HPI Hasso Plattner Institut
IT Systems Engineering | Universität Potsdam

Distributed Data Management
Replication

Thorsten Papenbrock

F-2.04, Campus II

Hasso Plattner Institut

Motivation

Scalability (Elasticity)

- If data volume, processing, or access exhausts one machine, you might want to spread the load on more machines.

Availability (Fault Tolerance)

- If data availability must be guaranteed even in the presence of failures and machine crashes, you might want to keep multiple distributed copies where one can take over for another failed one.

Latency

- If data is accessed from various locations, you might want to keep the data local to where it is needed.

➢ These requirements demand for replication and partitioning

Replication vs. Partitioning

Replication

our focus now

- Store copies of the same data on several nodes
- Introduces redundancy
- Improves scalability (parallel I/O; no memory scalability!)
- Improves availability (nodes can fully take the load of failed nodes)
- Improves latency (requests can be served by the closest/underutilised node)

For reads!

Partitioning

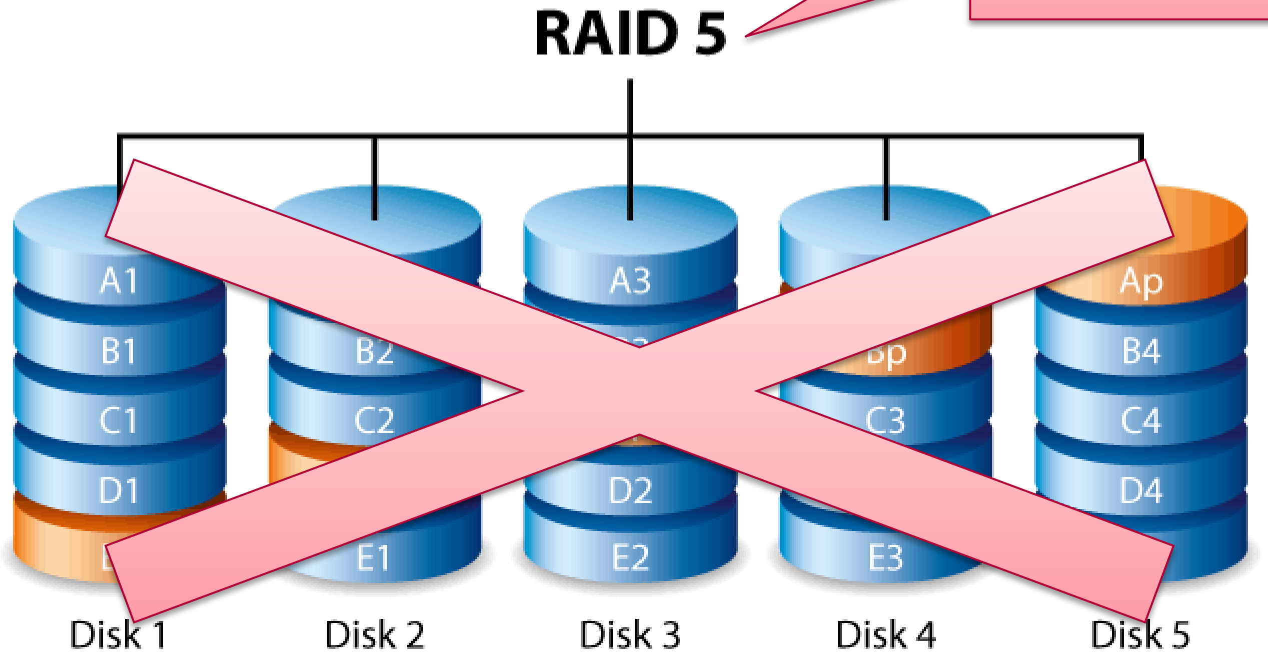
- Store the data split in subsets (partitions) on several nodes
- Also known as sharding
- Improves scalability (some parallel I/O; memory consumption)
- Improves availability (node failures take out only parts of the data)
- Improves latency (place partitions close to where they are accessed most)

Different mechanisms but usually used together

RAID?

Redundant Array of Independent Disks

- Usually make use of special hardware (RAID controller)
- Run the same, centralized logic on top (only storage distribution)



Replication

Challenges

1. Each node must be able to store a copy of the entire dataset
   - ➢ Use partitioning if not possible

In next chapter!

2. Change must be propagated to all other nodes
   - ➢ Single-leader, multi-leader, or leaderless replication algorithms

Replication Algorithm

- An algorithm that propagates changes to all replicas
- Replica:
  - A compute node that stores a copy of the data
- Leader-based:
  - A replication algorithm where (one or more) dedicated compute nodes are responsible to propagate change

Leader-based Replication

Leader

- Dedicated compute node (usually also a replica) responsible for propagating changes
- Also known as master or primary
- Accepts read and write queries
- Sends changes as replication log or change stream to followers

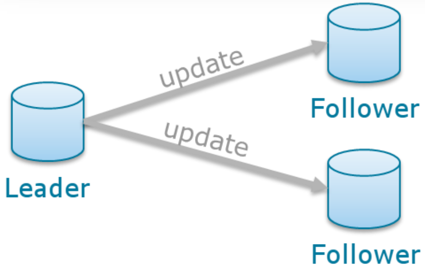
Follower

- General replica
- Also known as slave, secondary, or hot standby
- Accepts only read queries
- Receives changes from leader(s) and updates local copy accordingly:
  - Applies all writes in the same order as applied on the leader(s)

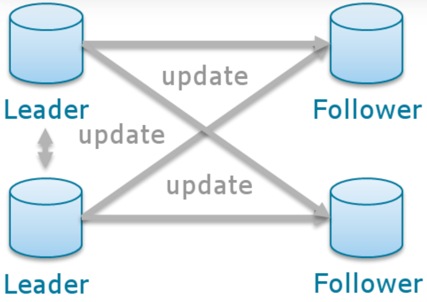
Overview

Replication

Single-Leader Replication



Multi-Leader Replication



Leaderless Replication

Replication

Single-Leader Replication	Multi-Leader Replication	Leaderless Replication
		

- Most relational databases:
  - PostgreSQL, MySQL, Oracle, …
- Non-relational databases:
  - MongoDB, RethinkDB, Espresso, …
- Message-passing frameworks:
  - Kafka, RabbitMQ, …

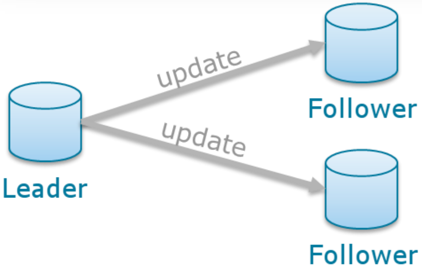
Usually single-leader

- Dynamo, Riak, Cassandra, Voldemort, …

Gaining popularity

Replication

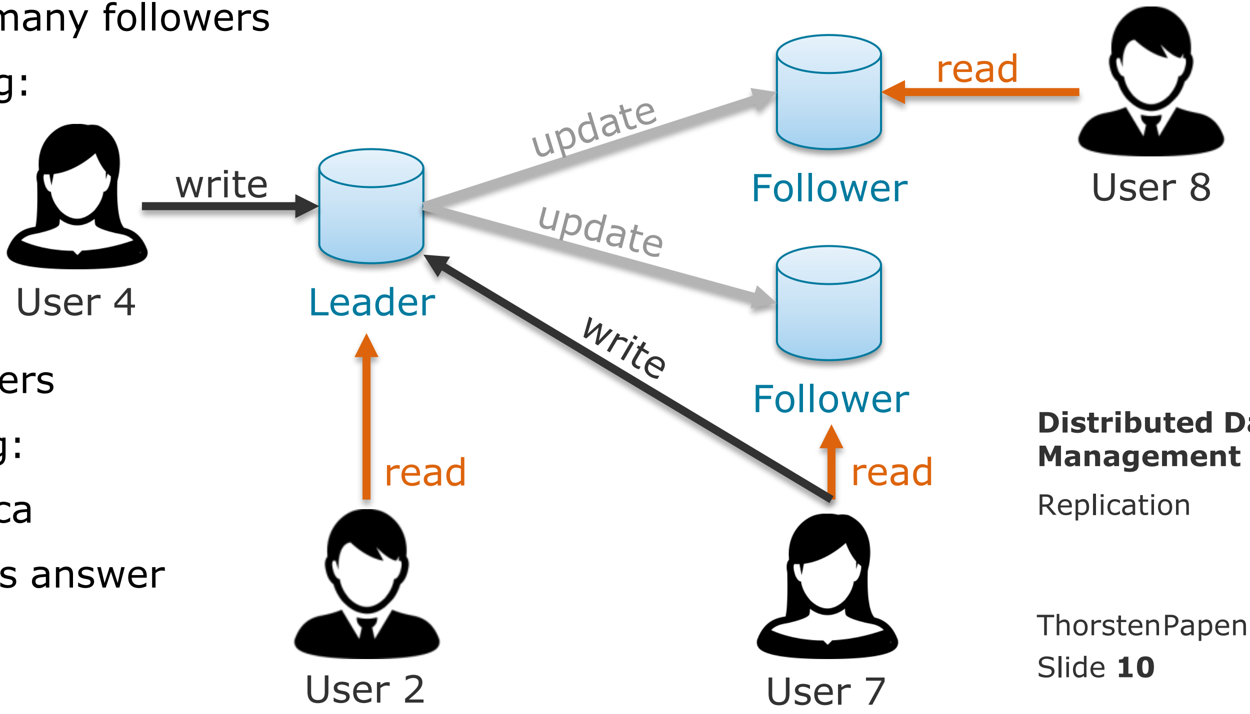
Single-Leader Replication	Multi-Leader Replication	Leaderless Replication



