Replication Laboratory

Distributed Systems (DS)

Casamenti Gianmaria Pasini Luca

gianmaria.casamenti@studio.unibo.it
luca.pasini8@studio.unibo.it

Dipartimento di Informatica – Scienza e Ingegneria (DISI)
Alma Mater Studiorum—Università di Bologna

Academic Year 2022/2023
June 10, 2024

Next in Line...

Motivation & Lecture Goals

- Replication allows companies such as Amazon, Dropbox, Google, and Netflix to move data close to their users, significantly improving non-functional requirements such as latency and reliability.
- To provide a seamless experience to their users, distributed systems often rely on data replication.
- For all those reasons, in this lecture we will use docker and java to simulate and learn replication.

Git Lab Repository

- You can find all the code you need for this Laboratory in **Code Lab Repository**.
- It will be possible to download it to your computer, we recommend you do it with the Git command.
- Using the provided code during the lecture will be critical for complete understanding of the concepts.

Introduction

- In computer science, replication refers to the technique of exactly reproducing certain data.

Why choose replication?

Achieve fault tolerance:

- Deal with failures.
- Deal with incorrect behaviors through redundancy.

Why choose replication?

Increase availability:

- Data may be available only intermittently in a mobile setting, for example a local replica in the mobile node can provide support for disconnected operations.
- Some data might become unavailable due to excessive load, for example release of a new operating system.

Why choose replication?

Improve performance:

- Sharing of workload to increase the throughput (the rate of item delivery over a communication channel) of served requests and reduce the latency for individual requests. **Examples**: replicate a Web server to sustain a higher number of users and reduce queuing effects for individual users.

- Replicate data close to the users to reduce the latency for individual request. **Examples**: cache in processors, local copy on mobile devices and content-delivery networks.

Not only qualities

A big limit is the **consistency** across replicas, changing a replica demands changes to all the others.

- What happens if multiple replicas are updated concurrently:
  - Write-write conflicts / read-write conflicts
  - What is the behavior in the case of conflicts?

Not only qualities

Latency does not improve over time as other performance metrics

- As other metrics improve, latency may easily become the limiting factor for performance

	1983	2022	Improvement
CPU speed	1x10Mhz	8x3.2 Ghz	> 2000x
Memory size	<= 2MB	32 GB	> 16000x
Disk capacity	<= 30 MB	4 TB	> 100000x
Network bandwidth	3 Mbps	> 10 Gbps	> 3000x
Latency (RTT)	2.54 ms	0.1 ms	< 30x

Not only qualities

Scalability vs **Performance**

- Replication may degrade performance!
- The cost to ensure that data is consistent across replicas can be very high

For example you have to wait until a write has been successfully propagated to all replicas before making any progress.

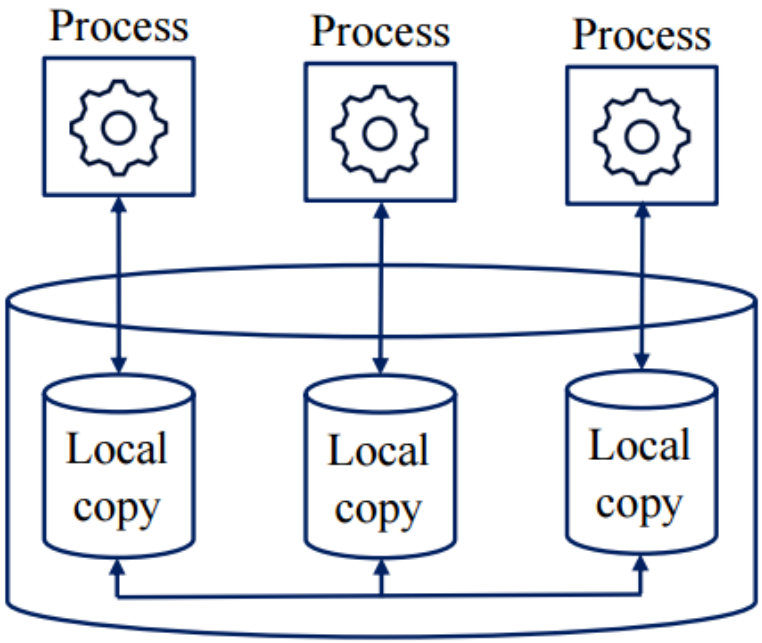
Where we find replication

- **Distributed file systems**
- **Content delivery networks**
  - Geographically distributed networks that replicate the content to better serve end users.
- **Collaboration platforms**
  - Files may be replicated in users' devices.
  - Support for disconnected operations.
  - Documents / files are reconciled upon reconnect.

