

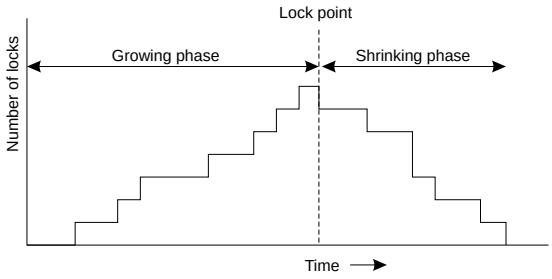
- **Rule 2:** If LOCK(T_i,x) has been granted, do not release the lock until OPER(T_i,x) has been executed by data manager.

  Guarantees LOCK $\rightarrow$ OPER $\rightarrow$ RELEASE order.

- **Rule 3:** If RELEASE(T_i,x) has taken place, no more locks for T_i may be granted.

  Combined with rule 1, guarantees that all pairs of conflicting operations of two transactions are done in the same order. ***Huh?***

Two-phase Locking (3/3)



Centralized 2PL: A single site handles all locks

Primary 2PL: Each data item is assigned a primary site to handle its locks. Data is not necessarily replicated

Distributed 2PL: Assumes data can be replicated. Each primary is responsible for handling locks for its data, which may reside at remote data managers.

Two-phase Locking: Problems

Problem 1: System can come into a **deadlock**. *How?* Practical solution: put a timeout on locks and abort transaction on expiration.

Problem 2: When should the scheduler actually release a lock:

(1) when operation has been executed
(2) when it knows that no more locks will be requested

No good way of testing condition (2) unless transaction has been committed or aborted.

Moreover: Assume the following execution sequence takes place:
$\mathsf{RELEASE}(T_i,\mathsf{x}) \rightarrow \mathsf{LOCK}(T_j,\mathsf{x}) \rightarrow \mathsf{ABORT}(T_i)$.

Consequence: scheduler will have to abort T_j as well (**cascaded aborts**).

Solution: Release *all* locks only at commit/abort time (**strict two-phase locking**).

Timestamp Ordering (1/2)

Basic idea:

- Transaction manager assigns a unique timestamp $\mathsf{TS}(T_i)$ to each transaction T_i.
- Each operation $\mathsf{OPER}(T_i,\mathsf{x})$ submitted by the transaction manager to the scheduler is timestamped $\mathsf{TS}(\mathsf{OPER}(T_i,\mathsf{x})) \leftarrow \mathsf{TS}(T_i)$.

Scheduler adheres to following rule:

If	$\mathsf{OPER}(T_i,\mathsf{x})$ and $\mathsf{OPER}(T_j,\mathsf{x})$ conflict
then	data manager processes $\mathsf{OPER}(T_i,\mathsf{x})$ before $\mathsf{OPER}(T_j,\mathsf{x})$
iff	$\mathsf{TS}(\mathsf{OPER}(T_i,\mathsf{x})) < \mathsf{TS}(\mathsf{OPER}(T_j,\mathsf{x}))$

Note: rather aggressive for if a single $\mathsf{OPER}(T_i,\mathsf{x})$ is rejected, T_i will have to be aborted.

Timestamp Ordering (2/2)

- **Suppose:** $\mathsf{TS}(\mathsf{OPER}(T_i,\mathsf{x})) < \mathsf{TS}(\mathsf{OPER}(T_j,\mathsf{x}))$, but that $\mathsf{OPER}(T_j,\mathsf{x})$ has already been processed by the data manager.
- **Then:** the scheduler should reject $\mathsf{OPER}(T_i,\mathsf{x})$, as it came in *too late*.

- **Suppose:** $\mathsf{TS}(\mathsf{OPER}(T_i,\mathsf{x})) < \mathsf{TS}(\mathsf{OPER}(T_j,\mathsf{x}))$, and that $\mathsf{OPER}(T_i,\mathsf{x})$ has already been processed by the data manager.
- **Then:** the scheduler would submit $\mathsf{OPER}(T_j,\mathsf{x})$ to data manager.
- **Refinement:** hold back $\mathsf{OPER}(T_j,\mathsf{x})$ until T_i commits or aborts.

Question: Why would we do this?

Optimistic Concurrency Control

Observation: (1) Maintaining locks costs a lot; (2) in practice not many conflicts.

Alternative: Go ahead immediately with all operations, use tentative writes everywhere (shadow copies), and solve conflicts later on.

Phases: allow operations tentatively $\rightarrow$ validate effects $\rightarrow$ make updates permanent.

Validation: Check two basic rules for each pair of active transactions T_i and T_j:

- **Rule 1:** T_i must not read or write data that has been written by T_j.
- **Rule 2:** T_j must not read or write data that has been written by T_i.

If one of the rules doesn't hold: **abort** transaction.