# A ROBUST DATA SYNCHRONIZATION FRAMEWORK FOR CLOUD-BASED DISTRIBUTED SYSTEMS

By

Musaeva Aliya Karimovna

A THESIS

Submitted in partial fulfillment of the requirements for the degree of

MASTER OF ARTS AND SCIENCE

In Computer Science

LAWRENCE TECHNOLOGICAL UNIVERSITY

2025

This thesis has been approved in partial fulfillment of the requirements for the Degree of MASTER OF SCIENCE in Computer Science.

Department of Arts and Science

| | |
|---|---|
| Thesis Advisor: | *Dr. Fan Li* |
| Committee Member: | *Dr. Oriehi (Destiny) Anyaiwe* |
| Committee Member: | *Dr. Wisam Bukaita* |
| Committee Member: | *Dr. Yash Patel* |
| Department Chair: | *Dr. Eric Martinson* |

# Dedication

## To my mother, teachers, and friends

who never hesitated to critique my work at every stage — without which I would neither be who I am nor would this thesis be what it is today.

To my mother, who has always been there to support me, even in the most difficult times.

To my father, whose kindness and calm strength have always guided me.

To my friends, without whom writing this thesis would have been a dull and lonely journey.

To my supervisor, Dr. Fan Li, who guided me with patience and insight throughout this process — always ready to offer thoughtful advice and helpful feedback, when I needed it most.

And to Alikhan, whose love, support, and unwavering belief in the best in me gave me strength, who always steers me in the right direction and stands by my side.

# Contents

**List of Figures** . . . . . **xi**

**List of Tables** . . . . . **xvii**

**Preface** . . . . . **xix**

**Definitions** . . . . . **xxi**

**List of Abbreviations** . . . . . **xxiii**

**Abstract** . . . . . **xxv**

**1 Introduction** . . . . . **1**

1.1 Data Synchronization . . . . . 1

1.2 Background of Distributed Databases . . . . . 3

1.3 Problem Statement . . . . . 4

1.4 Objectives and Research Questions . . . . . 4

1.5 Contribution . . . . . 5

**2 Literature Review** . . . . . **7**

2.1 Foundational Techniques . . . . . 7

2.2 Cloud-Native and Middleware Approaches . . . . . 9

2.3 Domain-Specific Synchronization . . . . . 11

2.4 Innovations in Fault Tolerance and Redundancy . . . . . 13

**3 System Design** . . . . . **17**

3.1 Coding and Caching Techniques . . . . . 17
3.2 Architecture Overview . . . . . 19
3.3 Data Placement Strategy . . . . . 20
3.3.1 Example: Calculating Batch Storage with $K = 4$ and $r = 2$ . 22
3.4 Caching and XOR . . . . . 24
3.4.1 Definition of XOR Packets . . . . . 24
3.4.2 Number of XOR Packets . . . . . 24
3.4.3 Number of Batches per XOR Packet . . . . . 25
3.4.4 Group Formation . . . . . 25
3.4.5 Batch Assignment Within Groups . . . . . 26
3.4.6 Placement into XOR Packets . . . . . 27
3.5 Example: XOR Packet Formation for 4 Nodes and 6 Batches . . . . 28
3.5.1 Group Formation . . . . . 29
3.5.2 Batch Assignment and Packet Construction . . . . . 30
3.5.3 Transmission Phase . . . . . 31
3.5.4 Decoding Rule . . . . . 32
3.6 Hub A as an Orchestrator . . . . . 33
3.7 Failure detection and Recovery Design . . . . . 34
3.7.1 Linear-Coded Recovery . . . . . 35
3.7.2 Solving the System . . . . . 37
3.7.3 Example – Linear-Coded Recovery at Node 1 . . . . . 38
3.7.4 Solving the Linear System at Node 1 . . . . . 40
3.8 Efficiency of Linear-Coded Recovery . . . . . 41
3.8.1 Uncoded Peer Recovery Cost . . . . . 42
3.8.2 Linear-Coded Peer Recovery Cost . . . . . 43
3.8.3 Communication Efficiency Gain . . . . . 43
3.8.4 Example: $K = 4, N = 6, r = 2$ . . . . . 44
3.9 Node Failure with MySQL Online . . . . . 45

**4 Implementation** . . . . . **47**
4.1 Technologies Used . . . . . 47
4.2 Databases . . . . . 48
4.3 Dataset Description . . . . . 50
4.4 Web Framework and Communication . . . . . 50
4.5 System Design Features . . . . . 51
4.6 Synchronization Modes: Direct, Partial, and Full . . . . . 57