Consistency models

- Focus on a **distributed** data store:
  - Shared memory, filesystem, database, ...
  - The store consists of multiple items like files or variables.

- Ideally, a read should show the result of the last write.



Distributed data store

Consistency protocols

- We first overview the main implementation strategies used in consistency protocols ...
  - **Single leader**
  - **Multi leader**
  - **Leaderless**

Single Leader Protocols

- **One of the replica is designed as the leader:**
  - When clients want to write to the datastore, they must send the request to the leader, which first writes the new data to its local storage.
- **The other replicas are known as followers:**
  - Whenever the leader writes new data to its local storage, it also sends the data to all its followers.

Single Leader Protocols

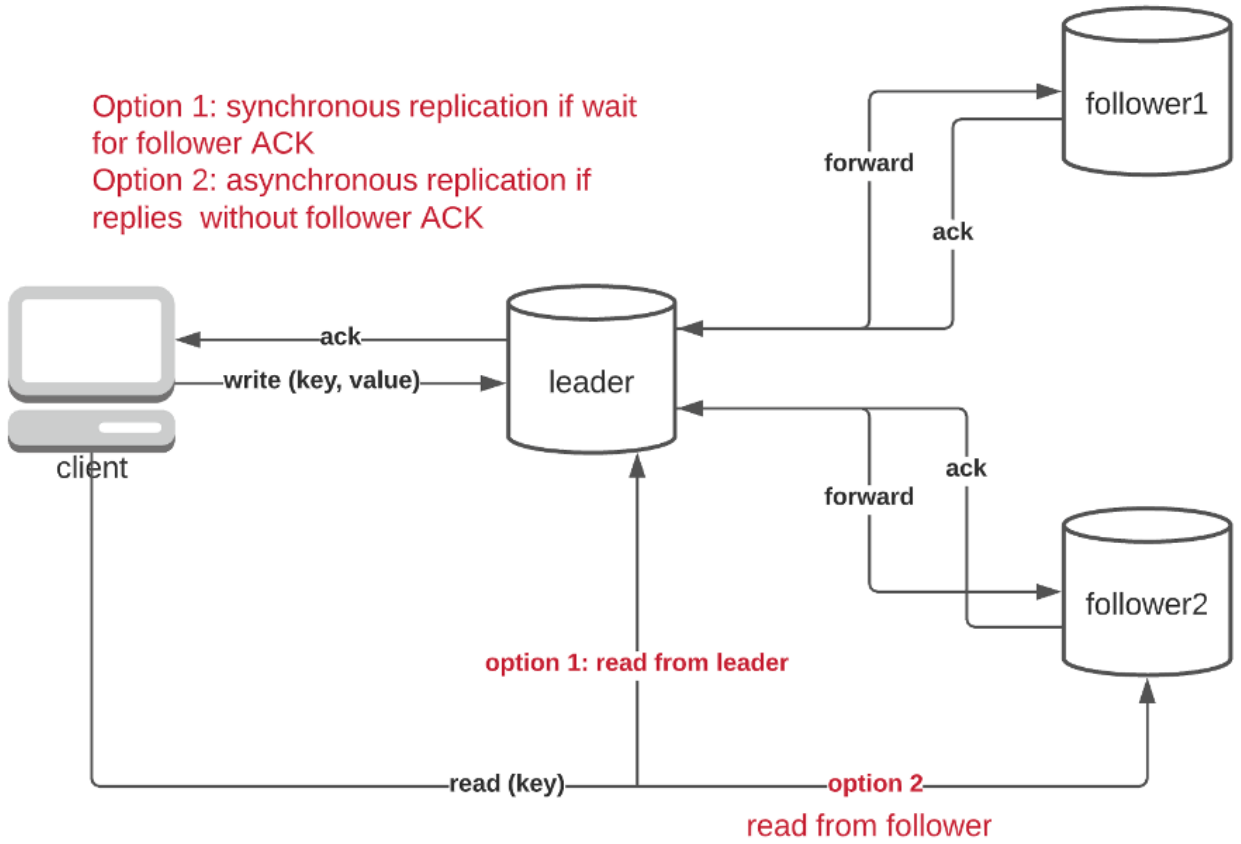


Figure: Single leader architecture

Single Leader Protocols

- **When a client wants to read data:**
  - In some systems, it queries the leader(followers only used as a ”backup”).
  - In other systems, it can query any replica.

