# 9 Replication in Distributed Databases

9.1 Replication and its issues
9.2 Consistency models
9.3 Classifcation of replication strategies
9.4 Update propagation
lazy / eager – primary copy / multimaster
9.5 Failure and recovery
9.6 Multimaster replication: version vectors
9.7 Replication in the real world
9.8 Replication and Consistency @ Google -
a Paxos application
9.9 Mobile Databases

x based on Bernstein / Newcomer, Kifer / A. Bernstein, Garcia-Molina , Oracle

# 9.1 Replication: motivation, issues

Freie Universität Berlin

***What*: copies of the **same data** at **more than one site – replica** [1]**

***Why*: Improve performance**

- Decrease ***response time***
  - Local reads are fast
  - Lower disk contention: less read on a single data set
- Increase ***throughput***
- Increase ***availability*** ("fail-stop nodes") ←
  - Site failure should not impact availability of data
- Archive Backup

In practice: Throughput, availability may become worse… Why??

[1] D: *Kopie, Replikat*

# Why availability? The Joys of real Hardware

Freie Universität Berlin

Typical first year for a new cluster (Google experience):

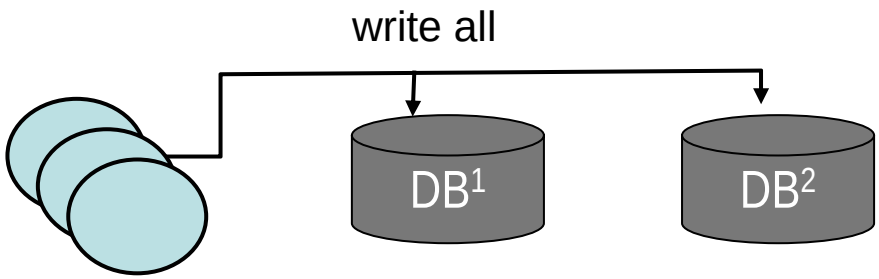
~0.5 **overheating** (power down most machines in <5 mins, ~1-2 days to recover)
~1 **PDU failure** (~500-1000 machines suddenly disappear, ~6 hours to come back)
~1 **rack-move** (plenty of warning, ~500-1000 machines powered down, ~6 hours)
~1 **network rewiring** (rolling ~5% of machines down over 2-day span)
~20 **rack failures** (40-80 machines instantly disappear, 1-6 hours to get back)
~5 **racks go wonky** (40-80 machines see 50% packet loss)
~8 **network maintenances** (4 might cause ~30-minute random connectivity losses)
~12 **router reloads** (takes out DNS and external vips for a couple minutes)
~3 **router failures** (have to immediately pull traffic for an hour)
~dozens of minor **30-second blips** for dns
~**1000 individual machine failures**
~thousands of **hard drive failures**
slow disks, bad memory, misconfigured machines, flaky machines, etc.

**Availability IS a challenge**, in particular with writes.

**Data by Jeff Dean, Google, http://labs.google.com/people/jeff**

# Spectrum of scenarios, solutions..

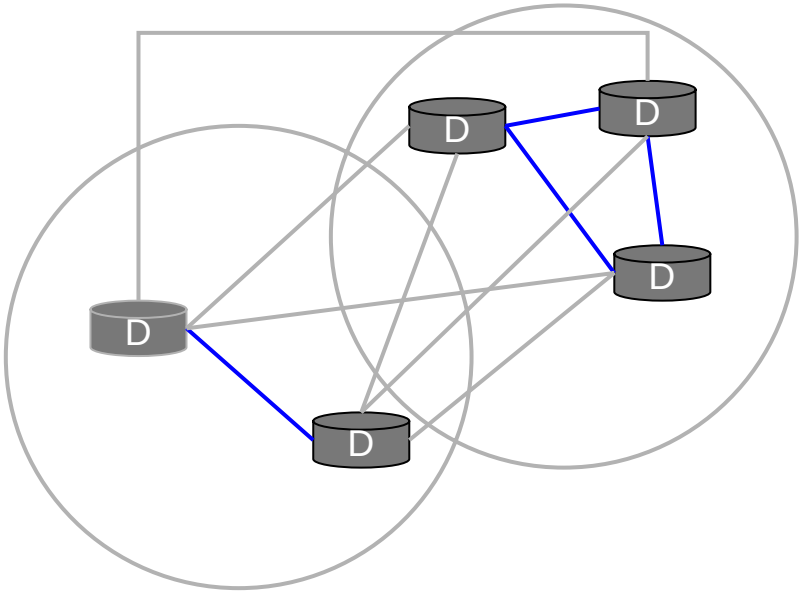
.. is large




Write in one TA?

(+) consistent read
(-) write overhead
(+) availibility

# Spectrum..




(1)**Replica autonomous:** each may independently read / write

(3) **Writes** are propagated *from time to time*

(4)If no updates for some time, replica will **eventually** *have the same state*

**Trade correctness for performance**

# Basic considerations

Freie Universität Berlin

***n nodes***

**WWW: What – Where – When ...**

- **What** to replicate
  - **full replication**
  - **partial replication** (not unusual for DB tables)

    full table, horizontal ("partitioned tables"), vertical (unusual)

- **Where** to update?
  - Update allowed at every node? (**Multimaster**)
  - One particular node (**Master copy**)

# Basic considerations

- **When** to update copies?
  - **Eager** (synchronous) – every replica always equal
  - **Lazy** (asynchronous) – propagate updates to replica after 'commit'

- **Granularity** of **operations**
  - individual read / write operations
  - multi step **transactions** (atomic)

**Consistency?**

# 9.2 Consistency models

Freie Universität Berlin

## Consistency definitions

Focus is on order of single operations, typical view in distributed systems

**strong (strict) consistency**: every read returns value of last write – write must be seen everywhere immediately

**w(x1) r(x1) @1** **r($x2_{old}$) w(x2) @2** not allowed

Write immediately visible: Strong requirement, global clock needed

How does it compare to serializability?

$r(x_i)$ means: read x @ replica i

# Sequential Consistency

**Sequential consistency**:

result is the same as if **all writes are seen in the same order**; may be different from real time order.