**5 Results and Evaluations** . . . . . **59**
5.1 Evaluation Across Varying Node Counts and Redundancy Levels . . 60
5.2 Recovery Time Simulation For Different Scenarios . . . . . 64
5.3 Comparison of Direct, Partial, and Coded Transmission Methods ($K = 4$, $N = 6$ and $r = 2$ values) . . . . . 70
5.3.1 Resource Usage (CPU, Memory) ($K = 4$, $N = 6$ and $r = 2$ values) . . . . . 71
5.4 Network Performance . . . . . 73

**6 Conclusion** . . . . . **75**
6.1 Summary of Findings . . . . . 75
6.2 Recommendations For Future Work . . . . . 76

**References** . . . . . **79**

**A Proof of Existence** . . . . . **87**
Appendix: Code Snapshots . . . . . 87

# List of Figures


3.1 Distributed Architecture . . . . . 19

3.2 Nodes Communicating With Each Other in order to Get Needed Batches . . . . . 45

5.1 Transmission Time Comparison for $K = 4$ and $r = 2$ values . . . . . 60

5.2 Transmission Time Comparison for $K = 5$ and $r = 2$ values . . . . . 61

5.3 Transmission Time Comparison for $K = 7$ and $r = 2$ values . . . . . 61

5.4 Transmission Time Comparison for $K = 4$ and $r = 3$ values . . . . . 62

5.5 Transmission Time Comparison for $K = 5$ and $r = 3$ values . . . . . 62

5.6 Transmission Time Comparison for $K = 6$ and $r = 3$ values . . . . . 63

5.7 Transmission Time Comparison for $K = 7$ and $r = 3$ values . . . . . 63

5.8 Method Comparison of Transmitting Data from Data Source to General Nodes . . . . . 70

5.9 CPU Usage . . . . . 71

5.10 Memory Usage Comparison Between Scenarios($K = 4$, $N = 6$ and $r = 2$ values) . . . . . 72

5.11 Comparison of Data Transmission Between Two Databases($K = 4$, $N = 6$ and $r = 2$ values) . . . . . 73

A.1 Python function `fetch_data()` for extracting and formatting records from all tables in a MySQL database. This function queries all tables, retrieves their records, and converts special data types like decimals and timestamps for batch processing and transmission. . . . . 88

A.2 Function `create_batches(data)` that splits the input dataset into evenly sized batches for distribution. This implementation ensures exactly 6 batches (or another fixed $N$), padding if necessary. It supports batch-based placement in the synchronization pipeline. . . . . 88

A.3 Python functions for batch distribution and XOR-coded packet generation. The `distribute_batches()` method assigns batches to nodes based on a fixed redundancy pattern. The `xor_batches()` function performs element-wise encoding across batches to produce parity packets. These packets are generated using `generate_xor_packets()` . . 89

A.4 Flask routes implemented in Hub A for relaying synchronization data between Hubs B and C. The `receive_from_b()` function accepts JSON payloads from Hub B and attempts to forward them to Hub C via an HTTP POST request. If the response is successful, Hub A logs the confirmation; otherwise, it handles and logs forwarding failures. The `receive_from_c()` function similarly accepts incoming data from Hub C, making Hub A a bidirectional relay point for synchronization traffic. . . . . 90

A.5 Flask-based status monitoring functions in Hub A. The `/ping` route allows external systems to confirm that Hub A is operational. The `check_mysql_status()` function attempts to establish a connection with the MySQL database and reports connection success or failure. Similarly, `check_b_status()` and `check_c_status()` verify connectivity to Hubs B and C, respectively, by issuing GET requests to their `/ping` endpoints. These functions support centralized health monitoring and error handling across the system. . . . . 91

A.6 Recovery and monitoring functions in Hub A. The `check_c_status()` function verifies the availability of Hub C via its `/ping` endpoint, while `check_sqlite_status_in_c()` attempts to connect to the simulated Oracle database at Hub C. The `notify_hub_b_to_recover()` and `notify_hub_b_to_reset()` functions are responsible for triggering or resetting the recovery mode in Hub B through HTTP POST requests. These functions enable Hub A to orchestrate fault detection and initiate appropriate recovery workflows. . . . . 92

A.7 Central monitoring loop in Hub A, implemented in `monitor_system()`. This function continuously checks the health status of MySQL (Hub B), SQLite (Hub C), and both hubs via their respective endpoints. If MySQL is found to be down, the system triggers a recovery request to Hub B. Once MySQL is restored, the loop detects its availability and sends a reset signal to disable recovery mode. This loop executes every 10 seconds and enables Hub A to act as a live orchestrator for failure detection and automated response. . . . . 93