Single Leader Protocols

- **Synchronous**
  - The write operation completes after the leader has received a reply from all the followers.
- **Asynchronous**
  - The write operation completes when the new value is stored on the leader.
  - Followers are updated asynchronously.
- **Semi-synchronous**
  - The write operation completes when the leader has received a reply from at least k replicas.
  - k is a configuration parameter in many replicated databases (e.g. Cassandra).

Single Leader Protocols

- **Synchronous replication is safer** (or semi-synchronous with k followers)
  - Even if k-1 replicas fail, we still have a copy of the data
  - Followers can recover from failures by asking updates to other replicas (catch-up recovery)

- **What happens if the leader fails?**
  - We can elect a new leader (failover)
  - Many tricky situations depending on the specific assumptions(Network may be partionized)
  - To ensure safety, we need some consensus protocol

Single Leader Protocols

- **No write-write conflicts possible**
  - The leader receives all write operations and determines their order
- **Read-write conflicts still possible**
  - Depending on the specific implementation
  - E.g., a client reads from an asynchronous replica and does not see writes it previously performed on the leader

Multi Leader Protocols

- Writes are carried out at different replicas concurrently
- **No single leader** means that there is no single entity that decides the order of writes
- It is possible to have **write-write conflicts** in which two clients update the same value almost concurrently
  - How to solve conflicts depends on the specific consistency model