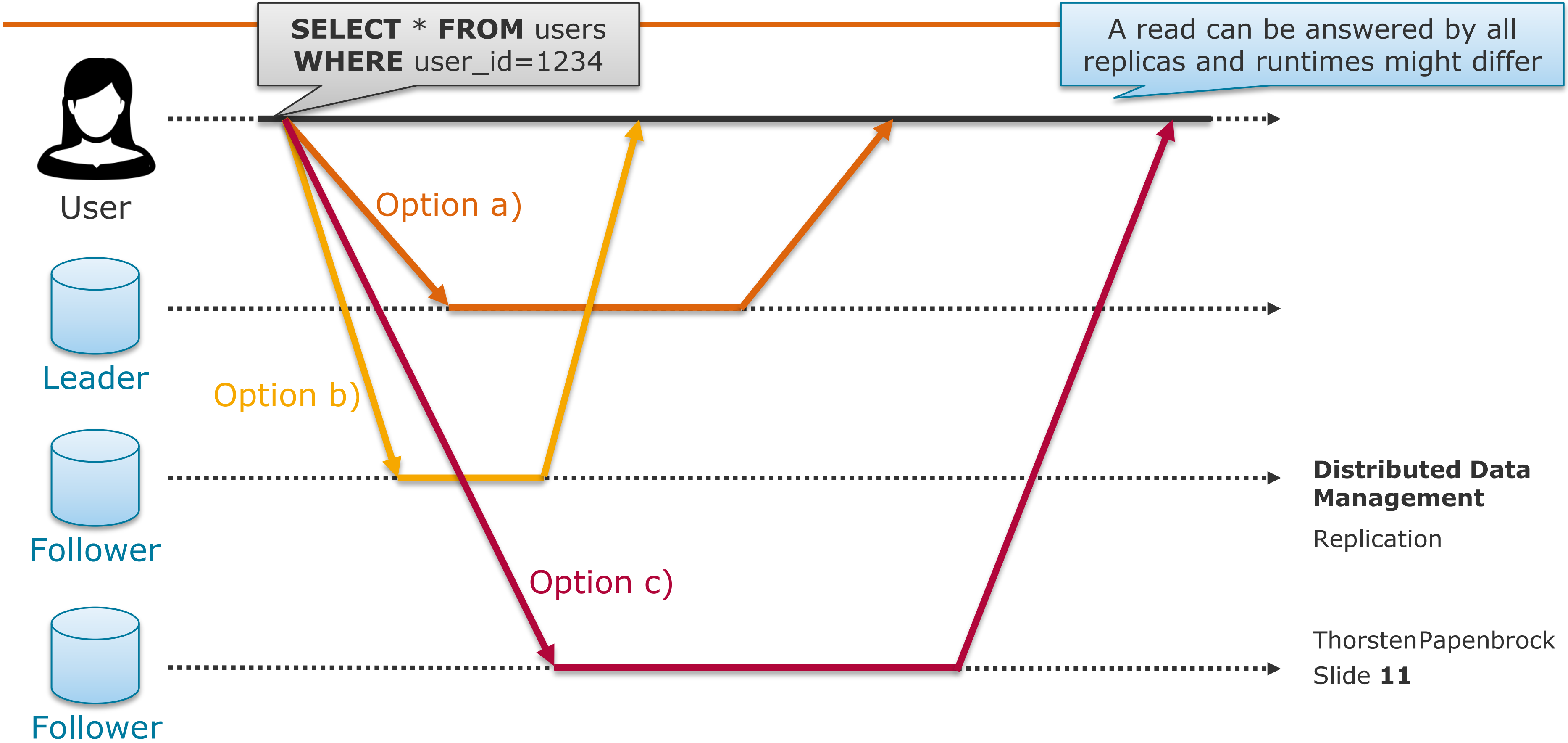
Concept

Single-Leader Replication

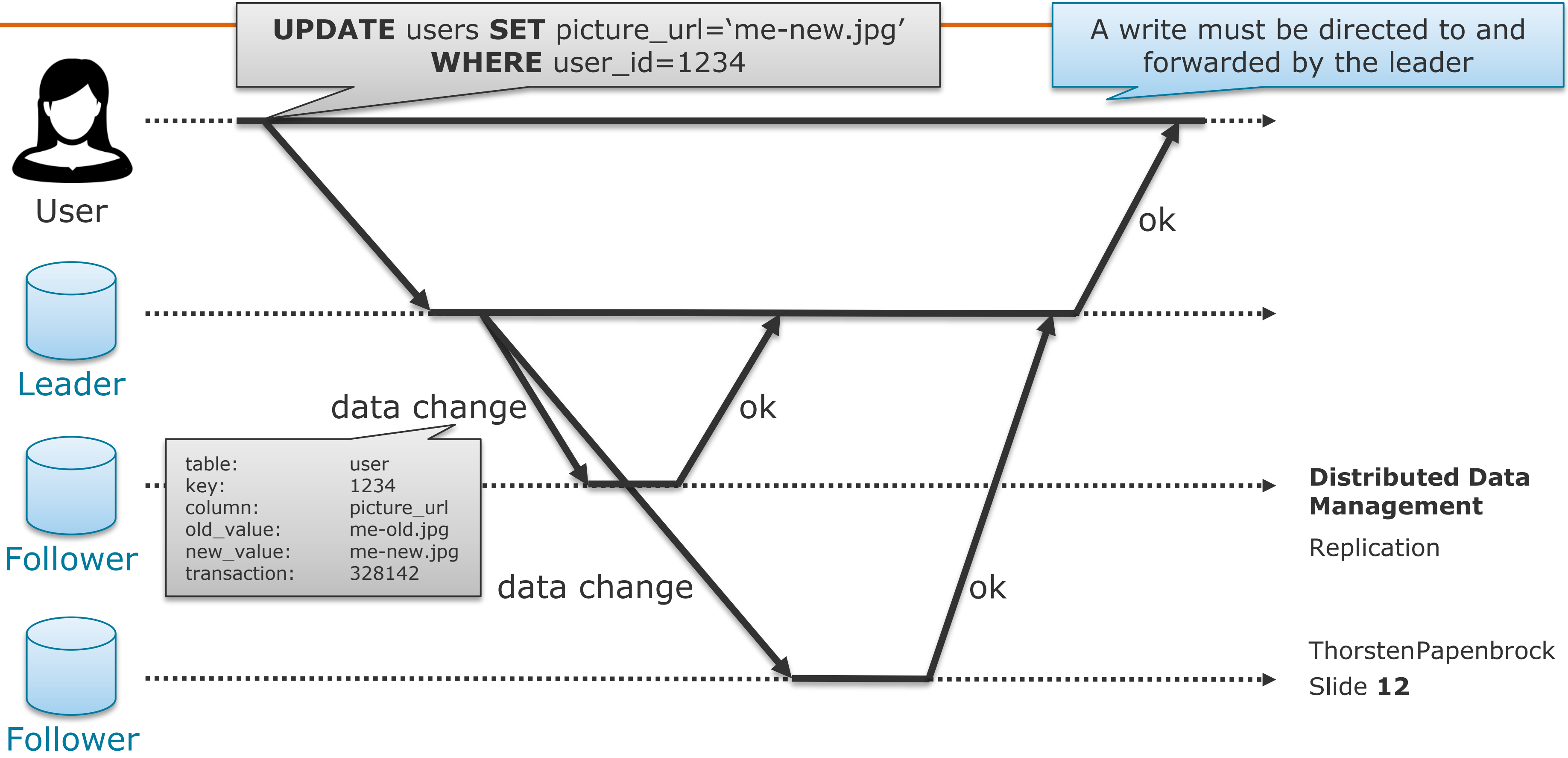
- One leader, arbitrary many followers
- Write-query processing:
  - Send to leader
  - Leader updates local storage
  - Leader sends changes to followers
- Read-query processing:
  - Send to any replica
  - Replica formulates answer from local data