A.8 Python function `send_to_hub_c(data)` for transmitting data from Hub A to Hub C. This function checks whether both Hub C and its local SQLite database are operational before proceeding. It builds a synchronization payload containing the data and metadata such as a UUID-based `sync_id`, a timestamp, and a direction tag. The payload is then sent via a POST request to Hub C's processing endpoint. Success or failure responses are logged, and error handling ensures that any connection issues are clearly reported. . . . . 94

A.9 Python function `send_to_hub_b_from_c_node(data)` is executed by Hub A to forward a complete dataset from Hub C to Hub B. This function constructs a POST request to Hub B's receiving endpoint, including a JSON payload. It logs success or failure messages depending on the response status and includes exception handling for transmission errors. Below the function, the application's entry point initializes a background monitoring thread and starts the Flask server on port 5001. . . . . 95

A.10 Function `fetch_data_from_sqlite()` is executed in Hub C to extract records from all tables stored in the SQLite database. It connects to `hub_c_sqlite.db`, retrieves table names dynamically, executes a `SELECT *` query for each table (limited to 12 rows per table), and formats the results into a list of dictionaries. This function enables downstream synchronization or transmission by packaging the retrieved data in a structured format. . . . . 96

A.11 Function `receive_from_a_and_store()` runs in Hub C and handles POST requests sent from Hub A. It validates the incoming JSON payload, extracts the data and synchronization metadata, and dynamically rebuilds a local SQLite table named `received_from_b`. To avoid schema mismatches, the function drops any existing table before generating a new one based on the structure of the incoming data. This mechanism ensures compatibility and seamless ingestion of synchronized content into Hub C's database. . . . . 97

A.12 Continuation of the `receive_from_a_and_store()` function. After preparing the SQLite table schema, this block inserts each data record using a safe string conversion method to avoid encoding issues. The function tracks insertion duration, prints sync completion messages, and returns a success status if all records are stored correctly. If any step fails, it logs the error and returns a 500 response to the caller. This implementation enables robust ingestion and timing validation for full sync operations (MySQL → Hub A → Hub C → SQLite). . 98

A.13 Ping route and entry point for Hub C. The `/ping` route provides a simple heartbeat check to confirm Hub C is operational. Upon execution, the script launches a background thread responsible for initiating synchronization to Hub A and runs the Flask application on port 5002. 98

# List of Tables


5.1 Linear-Coded Recovery Time for $K = 4$, $N = 6$, $r = 2$ . . . . . 64

5.2 Linear-Coded Recovery Time for $K = 4$, $N = 12$, $r = 2$ . . . . . 65

5.3 Linear-Coded Recovery Time for $K = 4$, $N = 18$, $r = 2$ . . . . . 66

5.4 Linear-Coded Recovery Time for $K = 4$, $N = 24$, $r = 2$ . . . . . 66

5.5 Linear-Coded Recovery Time for $K = 5$, $N = 10$, $r = 2$ . . . . . 67

5.6 Linear-Coded Recovery Time for $K = 5$, $N = 20$, $r = 2$ . . . . . 68

5.7 Linear-Coded Recovery Time for $K = 5$, $N = 30$, $r = 2$ . . . . . 68

5.8 Linear-Coded Recovery Time for $K = 5$, $N = 40$, $r = 2$ . . . . . 69

# Preface

This thesis was written in partial fulfillment of the requirements for the Master of Science degree in Computer Science at Lawrence Technological University.

The journey of developing this work has been both intellectually demanding and personally transformative. It represents months of research, design, implementation, and evaluation aimed at addressing challenges in distributed data synchronization systems. Throughout this time, I had the opportunity to deepen my understanding of fault-tolerant architectures, coding techniques, and scalable system design.

This project would not have been possible without the support and guidance of many individuals. I am especially grateful to my thesis supervisor, Dr. Fan Li, whose constructive feedback, encouragement, and deep insights helped shape this research from idea to completion. I would also like to thank my family and friends for their unwavering support and belief in me.

Writing this thesis has been a milestone in my personal and professional development. I hope that the ideas and findings presented here will contribute meaningfully to future research and real-world applications in cloud-based distributed systems.

# Definitions

This section provides definitions of key terms and symbols used throughout the thesis to ensure clarity and consistency.

**Node** An individual computing unit or storage point in a distributed system that participates in data synchronization.

**Batch** A subset of the total dataset, used to divide and distribute data more efficiently across nodes.

**Redundancy Level ($r$)** The number of nodes on which each batch is stored to ensure fault tolerance.

**Total Nodes ($K$)** The total number of storage or processing nodes involved in the distributed system.

**Total Batches ($N$)** The number of distinct data batches required for complete data distribution with redundancy.

**XOR-Coded Packet** A data packet formed by combining multiple batches using the XOR operation, allowing for efficient reconstruction of missing data.

**Linear-Coded Recovery** A method of restoring lost data at a node using linearly encoded packets received from other nodes.

