J. J. Strossmayer University of Osijek

Faculty of Electrical Engineering, Computing and Information Technologies Osijek
Kneza Trpimira 2b
HR-31000 Osijek
www.ferit.unios.hr

Lab Assignment 4:

Data Replication in Distributed Computing Systems

Academic year 2025/2026.

Luka Loina, univ. mag. ing. comp.

Distributed Computing Systems

Lab Assignment 4

How 43 Seconds Broke a Day: GitHub's Replication Lesson

In October 2018, GitHub suffered more than 24 hours of degraded service because of a subtle failure in its database replication setup. During routine maintenance, a 43-second network partition between data centers caused the automated failover system to promote a replica in another region as the new primary while some recent writes had not yet replicated across.

When connectivity was restored, different MySQL clusters each contained valid but conflicting writes, and there was no safe, automatic way to merge them. To avoid corrupting user data, GitHub effectively put parts of the site into read-only mode, accepted that some data would look stale or inconsistent, and spent many hours restoring from backups and carefully reconciling missing transactions.

This incident showed that data replication is not just about having copies of data, but about understanding replication lag, leader promotion, and consistency guarantees—exactly the trade-offs you will explore in this lab.

Exercise 4: Data Replication in Distributed Computing Systems

1 Introduction

Data replication is one of the most fundamental and consequential techniques in modern distributed computing. At its core, replication is the process of maintaining multiple identical copies (called replicas) of data across different machines (nodes) connected via a network. This seemingly simple concept has become essential to every system that operates at scale. Replication strategy directly determines whether a system can serve users reliably, respond quickly, and continue functioning when infrastructure fails.

This lab explores two fundamental approaches to handling data replication in distributed computer systems during node and network failures, keeping consistency of data while sacrificing availability of the system, and sacrificing consistency of data across nodes while keeping availability of services.

As previously mentioned, in this lab you will explore two trade-offs in data replication during infrastructure failure:

- Keeping data consistency – in some systems (like financial systems, inventory systems and similar) having different data between nodes is worse than having no data. In such cases, keeping strong consistency between nodes is required.
- Keeping data availability – in some systems (like social media feeds, caching layers and similar) users can tolerate temporarily different data across nodes as long as requests are served in a timely manner. In such cases, keeping data available is more beneficial than keeping data strictly consistent.

In the exercises, you will implement both ways for handling infrastructure failure in data replication systems and test their behavior. While doing exercise keep in mind that in many real systems we should not make single uniform choice about consistency of data. Different subsystems of larger application might need different trade-offs. For instance online marketplace might require consistency for checkout and billing subsystems but availability for product recommendations and user interaction logging subsystems.

1.1 Learning Objectives

By the end of this lab, you will be able to:

- Understand the different trade-offs in data replication in distributed computer systems.
- Implement two basic approaches to handling data replication in the face of infrastructure failures.
- Explain the CAP theorem and apply it to systems design.

2 Background

The motivation for data replication stems from three primary imperatives that modern distributed systems must address. First, replication improves **fault tolerance and high availability**: when stored on a single machine, data is vulnerable to hardware failure, network outages, or power loss. By distributing copies across multiple nodes, the system remains operational even if one or more nodes fail. A single replica creates a dangerous single point of failure; multiple replicas ensure the system gracefully degrades rather than failing completely.

Second, replication enables **improved performance and reduced latency**. Geographically distributed systems can place replicas close to end users, leveraging the fundamental constraint imposed by physics - the speed of light. A user accessing data from a server located on their continent experiences significantly lower latency than accessing a distant data center. Beyond geography, replication distributes read requests across multiple nodes, preventing any single server from becoming a bottleneck.

Third, replication **increases scalability**. As systems grow to serve millions of concurrent users, a single database node becomes insufficient. By replicating data across many nodes, the system scales horizontally - adding more hardware to distribute the load proportionally. This is fundamentally different from vertical scaling (adding more resources to a single machine), which has physical and economic limits.

2.1 CAP theorem

The CAP theorem represents one of the most significant intellectual contributions to distributed systems design. Formulated by Eric Brewer in 2000 and formally proven by Seth Gilbert and Nancy Lynch in 2002, the theorem provides a rigorous framework for understanding the impossible choices that architects must make when designing systems spanning multiple nodes across unreliable networks. It states that any distributed system can guarantee at most two of three properties: **Consistency**, **Availability**, and **Partition Tolerance**. In essence, the CAP theorem does not merely offer strategic guidance; it establishes a mathematical impossibility boundary.

Consistency in the CAP framework refers specifically to atomic (or linearizable) semantics: every read operation must return either the most recent write or an error, and all clients must see the same data at the same time, regardless of which node they query. When a write operation completes and the client receives acknowledgment, any subsequent read must reflect that write. This is fundamentally different from ACID consistency, which deals with transaction isolation and data integrity rules. In CAP's strict definition, consistency means the system behaves as if all operations execute on a single, centralized node in some order consistent with the actual temporal ordering of client requests. Violations are absolute: either a system maintains this order-respecting property or it does not.