Multi Leader Protocols

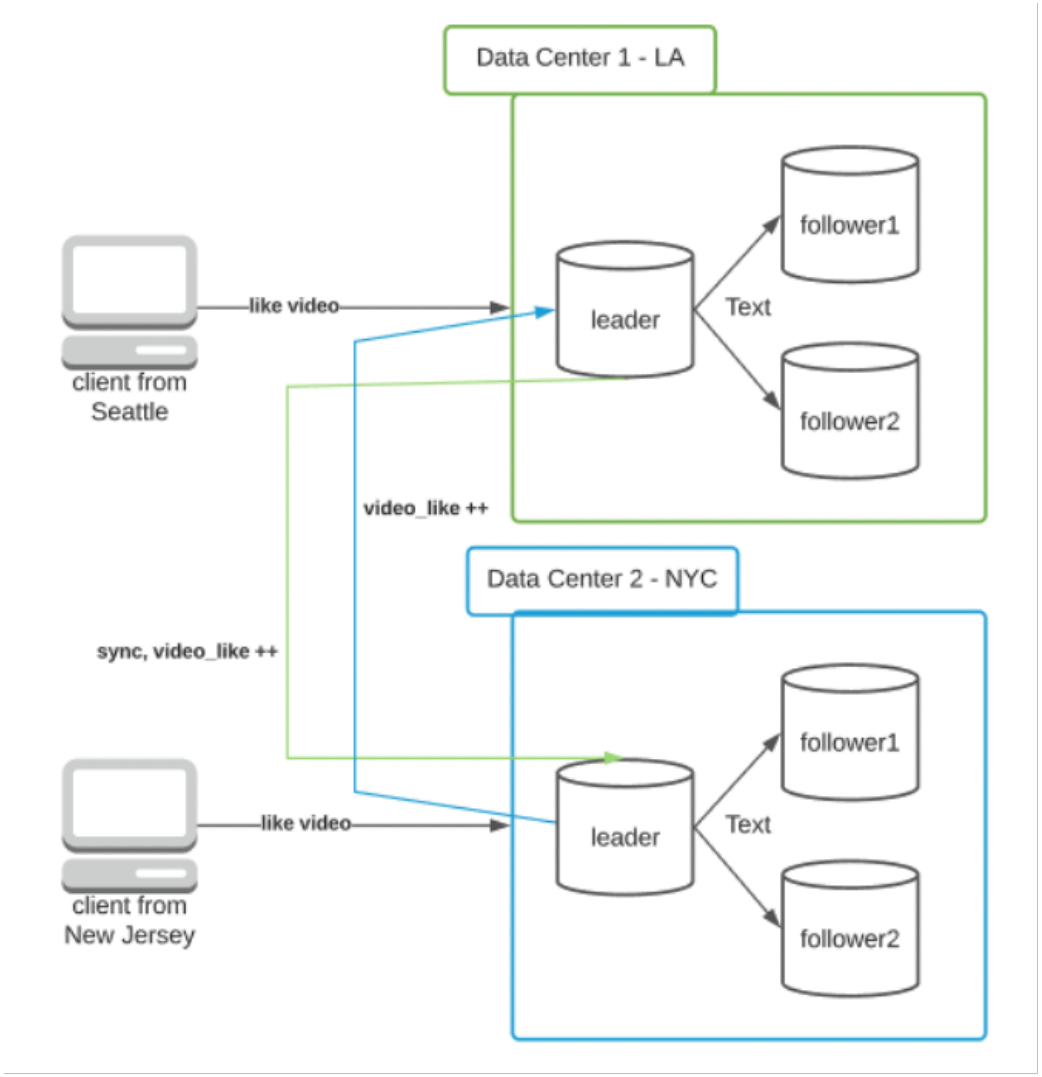


Figure: Multi leader architecture

Multi Leader Protocols

- In a multi-leader system, there are multiple nodes that act as leaders, each capable of accepting write operations. These leaders can operate independently or in cooperation, depending on the specific architecture of the system

- **Leader Replicas** not all replicas need to be leaders.
  Some replicas may act solely as followers or backups, receiving updates from leaders and primarily serving read operations

Multi Leader Protocols

- **Scalability** Distributing leader roles among different replicas allows the system to scale better, handling a larger number of write operations and improving load distribution
- **Fault Tolerance**, having multiple leaders enhances fault tolerance, as the failure of one leader does not compromise the entire system. Operations can be redirected to other leaders
- **Flexibility** provides greater flexibility in data and operation management, as different data partitions can be managed in parallel by different leaders

Multi Leader Protocols

- Multi leader protocols are often adopted in geo-replicated settings
  - Contacting a leader that is not physically co-located can introduce prohibitive costs
- In general, more difficult to handle …
  - Simultaneous writes can create conflicts
- … but in practice, conflicts are rare and easy to solve in several application scenarios
  - E.g., social network
    * Writes (posts, comments) are not frequent
    * Writes for one user / group of users often performed on the same replica
    * Conflicts are not critical: concurrent comments can be stored in different orders on different replicas

Multi leader protocols

- Multi leader protocols **natively** supported in some database
  - In addition to single leader protocols ì
    * Single leader protocols within a data center
    * Multi leader protocols across data centers

- **Nowadays** we can find this architecture in:
  - **Distributed databases** like Cassandra, each node can accept write operations for a portion of the data, acting as a leader for that specific data. However, not all nodes are necessarily leaders for all operations
  - **Distributed file systems**, some nodes may be designated as leaders to manage specific data blocks or directories, while other nodes act as passive or backup replicas

Leaderless protocols

- In single leader and in multi leader protocols, clients send writes to a single node
- In leaderless replication, the client contacts multiple replicas to perform the writes/reads
  - In some implementations, a coordinator forwards operations to replicas on behalf of the client
- Leaderless replication uses quorum-based protocols to avoid conflicts
  - Similar to a voting system
  - We need a majority of replicas to agree on the write
  - we need an agreement on the value to be read
- Leaderless replication used in some modern key-value / columnar stores
  E.g., Amazon Dynamo, Riak, Cassandra

Data-centric consistency models

- We now review the most widely used data-centric consistency models
- Ideally, we would like all operations to
  - Take place instantaneously at some point in time
  - Be globally ordered according to their time of occurrence
- This is not possible in a distributed system
  - There is no single clock available
- Even without considering time, we need to implement (expensive) coordination protocols to ensure global order

Sequential consistency

*"The result is the same as if the operations by all processes were executed in some **sequential order**, and the operations by each process appear in this sequence in the order specified by its program"*