Read

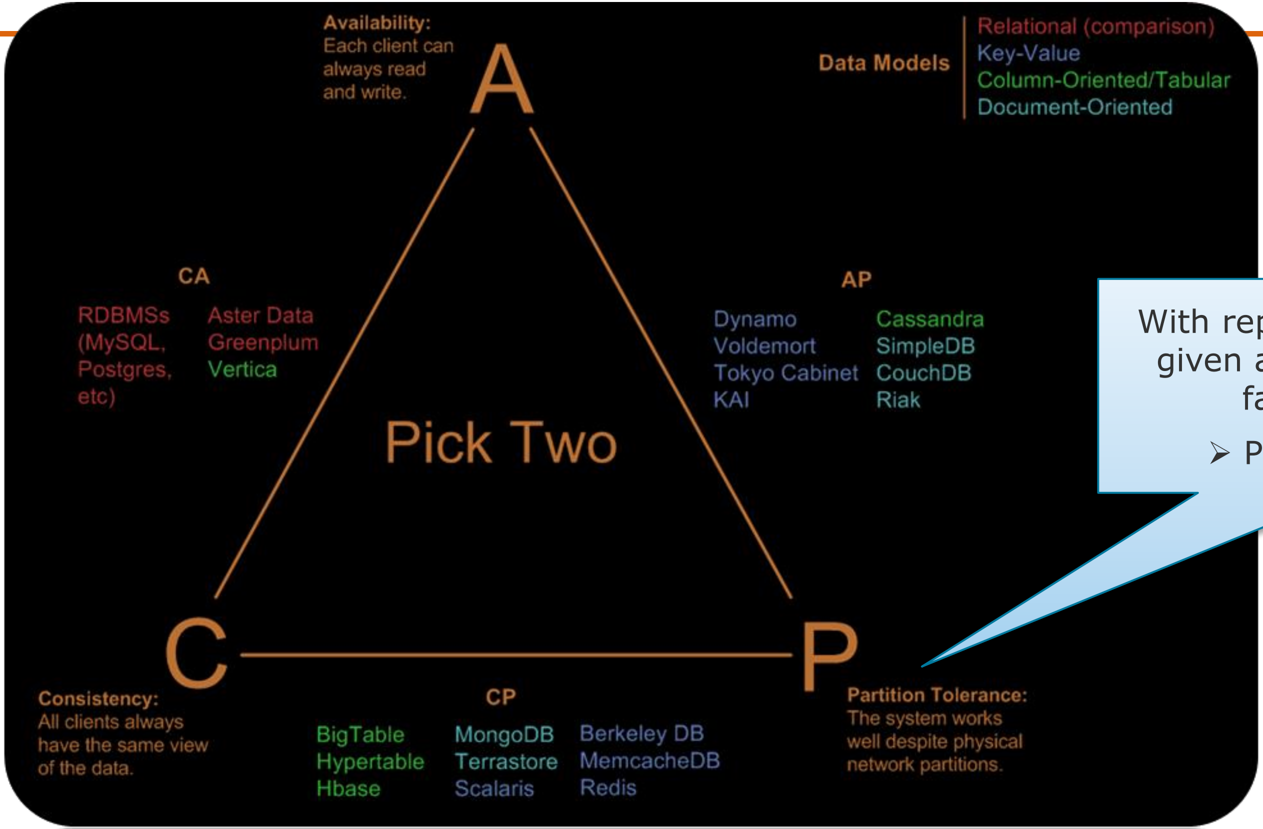

SELECT * **FROM** users
WHERE user_id=1234
A read can be answered by all replicas and runtimes might differ
User
Leader
Follower
Follower
Option a)
Option b)
Option c)

Write



UPDATE users **SET** picture_url='me-new.jpg'
WHERE user_id=1234
A write must be directed to and forwarded by the leader
User
Leader
Follower
Follower
data change
data change
ok
ok
ok
table: user
key: 1234
column: picture_url
old_value: me-old.jpg
new_value: me-new.jpg
transaction: 328142

CAP (Repetition)



With replication, partitions are a given and we need to tolerate faults and outages!

- P is set; drop A or C?

http://wikibon.org/wiki/v/21_NoSQL_Innovators_to_Look_for_in_2020

Write Propagation

Synchronous

- Write query returns when all replica updates returned
- Guarantees that write is system-wide applied when query returns
  - ➢ If leader fails, any follower can replace him
- Slow: unresponsive/crashing followers block all write queries

Drop Availability!

Semi-Synchronous

- Write query returns when one replica update returned
- Guarantees that the leader and at least one replica processed the write
  - ➢ If leader fails, at least one follower can restore its state (not trivial)
- Relatively fast: one response is quickly received even if some followers are slow

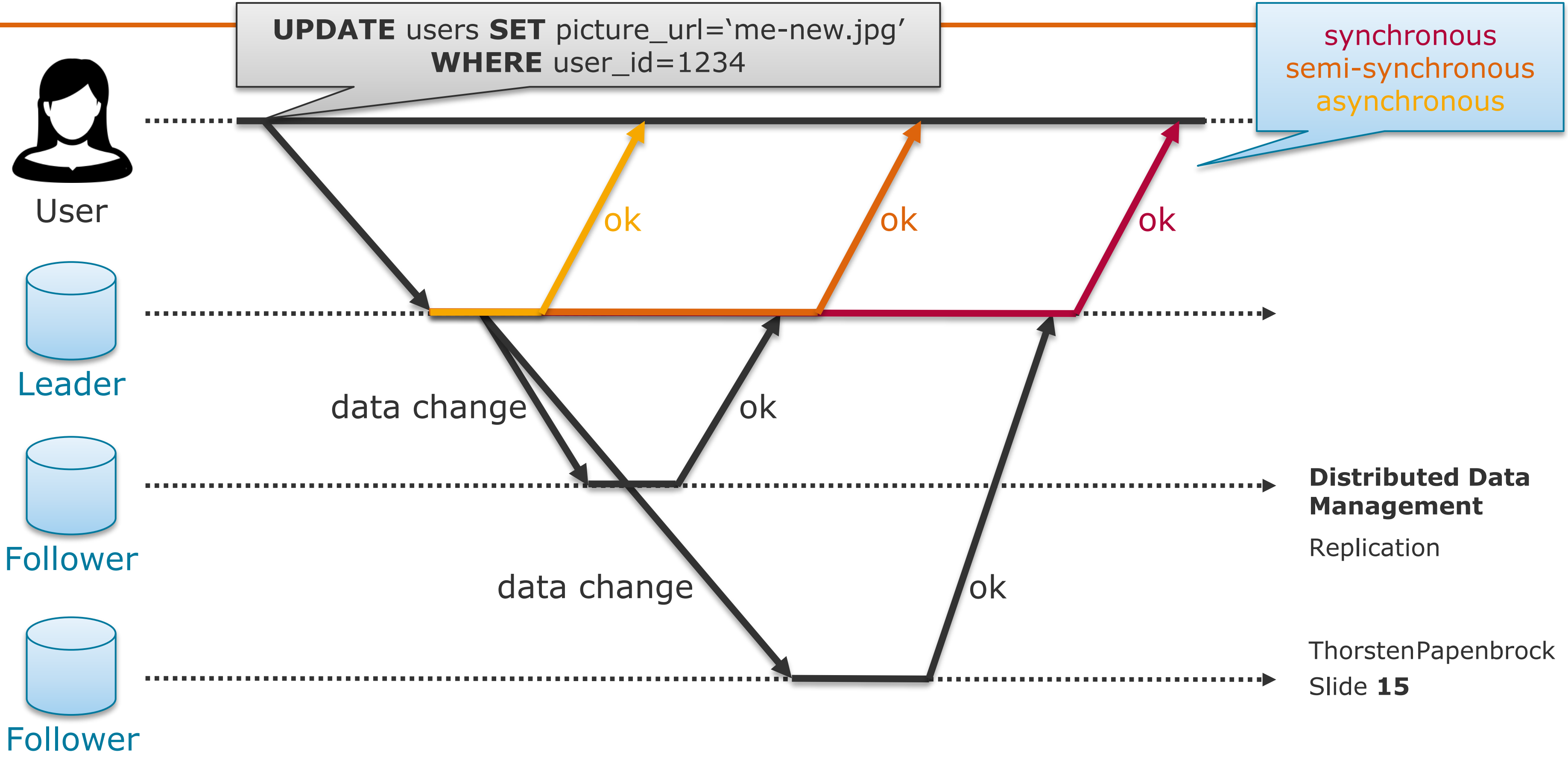
Drop both a bit!

Asynchronous

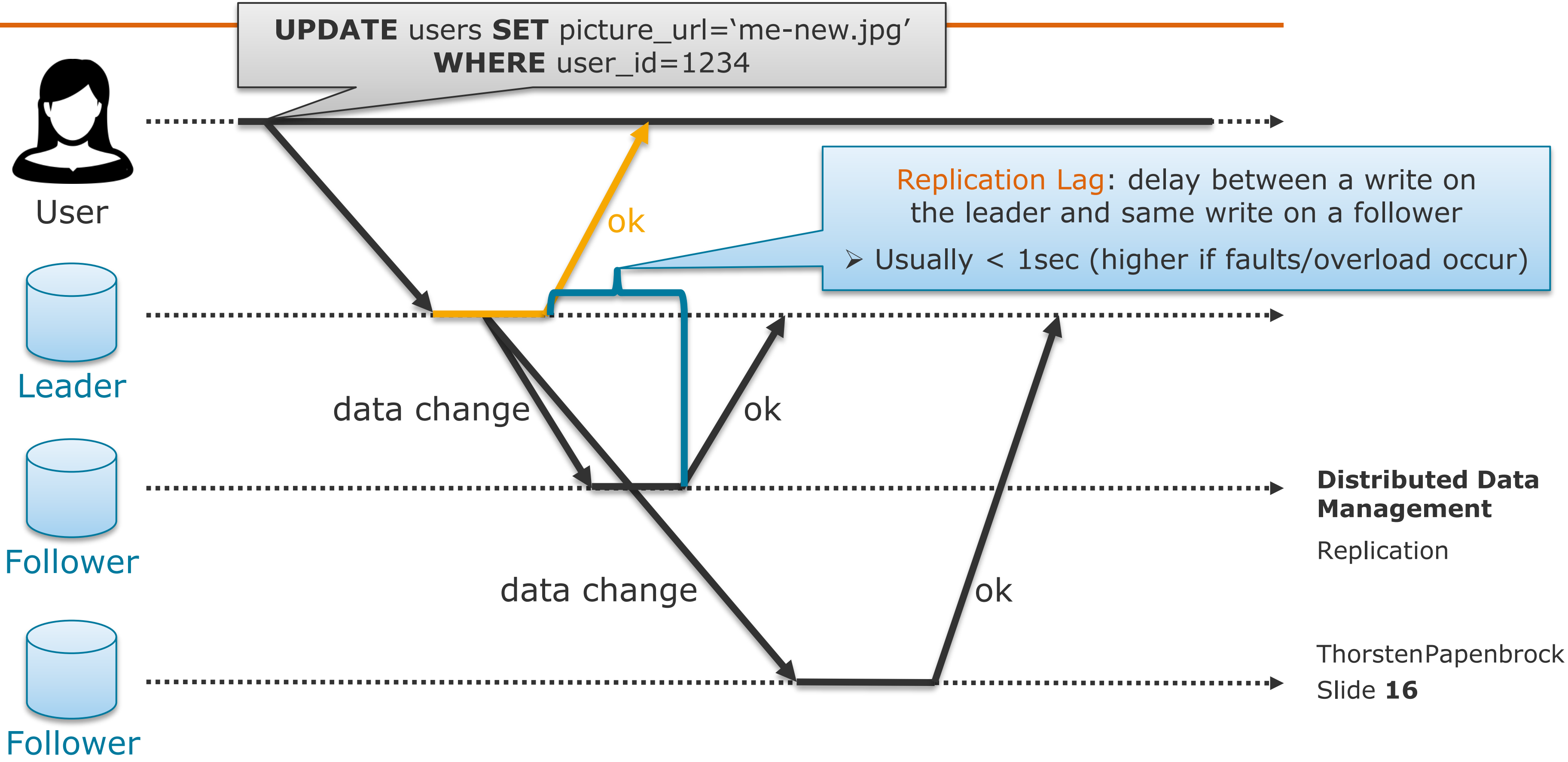
- Write query returns immediately
- No guarantees
  - ➢ If leader fails, writes might get lost; reads to different replicas may be inconsistent
- Fast: no waiting