1 2

r1[x1] r2[y1] w2[x1] w1[y1] r1[x2] w2[x2] r2[y2] w1[y2]

not serializable, but sequentially consistent

# Causal Consistency

Focus on **data dependency of** operations:

- if a computation *could depend on another*, all nodes seeing these operations (writes) should see them in the same order
- Otherwise order may be different and writes are said to **concurrent**

R1 : w(x)1
R2: r(x)1, w(y)5
causal dependent: y=5 *could* depend on x

# Causal consistency

Freie Universität Berlin

- all **causally related operations** must be seen **in causal order** by all nodes
- not causally related ('concurrent') operations may be seen in different order

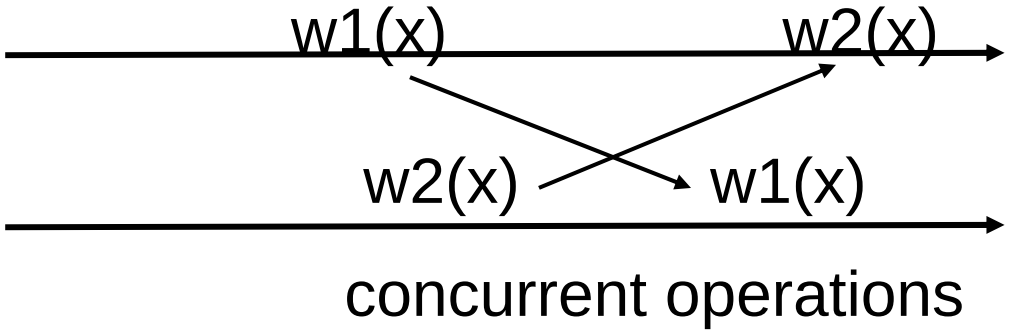
Let $op_1$, $op_2$ be operations on replica Ri, Rj respectively.

$op_1 <_c op_2$ ("**happens before**") if

- Ri = Rj and $op_1$ executed before $op_2$
- or Ri $\neq$ Rj an $op_2$ executed after $op_1$ received and executed
- or there is an $op_3$ $op_1 <_c op_3 <_c op_2$

# Consistency

Example: **causal consistent**




R1: w1(x) w2(x) R2: w2(x) w1(x)

Operations performed on replica in a casually consistent order.

To be clarified: "serializability" in a replicated system scenario

# One-copy-serializability

Basically means:

- the history H of *one copy* is serializable, i.e. has an equivalent serial execution S
- the histories *of all other nodes* are serializable with a *compatible serial execution* S

Formally:

A "Replicated Data history" H over a set of TA is **1-copy-serializable**, if its **Reads-from relation** is the same as from some 1-copy history H' and the conflict graph of H is consistent with that of H'

# Example

2 copies: R0, R1

Transactions TA1, TA2, TA3

r1(x0) r2(x0) w3(z0)r2(y0) w1(x0) r2(y1) w2(y0) r2(x1) r1(x1) w2(y1) w1(x1) w3(z1)

↓

r2(x0) r2(y0) w2(y0) w3(z0)r1(x0) w1(x0)

↓

w3(z1)r2(y1) r2(x1) w2(y1) r1(x1) w1(x1)

Conflicts only between copies of one replica

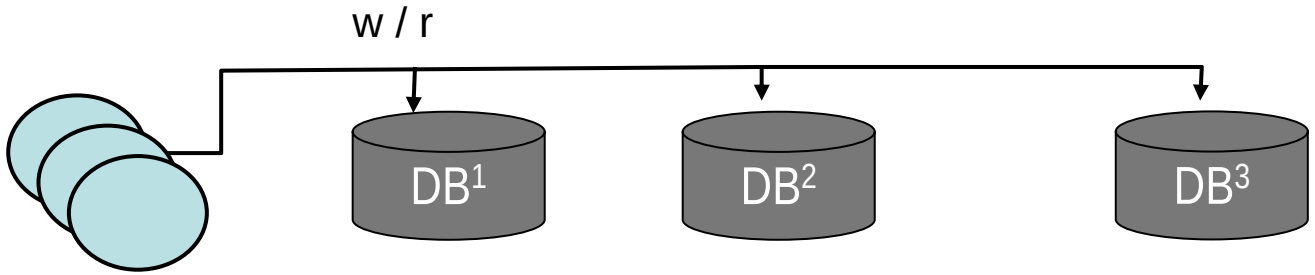
Conflict graph

| |
|---|
| T2 → T1 |
| T3 |

| |
|---|
| compatible serialization |
| T3 → T2 → T1 |

# 9.3 Classification and model

- n replica, m clients
- Total replication (just for convenience, partial replication same methods)
- Transactions initiated by clients @ one replica.
- Read / write model of transactions, lcok synchronization




Find an architecture , protocols and failure handling for keeping replica consistent w.r.t. transactions.

# Consistent replica

- Focus is now on the **state** of replica.

  (1) Transactions should always see the **same state** on all replica or

  (2) **One** replica is always **up to date** with respect to the executed transactions ("primary")

  (3) Transactions can be executed at **any replica**, state of replica must be exchanged.

- Obvious requirement:
  State of all replica should eventually be the same,
  e.g. if system stopped, no further updates running.

# Eventual consistency

A replication system is called **eventually consistent**, if each replica converges against the same state, when no update is performed .

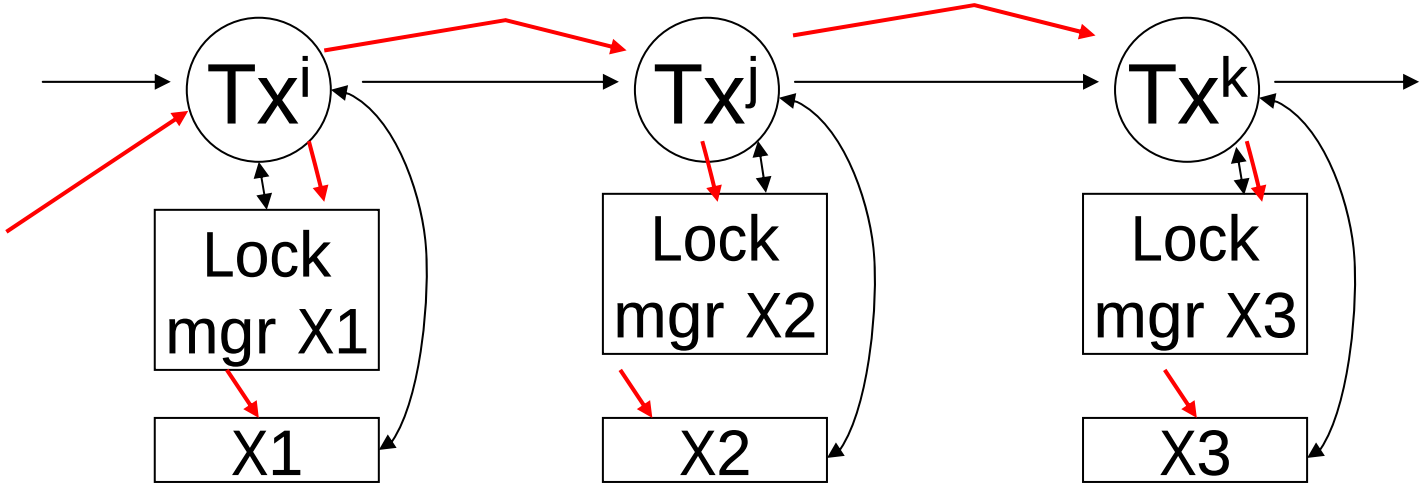
- Some systems are eventually consistent, but never have the same state (e.g. DNS)
- Serializability lost $\Rightarrow$ **conflict resolution** when applying updates to replica
- Failure again a serious problem