Sequential consistency

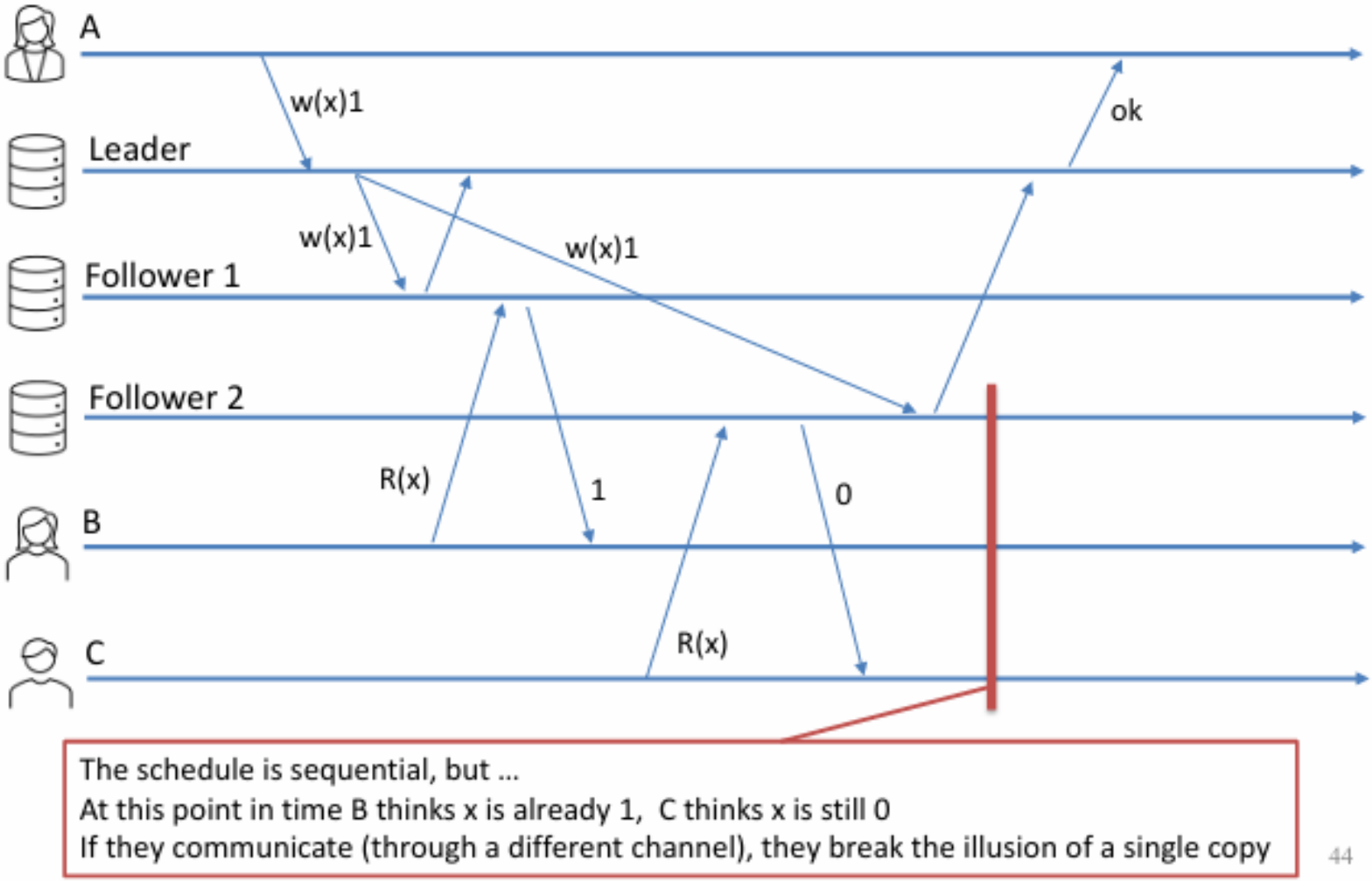


Figure: Sequential consistency senario

Sequential consistency

- Sequential consistency limits **availability**
  - No highly available implementations are possible
- Single leader protocols
  - Client needs to contact the leader
  - Leader must propagate the update to the replicas
    * In a synchronous way if we want to guarantee sequential consistency!
- Two main problems related to availability
  - **High latency** due to synchronous interactions
  - Clients are blocked in the case of **network partitions**

Linearizability

"Each operation should appear to take effect instantaneously at some moment between its start and its completion"

Linearizability

- Also known as strong / external / atomic consistency
- **Strongest possible consistency** guarantee in presence of replication
- Linearizability is a recency guarantee
  - When a client completes a write on the data-store, all clients need to see the effect of the write
  - This gives the illusion of having a single copy of the data store

Linearizability

- Why is sequential consistency not enough?
  - Sequential consistency does not include a notion of **time**
  - Linearizability includes a notion of time: operations behave as if they took place at a single point of (wall clock) time
  - It may require time to propagate an operation to all replicas, so the operation may not be instantaneously visible
  - Let's see our previous example again

Linearizability

- Linearizability is a **composable** property
  - If the operations on individual variables are linearizable …
  - … the global schedule is also linearizable

Linearizability

- Single leader replication may implement linearizability
  - The leader orders writes according to their timestamp
  - Replicas are updated synchronously and atomically
    * E.g., locking protocol to avoid reading while a write is in progress
  - Much more difficult to guarantee in the presence of failures
    * It is an agreement/consensus problem to determine if and how the network is partitioned and who is the leader