Drop Consistency!

Write Propagation

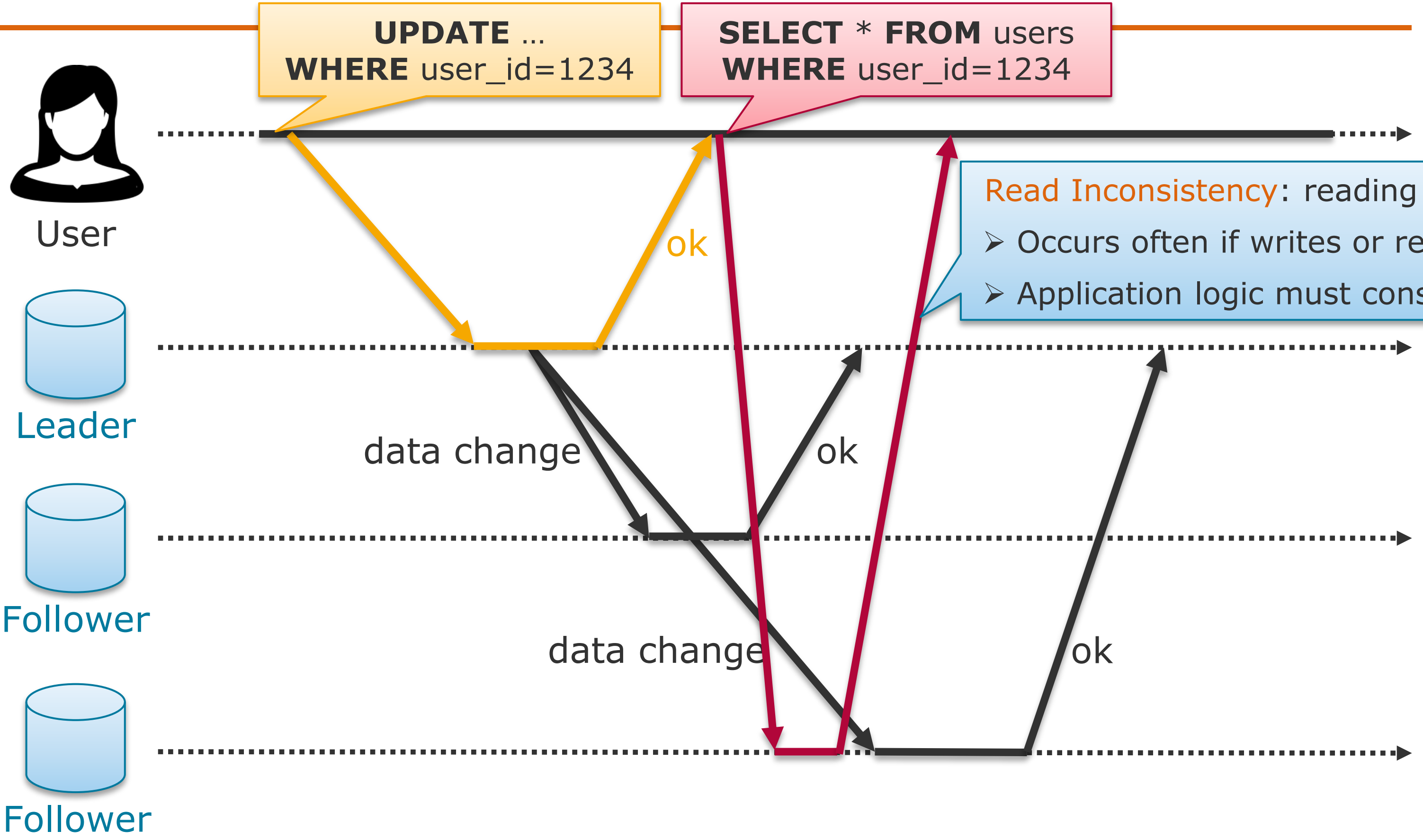


Replication Lag



Single-Leader Replication

Replication Lag



Read Inconsistency: reading of outdated data

- Occurs often if writes or reads are frequent
- Application logic must consider this!

BASE (Repetition)

BASE

- The BASE consistency model relaxes CAP dimensions:
  - **Basic Availability**: The database appears to work most of the time.
    - ➢ Availability might be less than 100%
    - ➢ "Most of the time" is often quantified as lower bound, e.g., 90%
  - **Soft-state**: Stores don't have to be write-consistent, nor do different replicas have to be mutually consistent all the time.
    - ➢ Stored data might be inconsistent, but the store can derive consistent states
  - **Eventual consistency**: Stores exhibit consistency at some later point (e.g., lazily at read time).
    - ➢ Usually consistent within milliseconds
    - ➢ Does not mean "no-consistency", which would be fatal for a store

Drop availability by being more **synchronous**

Drop consistency by being more **asynchronous**

Achieving some Consistency

Read-your-writes Consistency

- Queries should at least reflect all changes made by the same user
  - ➢ Redirect all reads to user-modified data to the leader

Update profile → Read profile

- Implementation examples:
  - a) Remember what data has changed and redirect related queries
  - b) Redirect all queries for X seconds after last own update

Monotonic Read Consistency

Multiple website refreshes

- A repeating query should always give the same result
  - ➢ Direct all reads to the same replica

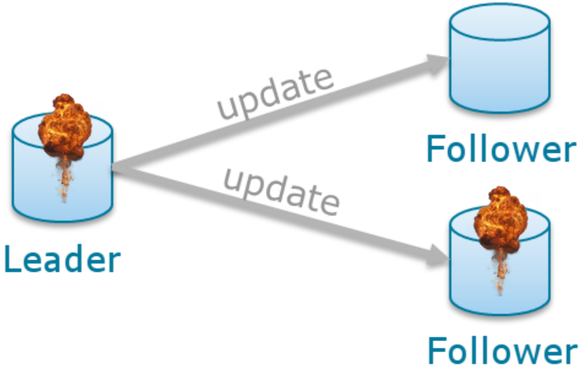
Consistent Prefix Read Consistency

Booking → Payment → Delivery

- Queries should see changes with a certain order in exactly that order
  - ➢ Always apply updates in the same order on all replica

Single-Leader Replication

Handling Node Outages



Follower failure: Catch-up recovery

- After failure handling (error handling, reconnect, restart, …):
  1. Replay log, if necessary, and look up last update in log
  2. Request all updates since last log entry from leader

This procedure is also used for other distributed systems, such as **Actor Systems**

Leader failure: Failover

1. Determine leader failure
   - If leader does not respond for a certain time, assume it to be dead
2. Choose a new leader
   - Either start a new leader, let all followers elect one of them as new leader, or let a controller node decide for a leader
   - Usually the follower with the most up-to-data data
3. Reconfigure system
   - Redirect write queries, make old leader a follower if it comes back, …

Single-Leader Replicat

Handling Node

Follower failure: Catch-up

- After failure handling (…):
  1. Replay log, if ne log
  2. Request all upda r

Leader failure: Failover

1. Determine leader failure
   - If leader does not respond for a certain time, assume it
2. Choose a new leader
   - Either start a new leader, let all followers elect one of t leader, or let a controller node decide for a leader
   - Usually the follower with the most up-to-data data
3. Reconfigure system
   - Redirect write queries, make old leader a follower if it comes back, …

Timeout dilemma: If the timeout occurred because of load spikes, failover handling can make it worse!

Because of these and further problems, many operations teams do failovers only manually

Split Brain: If two replica think they are leaders in single-leader setups, they can corrupt the data!

Implementation of Replication Logs

a) Statement-based replication

- The leader logs the INSERT/UPDATE/DELETE statements that it gets and sends these also as data changes to the followers
- Problem: non-deterministic functions (e.g., NOW() or RAND()), auto-increment columns, and side effects (e.g., trigger or stored procedures) might evaluate differently on each replica

b) Write-ahead log (WAL) shipping

- The leader logs all physical data changes (re-writes of disk blocks, appends to segment files, etc.) to a WAL, writes them to disk and sends them to the followers
- Problem: data changes specify which bytes were changed in which block and are therefore specific to a certain technology and version (must be the same for the entire distributed system!)