# Naiv solution

Freie Universität Berlin

Each copy is an independent data set

Read




Object X has copies X1, X2, X3

read(X): rlock(X1), rlock(X2), rlock(X3)

read(X) somewhere

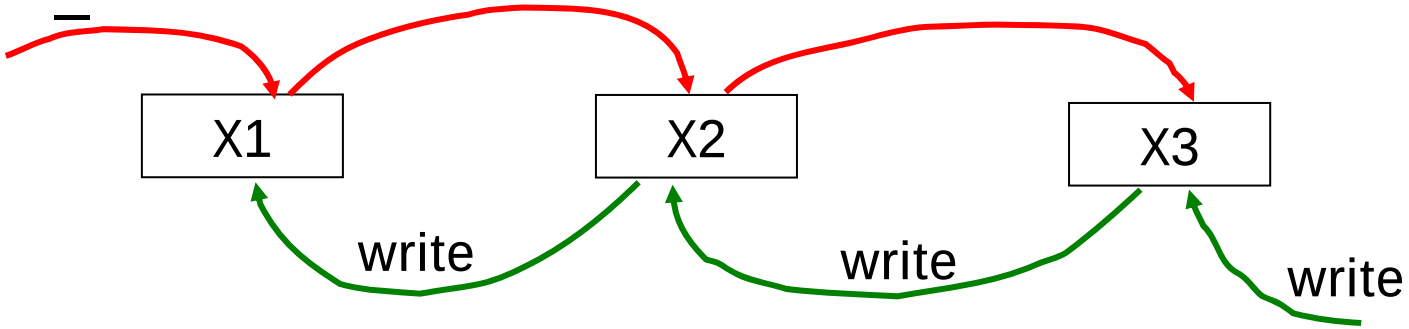
unlock (X1), ..., unlock (X3)

Transparent to application

# Naïve solution: Synchronous write

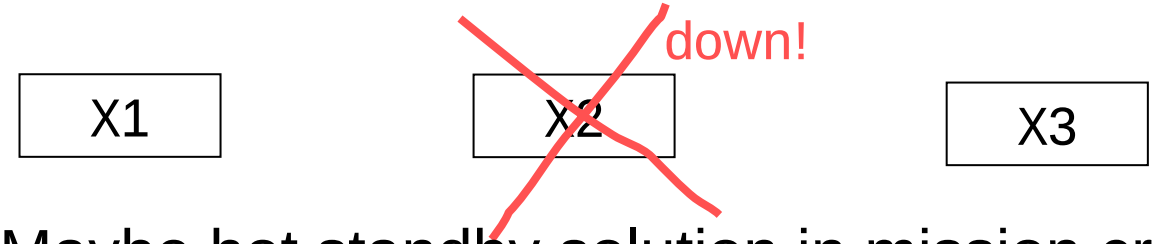
Write(X): wLock(X1), wLock(X2), wLock(X3)
write(X1) , write(X2), write(X3)
unlock(X1),…unlock(X3)

***RAWA**: read all, write all*




One copy serializable but...

LESS availability with standard protocols (2PL, 2PC)

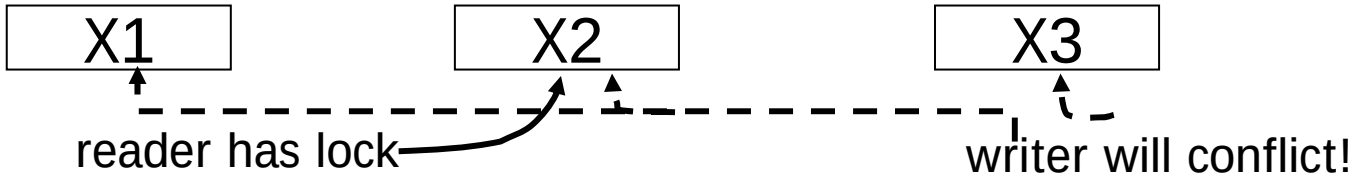



Maybe hot standby solution in mission critical applications

# Variation: ROWA

*Read one, write all (**ROWA**)*

Reader lock one, **Writer locks all copies**

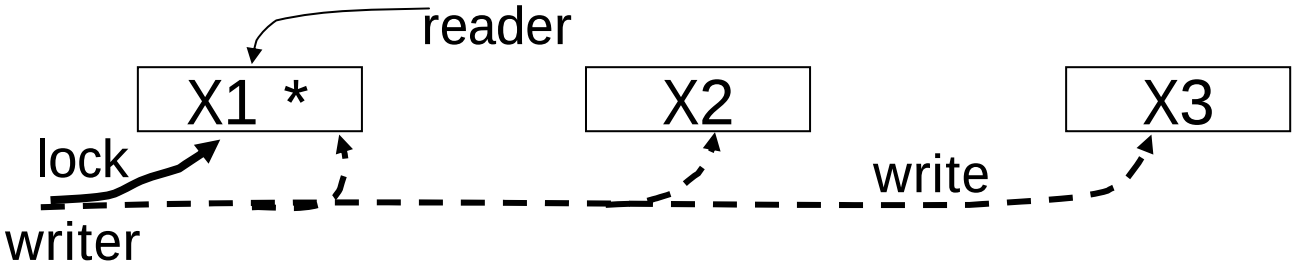



Good for readers – availability, not performance

Low availability for writers: one replica down – all down (for writers)

How to write copies?

# Primary copy (Master copy)




Select primary copy (for now statically)

*Read*: lock and read primary copy

*Write*: lock, write primary copy and all copies

- Could be good for <u>high availability</u> – *depends on updates of replicas*
- Not for performance:
  Potential contention at primary copy

# Basic solution to replication

Read lock all; write lock all: **RAWA** --

Read lock one; write lock all: **ROWA**

Read lock one, write lock all available: **ROWAA**

Read and write lock primary: **RPWP** (availability)

- Updates on replica? See below

- Measure of availability:

  Probability that application can read / write given N sites and probability p for site being operational

# Comparison

n = number of nodes with copies
P = probability that a node is operational

| | P(read avail.) | P(write avail.) |
|---|---|---|
| RAWA | $P^n$ | $P^n$ |
| ROWA | $1 - (1-P)^n$ | $P^n$ |
| RPWP: local commit | P | P |
| RPWP: 2PC | P | $P^n$ |
| | | |

Does RPWP make sense at all?