Linearizability

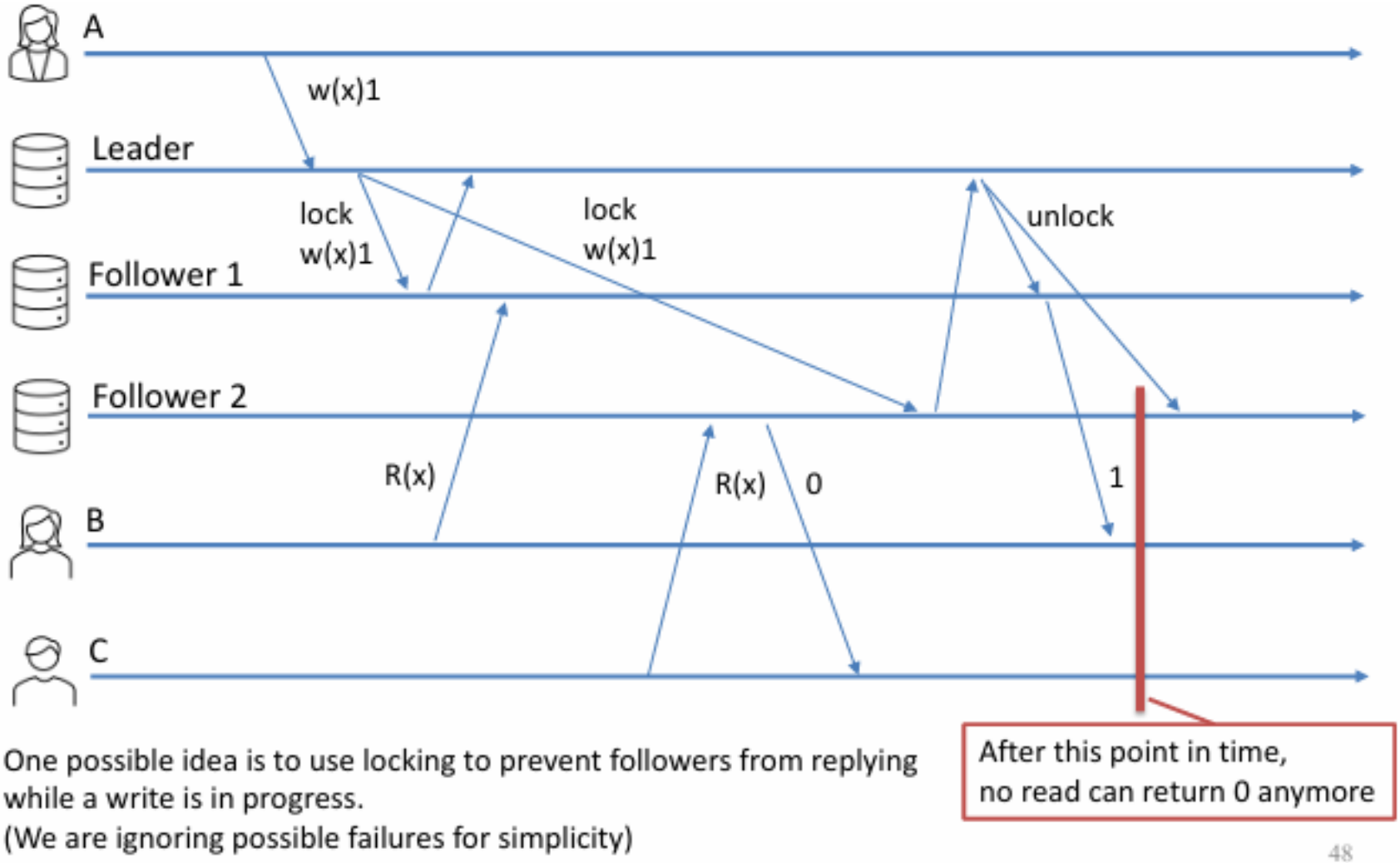


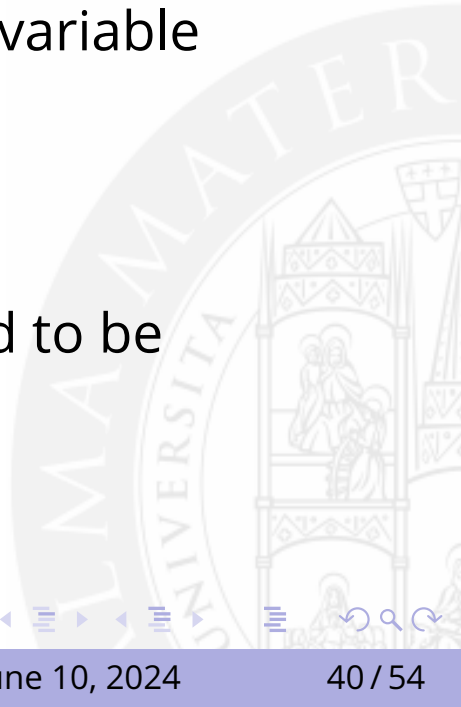
Figure: Linearizzation effect

Casual Consistency

"Writes that are potentially causally related must be seen by all processes in the same order. Concurrent writes may be seen in any order at different machines"