Single-Leader Replication

Implementation of Replication Logs

c) Logical (row-based) log replication

- The leader logs all logical changes and sends these to the followers
  - INSERT: new values
  - DELETE: row, old values
  - UPDATE: row, field, old values, new values
  - For transactions: id, start, end
- Problem: takes additional memory when used together with (physical) WAL

	T1	T2	T3	UNDO/REDO
1			start	<START T3>
2			read(B, t)	
3			t‘ = t+1	
4	start			<START T1>
5	read(A, u)			
6			write(B, t‘)	<T3,B,t,t’>
7			commit	<COMMIT T3>
8		start		<START T2>
9		read(B, v)		
10		read(A, w)		
11		v‘ = v+w		
12		write(B, v‘)		<T2,B,v,v’>
13		commit		<COMMIT T2>
14	read(B, x)			
15	x‘ = x+1			
16	write(B, x‘)			<T1,B,x,x’>
17	u‘ = u+1			<START CKPT (T1)>
18	write(A, x)			<T1,A,u,x>
19	commit			<COMMIT T1>
				<END CKPT>

Implementation of Replication Logs

Hasso Plattner Institut

c) Logical (row-based) log replication

- UNDO logging
  - Log entry: old value
  - Write order: (1) Log (2) Data (3) Commit
  - Restore: Read log backwards and restore any uncommitted/aborted value.
- REDO logging
  - Log entry: new value
  - Write order: (1) Log (2) Commit (3) Data
  - Restore: Read log forwards and re-write any committed value.
- UNDO/REDO logging
  - Log entry: old & new value
  - Write order: (1) Log (2) Data & Commit
  - Restore: Redo all committed changes in chronological order and undo all uncommitted changes in inverse chronological order.

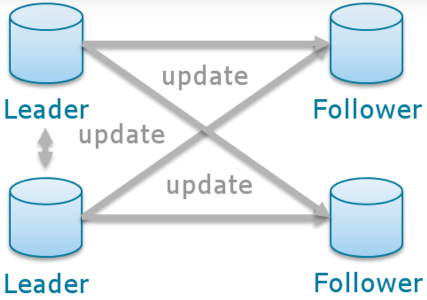
See lecture "Database systems II" by Prof. Naumann

Checkpointing
Allows to ignore successfully committed/aborted changes before that entry.

Replication

Single-Leader Replication

Multi-Leader Replication

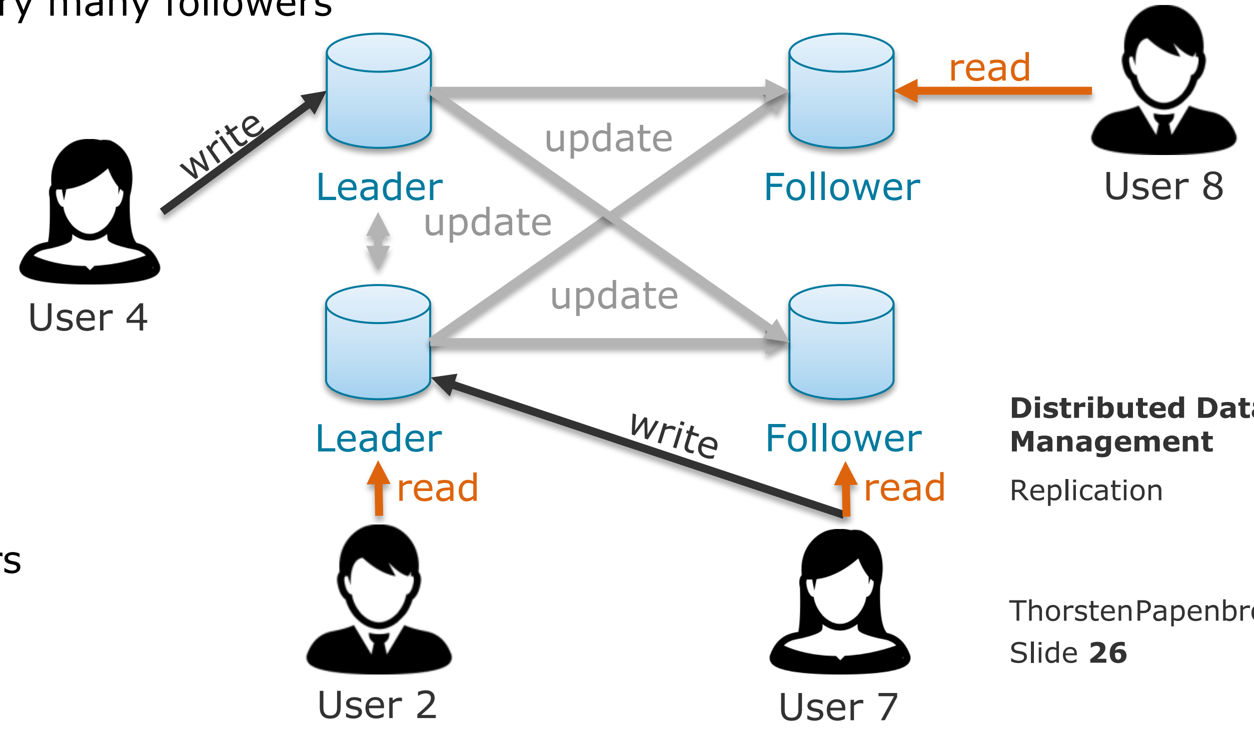


Leaderless Replication

Concept

Multi-Leader Replication

- Multiple leaders, arbitrary many followers
- Query processing like in single-leader setups
- Difference:
  - Write conflicts are possible
- Advantages:
  - Parallel writes
  - Leaders might die
  - Multiple datacenters



Conflict Resolution

Conflict

- Different leaders change the same item in different ways

Conflict Detection

- A change carries both new and old value
- A conflict occurred if the old value differs

table:	user
key:	1234
column:	picture_url
old_value:	**me-old.jpg**
new_value:	**me-new.jpg**
transaction:	328142

Conflict Resolution

- Inherently asynchronous, because both writes already succeeded
  - ➢ No chance to reject a conflicting write