Yes: e.g. Backup / Mirror,
Data Warehouse copy operation

# 9.4 Update propagation

Freie Universität Berlin

## When are replica updated?

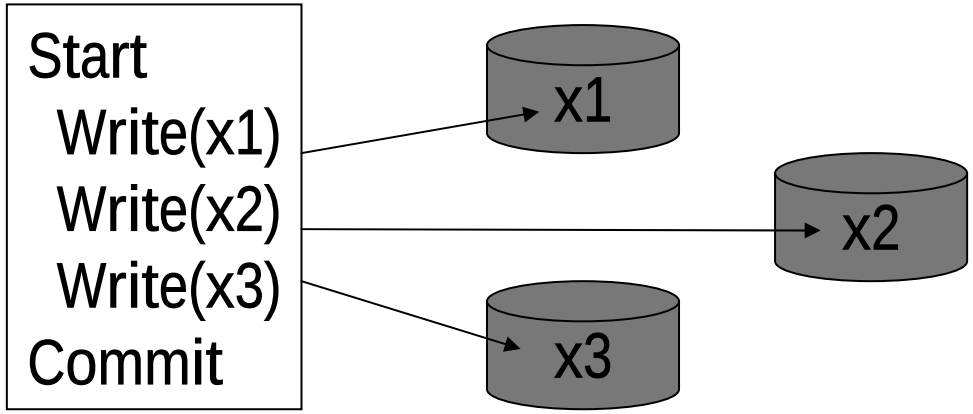
Replicas should behave functionally like non-replicated servers

**Synchronous ( eager) replication** - each transaction updates all replicas of every item it updates as part of TA

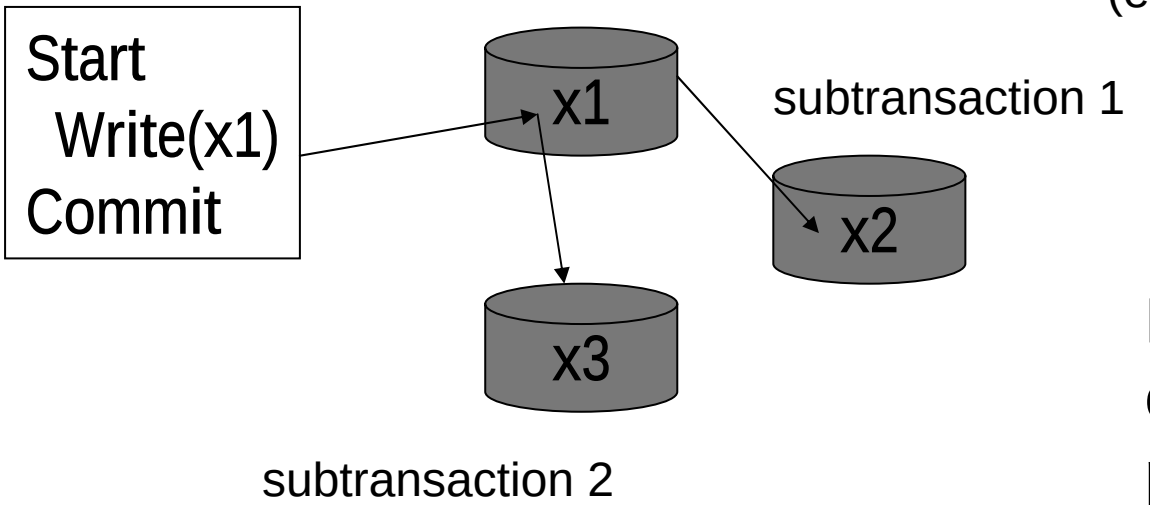
(corresponds to naiv solution of above: RAWA)

- The algorithms are complex and expensive, due to failure and recovery handling
- Too expensive for most applications due to heavy distributed transaction load
- No general-purpose database system products support it

# Synchronous write




Used in some high availability systems with modified 2PC: do not abort, if participating node is gone.
(e.g.HP Reliable TA Router )

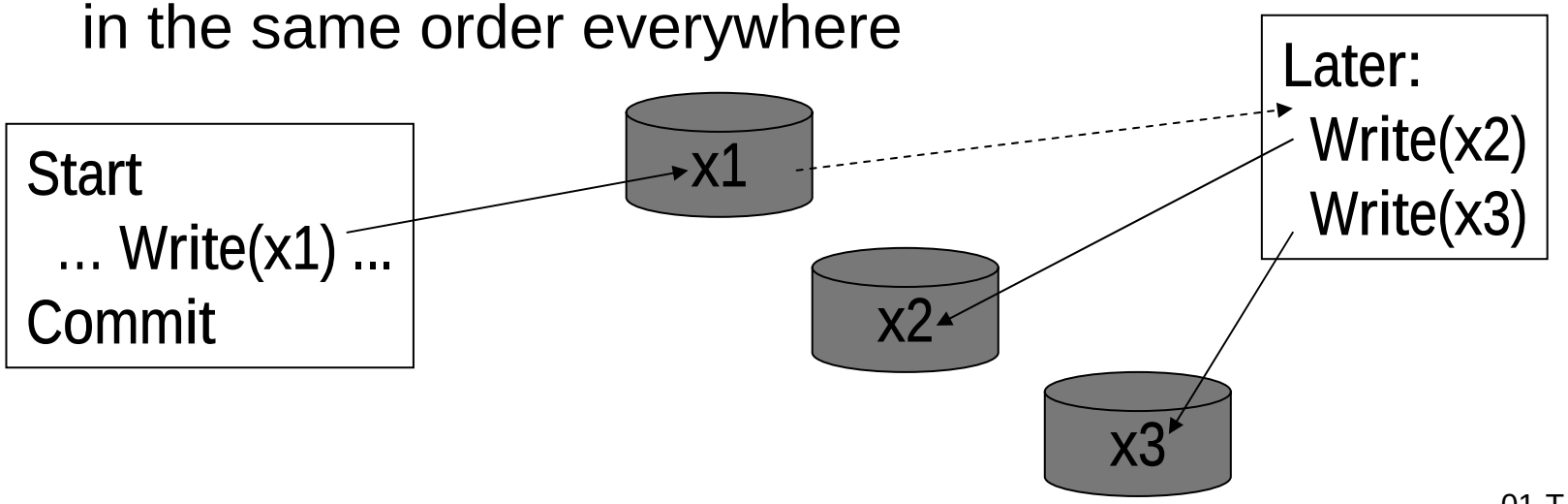



Note: assumes operations to be re-plicated.