Availability means that every request sent to a non-failing node must eventually receive a response. The system cannot timeout, delay indefinitely, or silently ignore valid requests. A node that has not crashed must answer. The response need not contain the most current data, and a reasonable response could even be an error message—what matters is that some response is guaranteed. This is notably different from "high availability" in software architecture terminology, which usually refers to system uptime and redundancy. CAP's availability is more absolute: a single request to a working node cannot go unanswered.

Partition Tolerance means the system must continue operating despite arbitrary network partitions—scenarios where messages between nodes are lost or delayed indefinitely. A partition is not a clean break between two halves of a network. Rather, it encompasses any scenario where communication between groups of nodes becomes unreliable: some messages may arrive, others may not, and the system cannot distinguish a crashed node from one that is merely slow to respond. A partition-tolerant system must handle these ambiguities and continue making progress.

The CAP theorem's power derives from its proof, which demonstrates this is not merely a design limitation but a mathematical impossibility. The proof unfolds through a simple but devastating scenario. Consider a minimal distributed system with two servers, G_1 and G_2, both maintaining a shared variable initially set to v_0. A client writes v_1 to G_1, and G_1 acknowledges success. Due to a network partition, this write cannot be communicated to G_2. The client then reads from G_2, which still holds the old value v_0.

At this moment, the system faces an inescapable choice. If the system is to remain **available**, G_2 must respond to the client's read request immediately. But G_2 cannot distinguish between two possibilities: either the network is partitioned and G_1 truly has the new value, or G_1 crashed and v_0 is the latest value. Responding with v_0 (for availability) violates consistency; the client just wrote v_1 elsewhere. Refusing to respond (to maintain consistency) violates availability.

This dilemma is not a bug in system design; it is structural. The formal proof shows that under the assumptions of asynchronous distributed systems with unreliable networks, **no algorithm exists that can simultaneously guarantee consistency, availability, and partition tolerance**. The impossibility is absolute, though Gilbert and Lynch's later work clarified crucial nuances: the system designer is not forced to abandon all three, but rather must carefully choose which property to weaken when a partition occurs.

2.2 Beyond CAP: The PACELC Theorem

In 2010, Daniel Abadi introduced the PACELC theorem, which extends CAP's reasoning. PACELC states: if a **P**artition occurs, the system must trade **A** (availability) for **C** (consistency); **E**lse (in normal operation), it must trade **L** (latency) for **C** (consistency). This theorem recognizes that CAP, while profound, ignored an equally important trade-off: even without partitions, consistent replication incurs latency costs. Synchronous replication ensures freshness but blocks on network delays. Asynchronous replication is fast but increases consistency lag. PACELC forces designers to articulate both partition-time and normal-time trade-offs explicitly.

3 Comparison and Experiments

In this section, you will design and perform experiments, collect data, and analyze the results to better understand trade-offs of different approaches to data replication.

3.1 Goal

Your objective is to:

- Extend base key-value store to implement different approaches to data replication.
- Measure response time on client process.
- Develop intuition for when each algorithm might be preferable in real distributed systems.

3.2 Student Tasks

Assignment 1. **Implement CP (Consistency + Partition Tolerance) system for key-value store**

1. Start with base key-value system.
2. Extend system to provide consistency and partition tolerance.
3. Run 1 leader and at least 1 follower process.
4. Measure response time for client requests. Compare response times between system when all nodes are running and when one or more follower nodes are not responding.
5. How does the response time of the CP system scale with respect to the number of nodes in the system?

Assignment 2. **Implement AP (Availability + Partition Tolerance) system for key-value store with eventual consistency**

1. Start with base key-value system.

2. Extend system to provide availability and partition tolerance with eventual consistency after connectivity of system is restored.
3. Run 1 leader and at least 1 follower process.
4. Measure response time for client requests. Compare response times between system when all nodes are running and when one or more follower nodes are not responding.
5. How does the response time of the AP system scale with respect to the number of nodes in the system?
6. Compare the measured response times between your implementations of CP and AP systems.

Assignment 3. **Expand AP system with conflict resolution (BONUS)**

1. Start with AP system implemented in Assignment 2.
2. Extend system with conflict resolution to resolve cases when same data is changed on multiple nodes before synchronization.
3. Analyze strengths and weaknesses of chosen resolution strategy.

3.3 Final Deliverables

By the end of this section, you should have:

1. A key-value store that maintains strong data **consistency** during network failures, even at the cost of temporarily reduced **availability**.
2. A key-value store that maintains **availability** during network failures with eventual **consistency** and **conflict resolution**.
3. Measured performance data for each system from the client's perspective.