a) Last write wins: always accept the write with the highest ID/timestamp/…

b) Merge: order the values (e.g. alphabetically) and store both concatenated

c) Application managed: write a conflict data structure and report it

See SVN, GIT, …

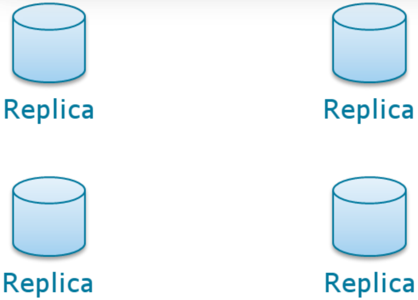
Overview

Replication

Single-Leader Replication

Multi-Leader Replication

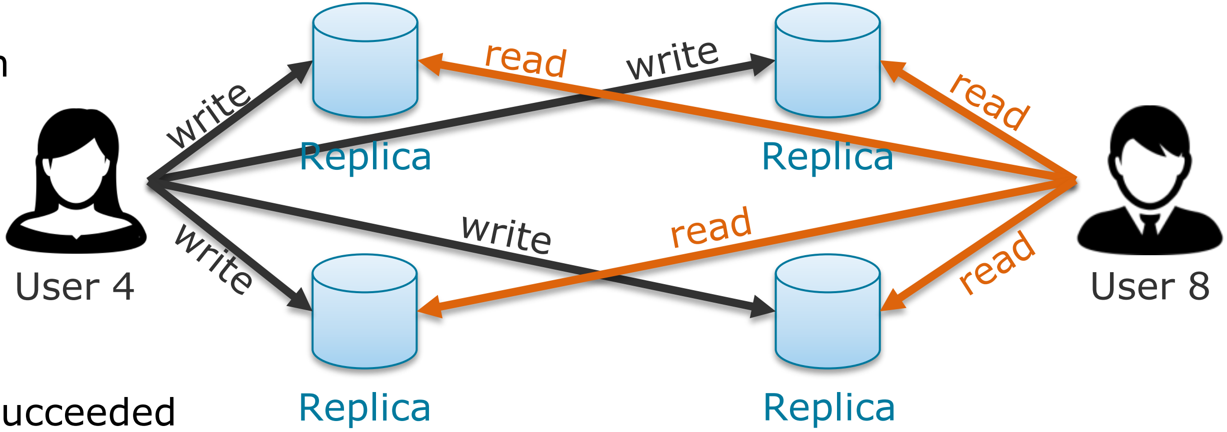
Leaderless Replication



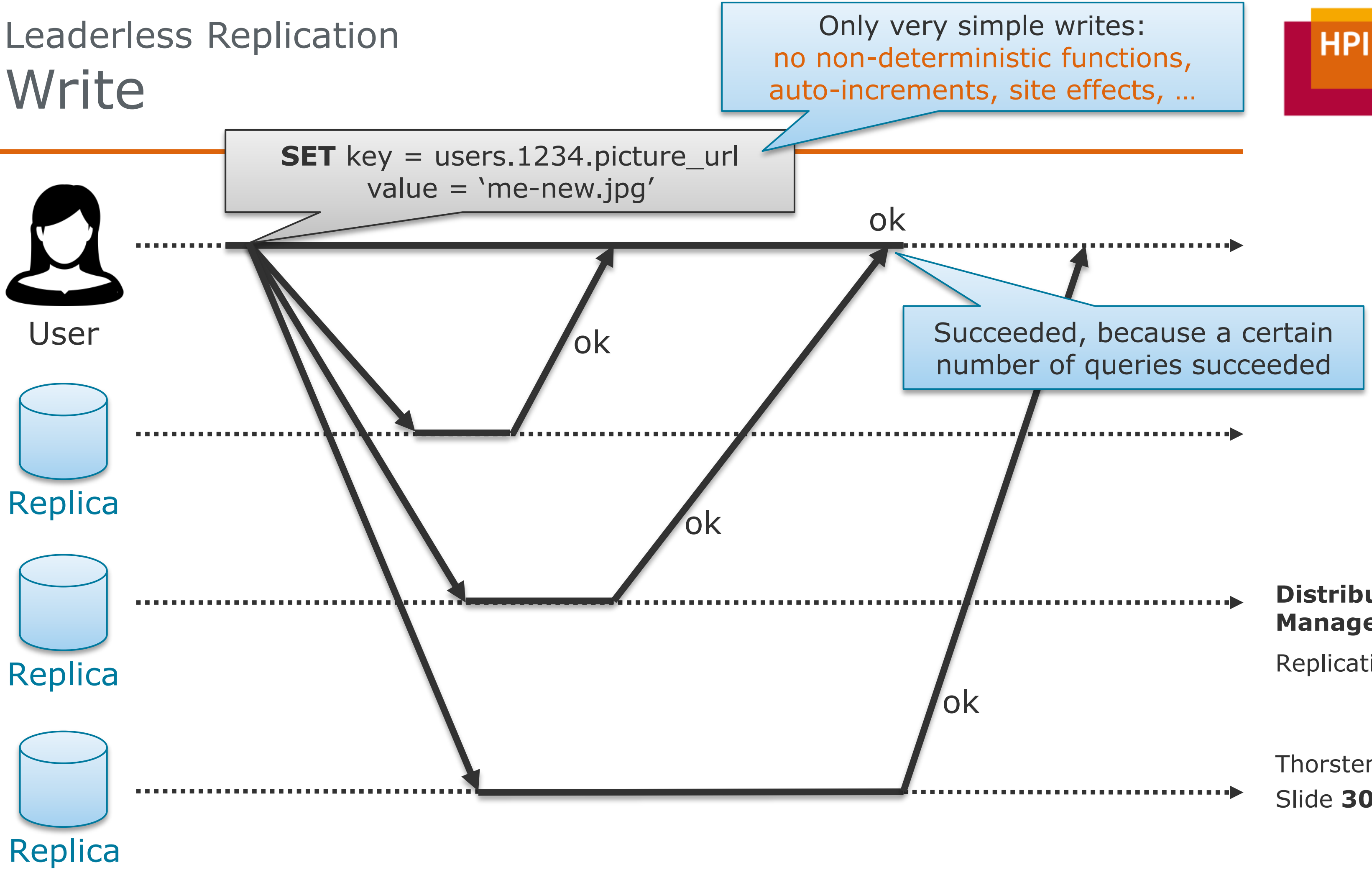
Concept

Leaderless Replication

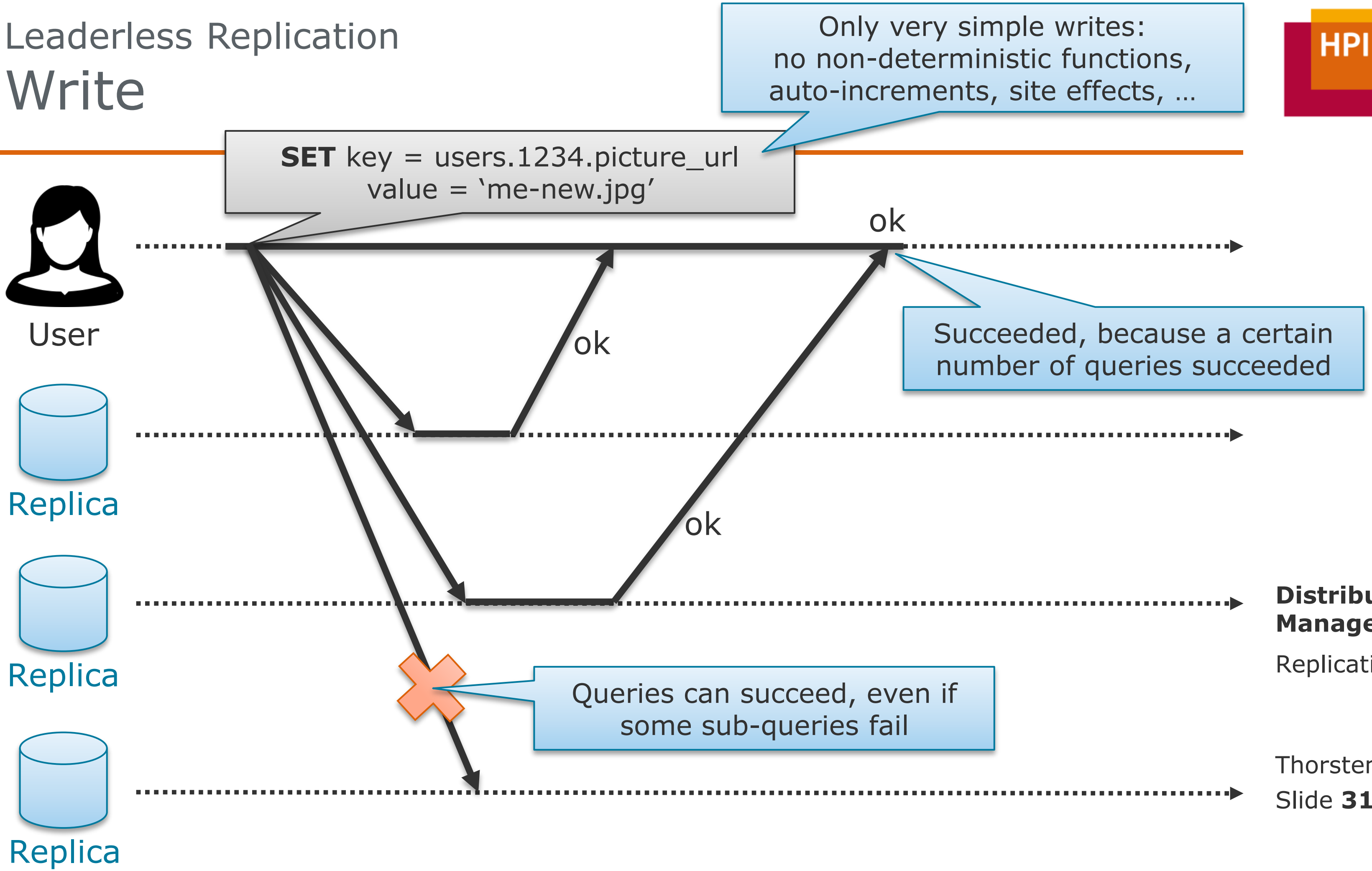
- No leader-follower distinction
  - All replica take writes
- Read and write queries are send to all replica:
  - If a certain number of queries succeeded, then the overall query succeeded
  - ➢ Tolerates some failing or slow replicas
  - ➢ No blocking change propagation by replica
- Advantages:
  - Parallel writes
  - No special roles for replica