# Asynchronous (lazy) Replication

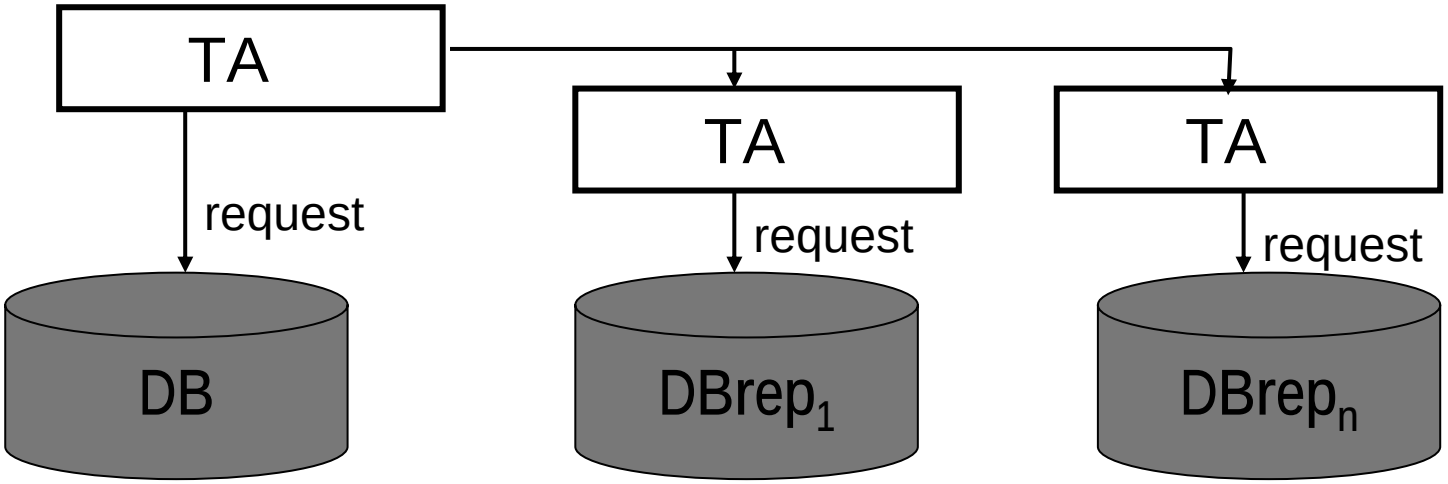
Each transaction updates a replica and the **update** is **propagated later to other replicas**

- Apply **deferred (lazy) updates** to all replicas in **the same order**
- If new updates stopped, replicas would converge to the same state.
- Minimize synchronization expense of applying updates in the same order everywhere



# Asynchronous Update propagation

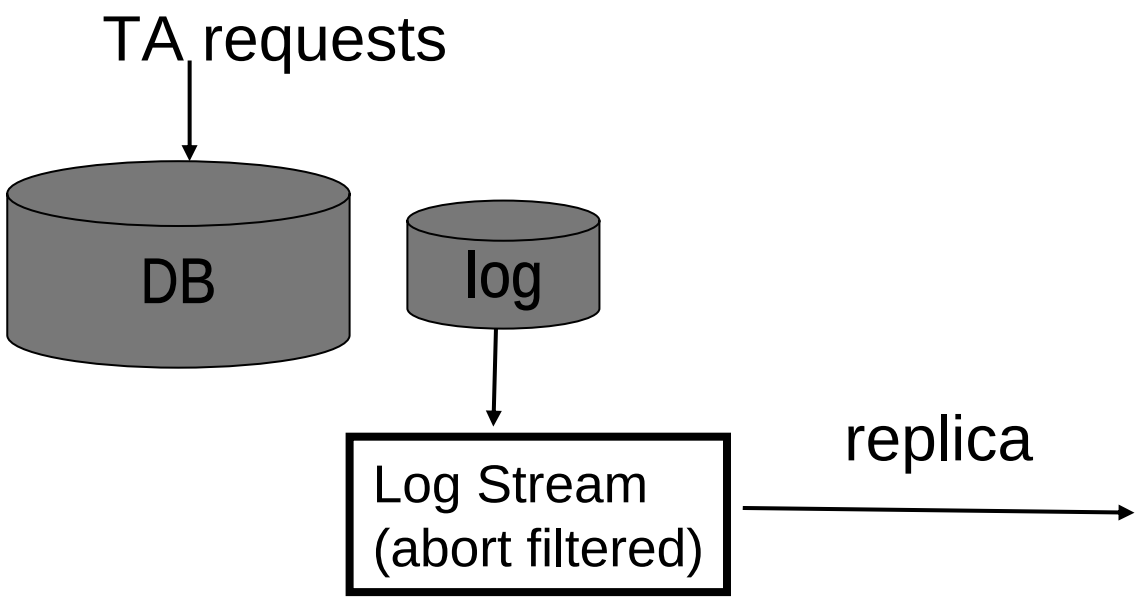
- Transaction replication propagation (when, what?)




- Slow, if requests are processed serially (according to TA-id)
- Variants: concurrently process TA at different replica concurrently in a distributed transaction – and take care of possible **timing differences**

# Asynchronous Update propagation

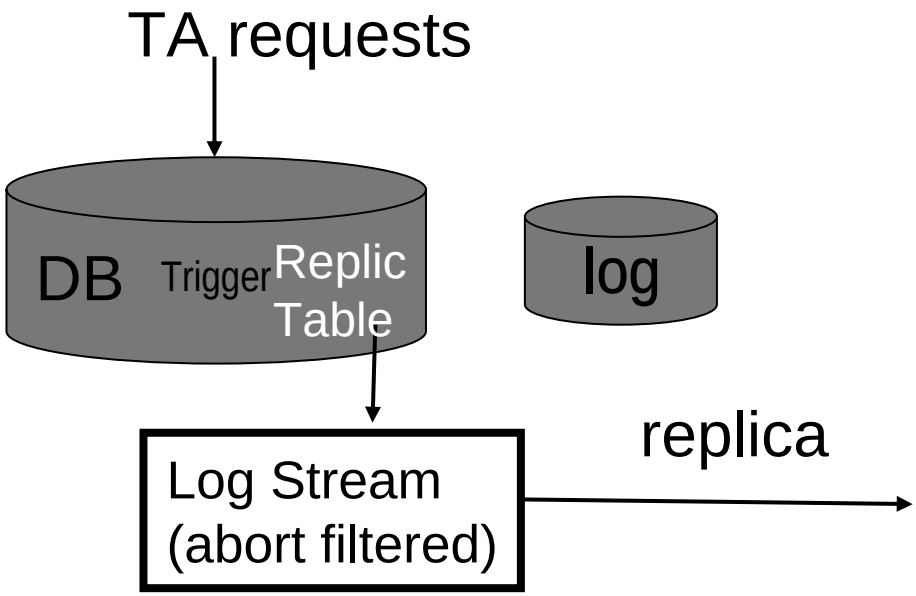
## Log stream propagation




In most systems:
- gather updates in log file (as usual)
- post-process: only committed updates
- send asynchronously to replicas

# Asynchronous Update propagation

"**Log table**" (Replication log table)




Replication **log table** contains all updates, which is transferred to replica periodically

# Asynchronous Update Propagation

Collect updates at primary using **triggers** or the **log**

- Triggers (Oracle8, Rdb, SQL Server, DB2, …)
  - On every update at primary, a trigger fires to store the update in the update propagation table.

- Post-process the log to generate update propagations (SQL Server, DB2, Tandem Non-Stop SQL f)
  - Off-line, so saves trigger and triggered update overhead, though R/W log synchronization also has a cost
  - Requires admin (what if log reader fails?)