Casual Consistency

- Causal consistency defines a causal order among operations
- More precisely, causal order is defined as follows:
  - A write operation W by a process P is causally ordered after every previous operation O by the same process
    * Even if W and O are performed on different variables
  - A read operation by a process P on a variable x is causally ordered after a previous write by P on the same variable
  - Causal order is transitive
- It is not a total order
  - Operations that are not causally ordered are said to be concurrent



Why causal consistency

- Easier to guarantee within a distributed environment
  - Smaller overhead
- Easier to implement

Exercises Introduction

From this point we will discuss some exercises placed in **Code Lab Repository**

- Exercise 1– **Single leader java**
- Exercise 2–**Command line docker replicated DB**
- Exercise 3– **Docker-compose replicated DB**
- Exercise 4– **Mandatory exercise**

Exercise 1

- We can find Exercise 1 in the ”java replication” package
- It is a Gradle project with some example as single-leader or multi-leader protocol of replication in java.

Single-leader in java

Inside single leader package we can find a replicated database composed of 3 parts:

- **Client**: the client sends requests to the leader or follower
  - He sends READ requests to both leader and follower
  - He sends WRITE requests to the leader
- **Leader**: the leader manages the requests
  - When he receives a WRITE it writes to its database and also propagates this information to its followers
  - When he receives a READ it responds with the correct data if found
- **Follower**: The Follower manages WRITE and REPLICATE requests
  - When he receives a REPLICATE from the leader he writes the replicated data in his database.
  - When he receives a READ it responds with the correct data if found

Single-leader in java

```
1  if (parts[0].equals("WRITE")) {
2      String[] kv = parts[1].split(" ", 2);
3      synchronized (dataStore) {
4          dataStore.put(kv[0], kv[1]);
5      }
6      replicateToFollowers(parts[1]);
7      out.println("WRITE successful");
8  } else if (parts[0].equals("READ")) {
9      String value;
10     synchronized (dataStore) {
11     value = dataStore.get(parts[1]);
12     }
13     out.println(value != null ? value : "Key not found");
14 } else {
15     out.println("Invalid request");
16 }
```

Exercise 2

- We can find Exercise 2 in the ”command-line replication” package
- In this first exercise we can see how to create a replicated database using only the command line
- It is explained in detail in the README file

Command-line replication

Exercise 1 -- Creating a MongoDB Replica Set Using Docker

In this exercise we will create a very simple replica set without docker-compose (next exercise).

Installation

First, we need to install Docker on our system. If you're using your personal computer, you can install Docker desktop.

To verify that you have docker installed run :

```
docker -v
```

This should output the version number.

Next, we need to make sure our docker daemon is running. If you run the command:

```
docker image
```

You should be able to see the list of images you currently have on your system.

Next, we will get the latest version of the official Mongo image, by running:

```
docker pull mongo
```

Command-line replication

- In this exercise we can find all the stuff that we need to create a functioning replicated database using docker
- allows us to review the basic docker commands seen in the previous lessons like:
  - docker run
  - docker image
- and more complex one

Exercise 3

- We can find Exercise 3 in the ”docker compose replication” package
- It is instantiated with MongoDB, and managed by Docker-compose
- With ”docker compose up” command being inside the package we will run it up.