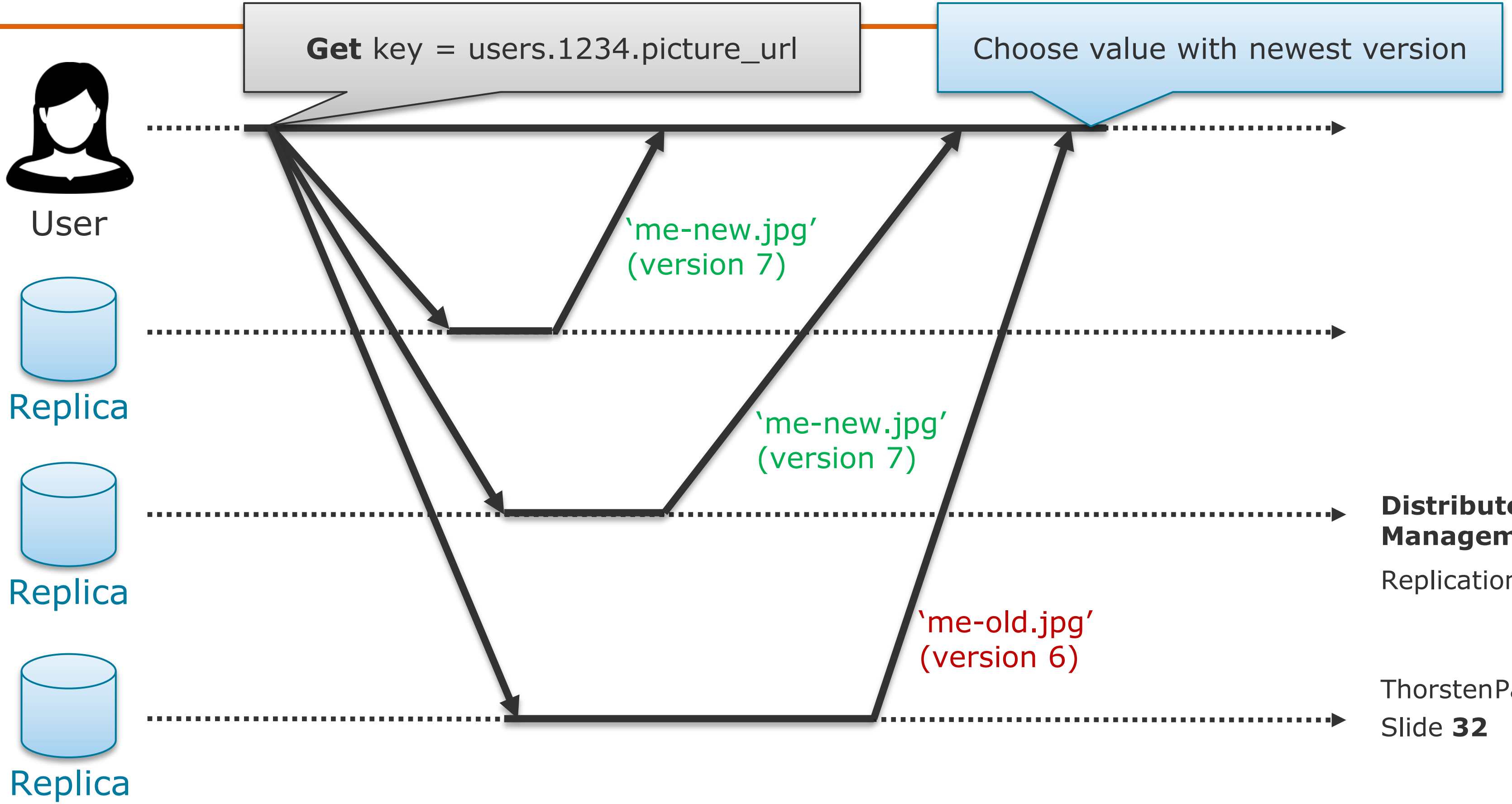
Write


Only very simple writes:
no non-deterministic functions,
auto-increments, site effects, …
SET key = users.1234.picture_url
value = ‘me-new.jpg’
ok
User
ok
Succeeded, because a certain
number of queries succeeded
Replica
ok
Replica
ok
Replica

Write


Only very simple writes:
no non-deterministic functions,
auto-increments, site effects, …
SET key = users.1234.picture_url
value = ‘me-new.jpg’
ok
User
ok
Succeeded, because a certain
number of queries succeeded
Replica
ok
Replica
Queries can succeed, even if
some sub-queries fail
Replica

Read


Get key = users.1234.picture_url
Choose value with newest version
User
Replica
Replica
Replica
'me-new.jpg'
(version 7)
'me-new.jpg'
(version 7)
'me-old.jpg'
(version 6)

Quorum Consistency

Quorum

- Given n nodes, the quorum (w,r) specifies …
  - the number of nodes w that must acknowledge a write and
  - the number of nodes r that must answer a query.

Quorum Consistency

- If $w + r > n$, then each query will contain the newest version of a value.
  - ➢ Identify the newest value by its version (not by majority!).
- The quorum variables are usually configurable:
  - ➢ Smaller r (faster reads) causes larger w (slower writes) and vise versa.
- The quorum tolerates …
  - $n - w$ unavailable nodes for writes.
  - $n - r$ unavailable nodes for reads.

Quorum Consistency

Quorum Changes

- Given a quorum (w,r), can we change it at runtime?
  - Increase w:
    - Yes, new values are written in a more reliable way.
  - Increase r:
    - Yes, existing values are read in a more reliable way.
  - Decrease w:
    - Yes, if $w + r > n$ still holds so that new values are read reliably.
  - Decrease r:
    - Yes, if $w + r > n$ still holds with the smallest w used to write any value in the database.

Quorum Consistency

Quorum Changes

- Given a quorum (w,r) and n-1 nodes (one has left the cluster), can we change the quorum?
  - Increase w:
    - Yes, new values are written in a more reliable way.
  - Increase r:
    - Yes, existing values are read in a more reliable way.
  - Decrease w:
    - Yes, if $w + r > n$ still holds with n being the current number of nodes.
  - Decrease r:
    - Yes, if $w + r > n$ still holds with n being all the nodes including the left node and the smallest w used to write any value in the database.

Pitfalls

Don't rely on time! (see later lecture)

Concurrent writes

- If write conflicts are resolved using timestamps, clock skew can cause older values to overwrite newer values (user clocks usually not in sync!)

Concurrent write and read

Don't expect strict consistency!

- If a read process interferes with a write process, the new values might be underrepresented

Apparently failed write

- If a write fails, it might still have silently succeeded on some nodes (only responses lost)

Failing node

- If a node with a new value recovers an old value after a crash, the quorum might be violated

Overall problem:
Loss of quorums and, hence, violation of consistency

What about BASE?

Eventual Consistency

Change Propagation Protocols

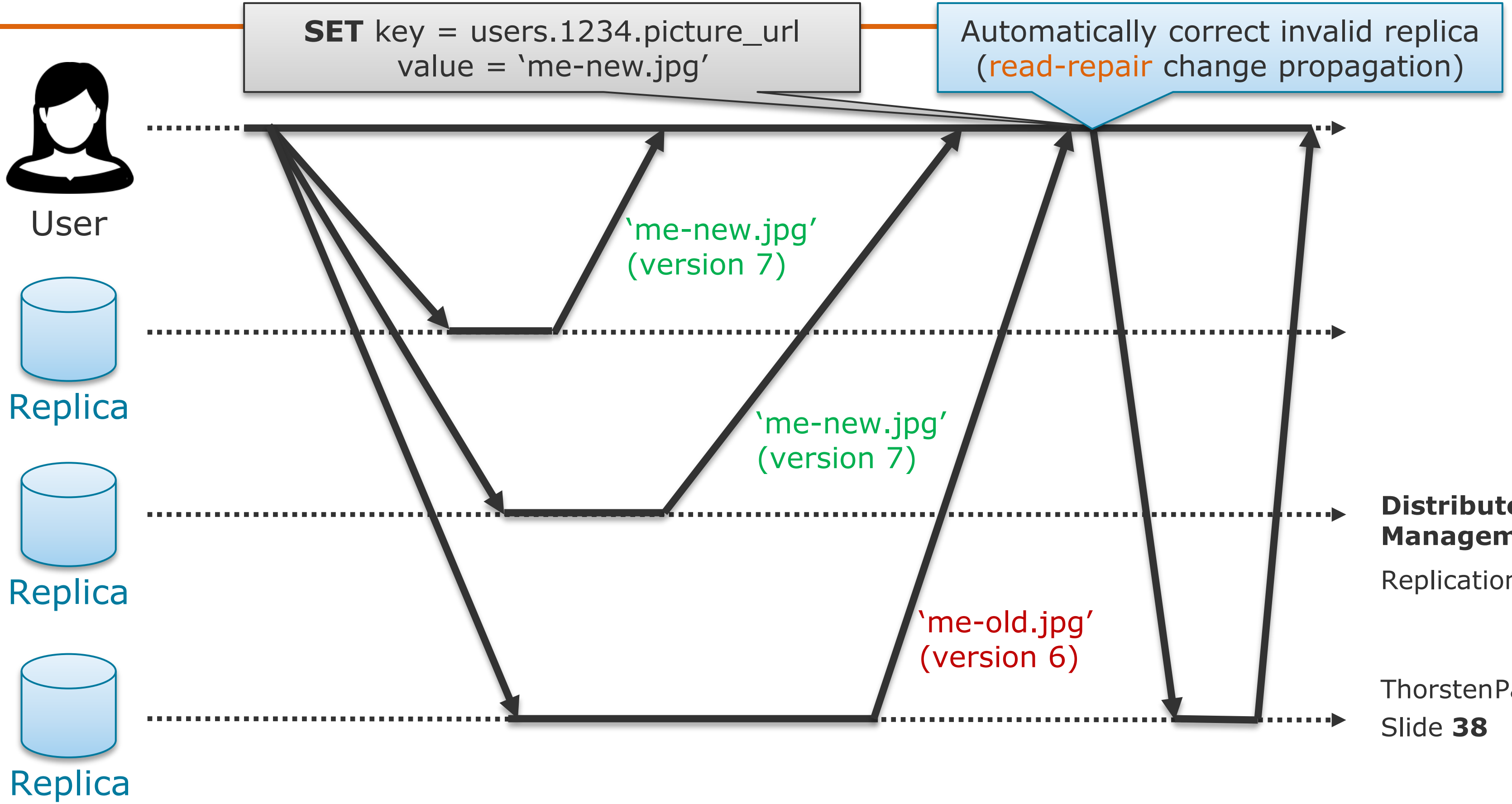
a) Read-Repair:

- Upon reading outdated values, **users** initiate value updates
- Passive change propagation

b) Gossip:

- All **replicas** run local agents that periodically match their states
- Active change propagation

Read-Repair



SET key = users.1234.picture_url
value = 'me-new.jpg'
Automatically correct invalid replica
(read-repair change propagation)
User
Replica
Replica
Replica
'me-new.jpg'
(version 7)
'me-new.jpg'
(version 7)
'me-old.jpg'
(version 6)

Gossip

Also known as
epidemic propagation

Gossip Protocol

- All replicas run local agents that periodically match their states
- Agent algorithm:
  - With a given (typically low) frequency:
    - Select a remote agent at random
    - Share any new changes since last contact
- Properties:
  - Robust spread of information tolerating node- and network-faults
  - Information converges with probability of 1
  - Information converges in logarithmic time in the number of agents
    - In each “round”, the number of agents with a particular change approximately doubles (ignoring redundant matches)

Exponentially rapid convergence!