Optionally identify updated fields to compress log

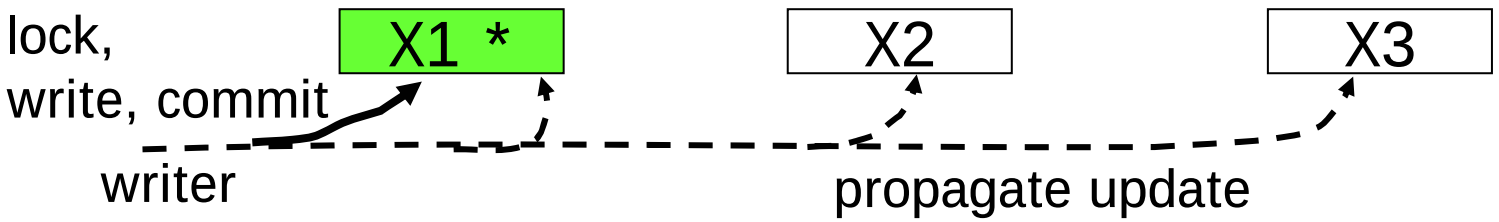
Most DB systems support this today

# Lazy update propagation: processing at replica

Update *a primary copy* (master) and *commit locally*

Send updates to copies ("secondaries") as a log stream

But: TA1: r1(x) ; x = x+1, w1(x)@X1
TA2: r2(x) ; x = x*1.6, w2(x)@X1 Preserve order of updates!

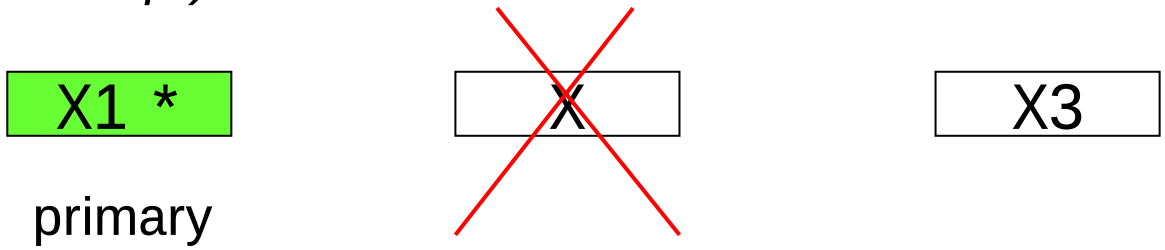



Write(X):
Get exclusive X1* lock
Write new value into X1*
⋮
Commit at primary; get sequence number
⋮
Perform X2, X3 updates in sequence number order

Note: only "state changes of DB" → replica, NOT whole TAs

# Availability - Primary copy

*Copy ("secondary", "backup") failure*




- Only **restart protocol** needed for a replica X
- Take standard restart, but get additional log entries from primary for updates missed in down-phase
- Slightly more complicated than recovery from local log: crash (of primary or X) during updates?

But: recovery must be idempotent anyway

# Availability

*Primary failure*

Avoid failure of the whole system: **without primary NO write**

Solution: a secondary (copy) is promoted to ***new primary***

Steps to take:

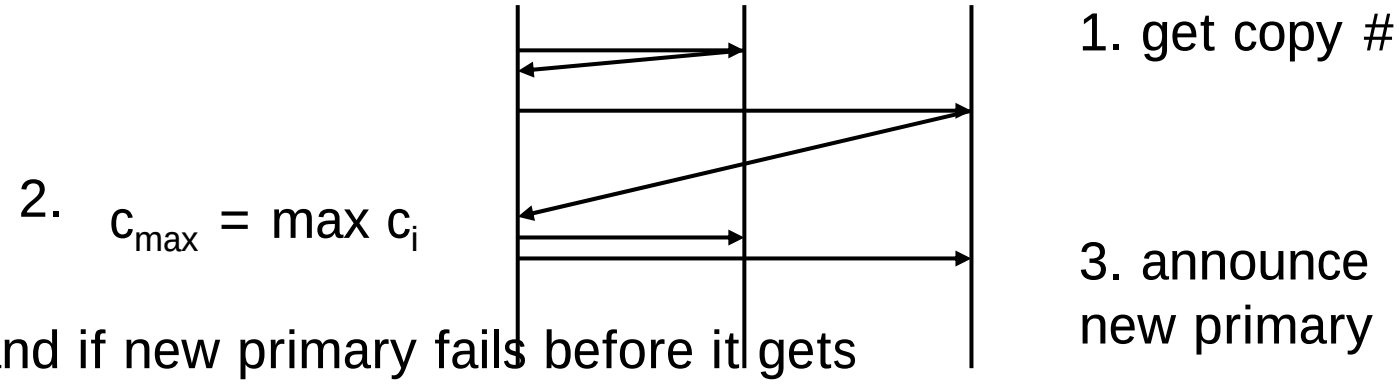
- Agree on *new primary* among secondaries
- Ensure new primary has seen all previously *committed transactions*
- Resolve *pending transactions*
- Resume processing

# Elect new primary

Presumption: every secondary has a **copy-number c,** secondary with **maximum c wins**

How?

- First secondary s1 which notices primary failure asks for copy number $c_i$ of every secondary (additional assumption: every node knows all copies)

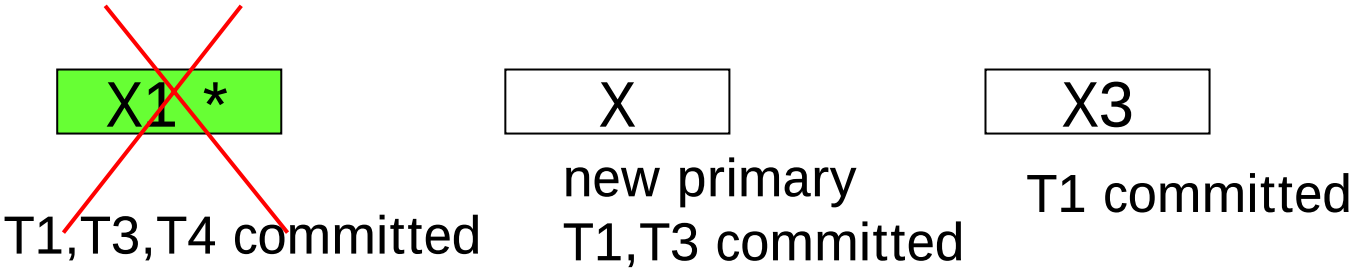



... and if new primary fails before it gets the announcement?

# Primary copy: takeover

Make sure that new (elected) primary has received all updates of former primary

a) asynchronous update




T2: abort when X1 recovers (as a secondary)

T3: X sends updates to X3

... but **how does X know??**

Solution: during election everyone sends id of last log-entry

T4 is lost !

# Primary copy: takeover

b) synchronous update

~~X1 *~~

T3 committed,
T4 ?

X

new primary
T3, T4: Wait for
TA-coordinator

X3

T3 committed
T4: Wait for
TA-coordinator

2PC: Blocking!

# Majority vote / quorum consensus

Freie Universität Berlin

Is there something **between ROWA and primary copy**?