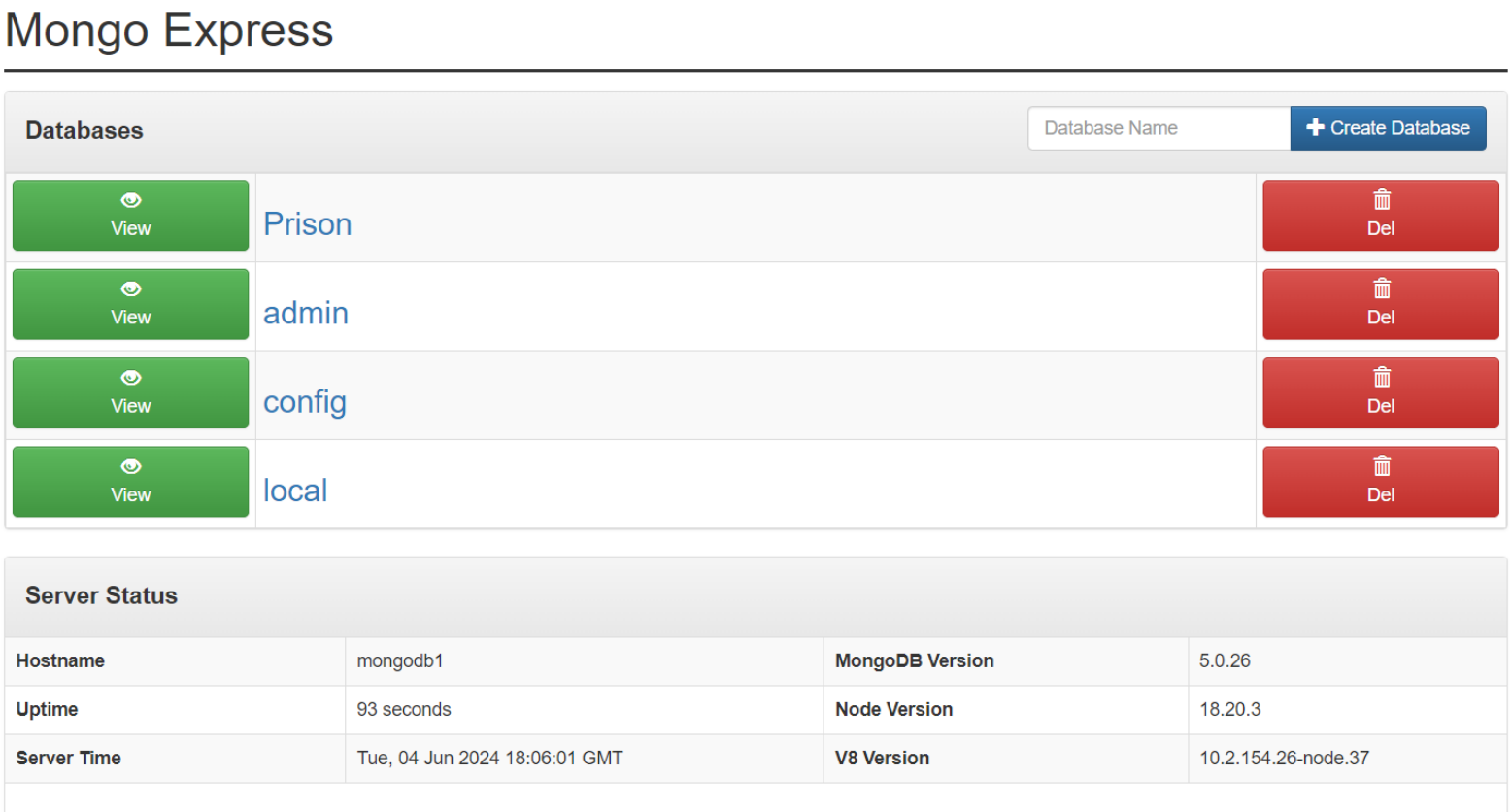
Docker-compose replication

- In this exercise we have one leader database and two followers, that have the same data but one of them goes on only if the leader stops working.
- Here we use majority as quorum so if there will be more than (N - (N/2 + 1)) failures the system would stop working (N = leader + followers)

Docker-compose replication

We can interact with it in two different localhost address:

- 8081, will appear the DB after being logged in with username(admin) and password(pass)



Prison is a working db previusly istantiated with tables and records

Docker-compose replication

- 3000, is a web page that utilize Javascript, we can use it for sending command to the DB by URL:
  - Test, if the database is working you get all the records of the db printed on the screen
  - Count, writes the number of records of the database

Docker-compose replication

- This exercise shows us the resilience that replication provides to the system
- By typing the command **”docker stop mongodb1”** we will simulate a failure on the leading database
- However, this failure will not be perceived by the user because one of the two followers will immediately take his place
- With N=3 another failure would not be supported by the system that would crash

Mandatory Exercise

- Mandatory exercise
  - This exercise is mandatory
  - Its submission and understanding will be certainly checked before the final exam

- Activity

  you have seen how to create a single leader replicated database, you have all the knowledge necessary to develop a multi leader database on your own