Gossip

Gossip Protocol

- Example:
  - 100,000 replicas (= agents)
  - 3 sec gossip frequency
- What is the expected time for one change being known to all replicas?

round:	0																		
replicas:	1																		

Gossip

Gossip Protocol

- Example:
  - 100,000 replicas (= agents)
  - 3 sec gossip frequency
- What is the expected time for one change being known to all replicas?

round:	0	1	2																
replicas:	1	2	4																

Gossip

Gossip Protocol

- Example:
  - 100,000 replicas (= agents)
  - 3 sec gossip frequency
- What is the expected time for one change being known to all replicas?

round:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
replicas:	1	2	4	8	16	32	64	128	256	512	1k	2k	4k	8k	16k	32k	65k	131k

Rounds of gossip:

$2^{rounds} > replicas \Leftrightarrow rounds > log_2(replicas)$

Gossip

Gossip Protocol

- Example:
  - 100,000 replicas (= agents)
  - 3 sec gossip frequency
- What is the expected time for one change being known to all replicas?

round:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
replicas:	1	2	4	8	16	32	64	128	256	512	1k	2k	4k	8k	16k	32k	65k	131k

Rounds of gossip:

$$2^{rounds} > replicas \Leftrightarrow rounds > log_2(replicas) \Rightarrow rounds > log_2(100{,}000) \approx 16.61$$

Expected time to convergence:

$$time = rounds * frequency \Rightarrow time = 17 * 3\ sec = 51\ sec$$

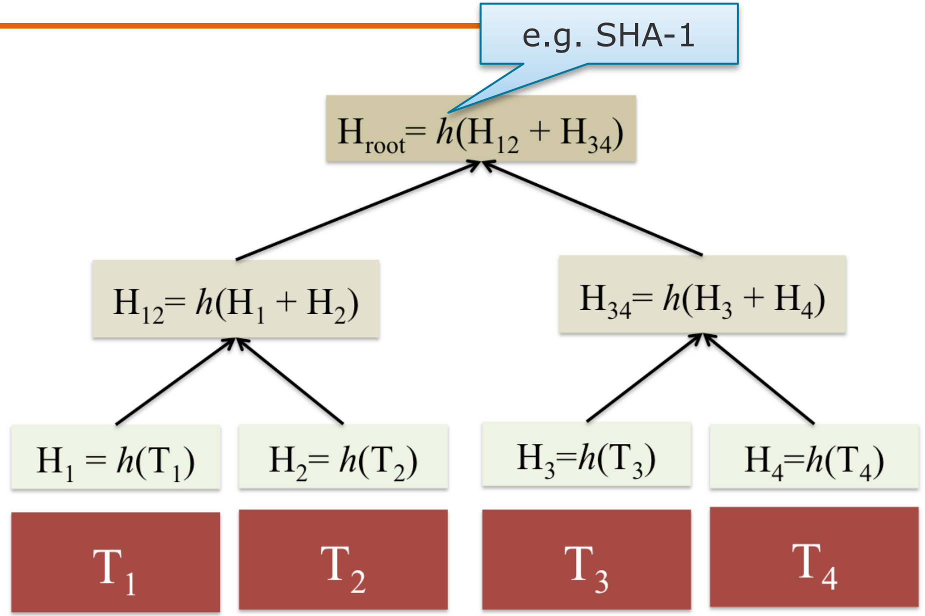
Gossip

Gossip Protocol

- General conditions:
  - Interactions happen periodically and pair-wise between random agents
    - ➢ To ultimately reach all agents!
  - Interactions change the state of at least one agent to reflect the state of the other
    - ➢ Change to the most recent version!
  - Interaction frequency is low compared to typical message latencies
    - ➢ Protocol costs are negligible!
  - Information exchange leads to redundancy due to the replication
    - ➢ Some updates are communicated to one agent multiple times!
  - Information exchanged during interactions is of bounded size
    - ➢ Not entire database!

How to quickly find only the changed areas?

Gossip

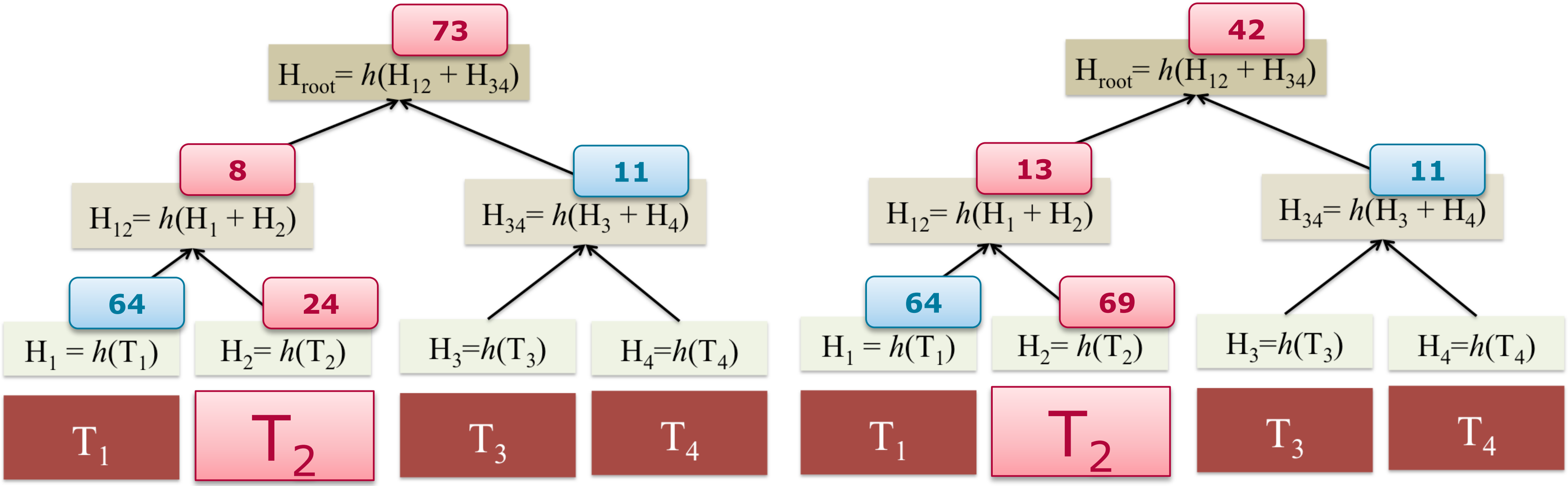


Merkle Trees

- Hash trees:
  - Leaves are hashes of the data
  - Inner nodes are hashes of child nodes
- Usually binary search trees, but higher degrees are possible
- Hashes identify same data, i.e., if two nodes in two trees have the same hash, then their underlying data is the same
  - ➢ Change identification algorithm:
    - Match Merkle Trees level-wise for differing hashes
    - Exchange data with differing hash paths
- Uses: Amazon Dynamo, Cassandra, and Riak

Exchanged data small and bound by tree height

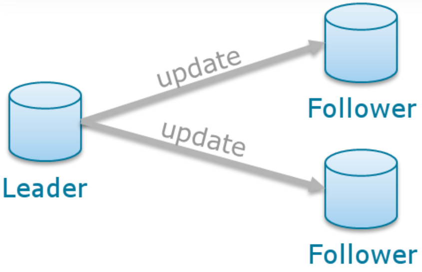
Gossip



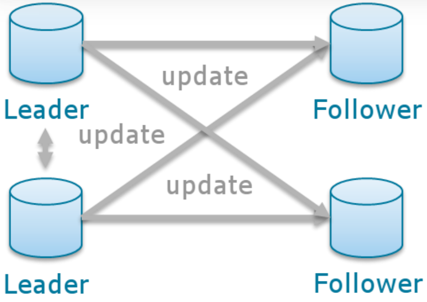
73
$H_{root} = h(H_{12} + H_{34})$
8
$H_{12} = h(H_1 + H_2)$
11
$H_{34} = h(H_3 + H_4)$
64
$H_1 = h(T_1)$
24
$H_2 = h(T_2)$
$H_3 = h(T_3)$
$H_4 = h(T_4)$
T_1
T_2
T_3
T_4
42
$H_{root} = h(H_{12} + H_{34})$
13
$H_{12} = h(H_1 + H_2)$
11
$H_{34} = h(H_3 + H_4)$
64
$H_1 = h(T_1)$
69
$H_2 = h(T_2)$
$H_3 = h(T_3)$
$H_4 = h(T_4)$
T_1
T_2
T_3
T_4

Replication

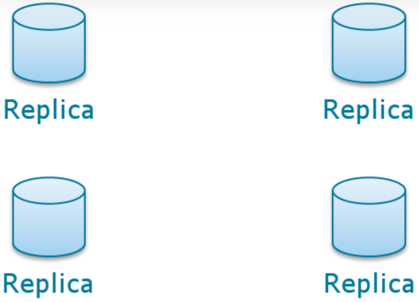
Single-Leader Replication



Multi-Leader Replication



Leaderless Replication



Check yourself

Consider a replication scenario with 3 replicas using quorum-consistency (as specified on slide 30). The last accepted write to a particular data item succeeded on 2 out of 3 nodes:

Replica 1	Replica 2	Replica 3
✓	✓	X

1. Which quorum configurations are possible if the quorum shall guarantee that queries read the newest version?
2. For each of those configurations list all combinations of unavailable nodes such that the next read query would still succeed.

Chapter 5. Replication