Yes: don't acquire lock everywhere but only a "sufficient number" of locks

n copies, if a writer has acquired $m > n/2$ locks, no other writer can write on the locked object (assumption: all objects have same state!)

→ *Majority consensus*

Useless with two copies ⇨ assign weights to copies

# Quorum consensus

For every object assign weight $v_i$, $\mathbf{v} = \Sigma\, v_i$

Let

**r** = quorum for read

**w** = quorum for write

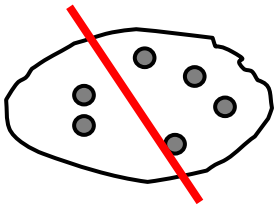
The following invariants must hold:

(1) $w > v/2$ "at most one writer"

(2) $r+w > v$ "no write during reads"

May be used in case of net partitioning

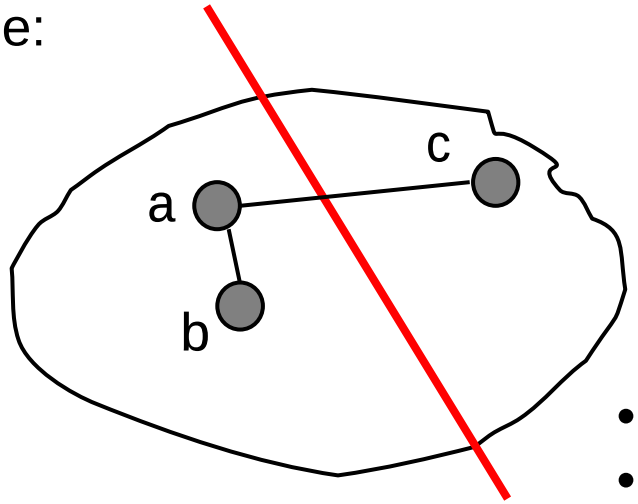
Voting will decide,
which part may
carry on with a new (dynamic) primary



# Quorum consensus

Correctness?

Counterexample:

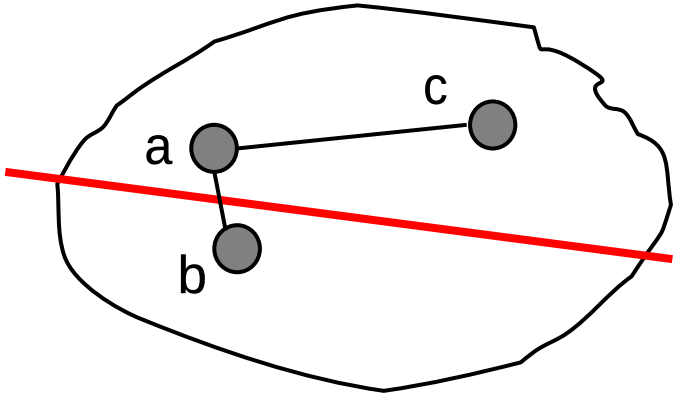



- One vote ($v_i$ =1) for each node
- replicated data x

Transaction T: x=x+1 at a and b (majority)

Communications link a – c recovered, but...

# Quorum consensus




T2: read(x) at c // stale value!
T2 write(x) at a,c

Inconsistency,
not recoverable

Solution?

- Each node: list of committed TAs
- Before quorum is built:
  - compare list of committed TA
  - apply missing updates
- resume

# Quorum consensus

Freie Universität Berlin

## Can quorum consensus be used for synchronization?

In some sense – combined with versions (or timestamps):

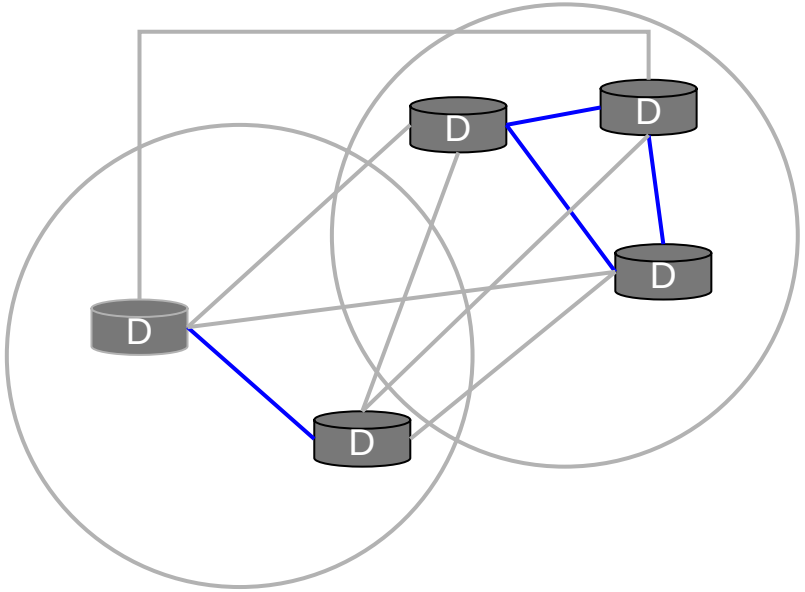
- **Timestamp for each object**
- read (x) : read copy with maximum timestamp ts
  r+w > v implies: at least one copy was member of the last write quorum
- write(x) : write any copy and increase ts of copy read

Only for synchronization of replica update, not general TAs

Special cases:

- **ROWA** as special case: **r = 1, w = v**
- **Primary copy: weight of pc = 1, w = r = 1**

# Lazy update: the general case




"**Multimaster Replication**"

- Transactions read / write an **arbitrary copy** – typically the local one
- **TA effects are propagated lazily**
- **Conflicts** have to be solved
- Global sequence numbers?
- Consistency?

# Multimaster Replication with primary copy

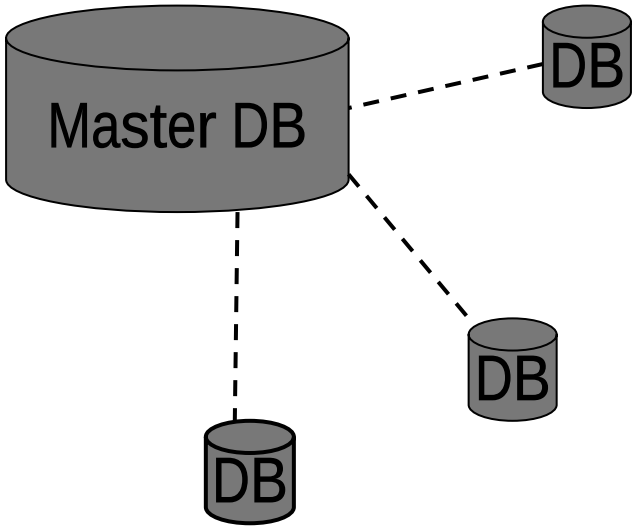
## Independent update at two or more sites

Does it make sense??

Particularly useful, when each master only responsible for part of the data (e.g. salesperson with his customers and their orders)

(1) Go offline
(2) Update
(3) Reconnect
(4) Exchange updates with other masters

Low conflict potential, but….




Disconnected copies

# Multimaster Replication

Race condition:

| *Replica 1* | *Primary* | *Replica 2* |
|---|---|---|
| Initially x=0 | Initially x=0 | Initially x=0 |
| T1: X=1 | | T2: X=2 |
| Send (X=1) | | Send (X=2) |
| | X=1 | |
| | Send (X=1) | |
| | X=2 | |
| | Send (X=2) | |
| X=2 | | X=1 |

*Bad: end up in different states*

# Multimaster Replication

Freie Universität Berlin

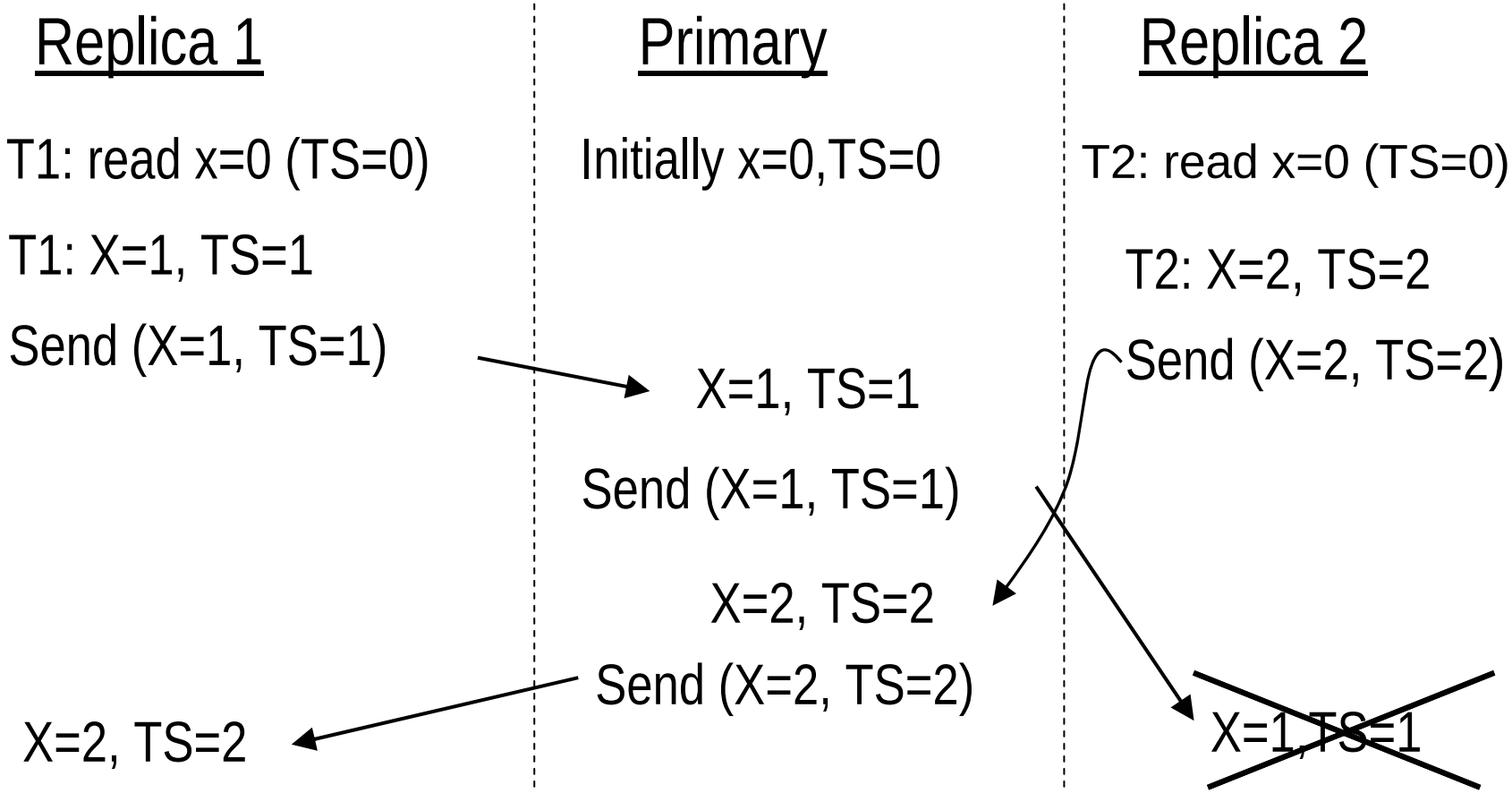
Employ *Thomas write rule* when updating primary state:

- every object has timestamp ts(x)
- apply update of x at replica to primary only
  if ts(x)@replica < ts(x)

If isolation of transactions is a goal,
this only works for blind writes and inserts
e.g. change address of customer,
insert a new order record,
but not: increase tax by one percent

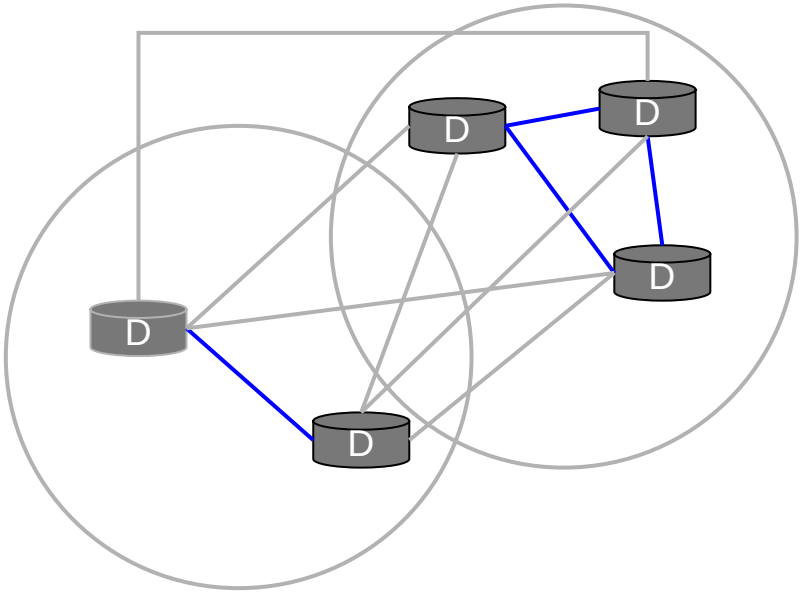
Important for server based "Mobile Databases": Oracle Lite,
…

# Thomas write rule: no guarantee for serializabilty




- Replicas end in the same state, but neither T1 nor T2 reads the other's output, so the execution isn't serializable.

# 9.6 Multimaster replication




"All citizens are equal"

- May results in **unsolvable conflicts**
- Detected when updates are propagated
- Need auxiliary data which reflect the update of object x at different sites

Not only an academic study: in some applications data may be updated according to geo location: "Employees in Berlin / New York " . Updates primarily at home location.