**Cache** Locally stored data used for reducing repeated transmissions and enabling recovery operations.

**Hub A, B, C** Architectural components in the synchronization framework. Hub B originates the data, Hub A coordinates and monitors, and Hub C receives and processes the data.

**Recovery Mode** A system state triggered when the source (e.g., MySQL) is unavailable, initiating peer-based recovery of missing data.

# List of Abbreviations

This section defines the abbreviations and acronyms used throughout this thesis. Understanding these terms is essential for clarity and consistency in discussing technical components related to distributed systems, data synchronization, and database architecture.

| | |
|---|---|
| API | Application Programming Interface |
| DBMS | Database Management System |
| HTTP | HyperText Transfer Protocol |
| JSON | JavaScript Object Notation |
| MTU | Michigan Technological University |
| MySQL | My Structured Query Language |
| SQL | Structured Query Language |
| XOR | Exclusive OR |
| REST | Representational State Transfer |
| CPU | Central Processing Unit |
| RAM | Random Access Memory |
| K | Number of nodes in the system |
| N | Total number of unique data batches |
| r | Redundancy level (number of nodes each batch is stored on) |
| s | Size of each batch |
| GF($2^m$) | Galois Field of $2^m$ elements |
| LTU | Lawrence Technological University |
| P2P | Peer-to-Peer |
| RDMS | Relational Database Management System |
| ETL | Extract, Transform, Load |

| | |
|---|---|
| CDN | Content Delivery Network |
| IoT | Internet of Things |

# Abstract

Data synchronization is critical in distributed systems to ensure consistency and communication between nodes. However, previous synchronization methods often face inefficiencies and lack sufficient fault tolerance, making them susceptible to issues such as database crashes or network failures. In this work, we propose an approach to building a fault-tolerant, robust, and efficient data synchronization system by integrating advanced caching mechanisms techniques. The system optimizes data transmission by caching incremental updates, thereby minimizing the need to send entire datasets and reducing network bandwidth usage and latency. To further enhance reliability, we implement strategies that allow for database recovery even in the event of system failures, ensuring data integrity and availability. By strengthening intermediary nodes to handle preprocessing and act as recovery points, the system guarantees continuous operation and resilience during critical failures, such as a MySQL crash. The proposed solution effectively balances efficiency with fault tolerance, offering a scalable and resilient data synchronization mechanism that ensures data consistency across distributed systems, even in the face of failures. This system is particularly beneficial for real-time applications requiring high availability and low-latency synchronization.

# Chapter 1

# Introduction

To understand the challenges addressed in this research, it is essential to first explore the fundamentals of data synchronization in distributed systems. This subsection introduces the core principles, goals, and mechanisms of synchronization, emphasizing its importance for maintaining consistency and availability across multiple database nodes.

## 1.1 Data Synchronization

Data synchronization involves a collection of methods, strategies, and protocols designed to keep multiple copies of a database—spread across different servers, systems, or locations—in sync. The goal is to guarantee that all these instances reflect the same, current data, providing consistency and accuracy across the board. It is a critical process in distributed computing environments, where data is replicated, partitioned, or shared among multiple nodes to support high availability, fault tolerance,

load balancing, and scalability. The main goal of synchronization is to keep data consistent and accurate. It ensures that any updates made to one copy of the database are correctly shared and reflected across all other copies. In distributed architectures, especially those supporting real-time or near-real-time applications, users and processes interact with the system from various endpoints, often concurrently. These interactions could include adding new records, modifying existing ones, or removing data across various parts of the database. Without an effective synchronization mechanism, such parallel and distributed operations could lead to inconsistent data views, update conflicts, stale records, or even violations of business rules and data integrity constraints. Data synchronization is essential for making sure that distributed systems remain accurate, dependable, and consistent over time. The synchronization process typically begins with change detection, where the system identifies which records or data elements have been modified in the source database since the last synchronization cycle. This step can be implemented through various means, such as database triggers, transaction logs, timestamps, version numbers, or custom change tracking mechanisms. Once the changes are identified, they must be transmitted to the target database using an efficient and reliable communication channel. Depending on the system design, this transmission may occur through messaging queues, direct API calls, replication frameworks, or customized synchronization protocols. After receiving the changes, the target systems apply them to their local database, ensuring alignment with the original data. Conflict resolution plays a important role in the synchronization process, particularly in systems that support updates in both directions. In such environments, two or more nodes may modify the same data independently, leading to conflicts that must be resolved using predefined policies or conflict-handling logic.

## 1.2 Background of Distributed Databases

A distributed database is a collection of data that is stored across multiple physical locations but appears to users as a single database system. The data could be stored on different servers located in various regions or spread across multiple administrative domains. Despite the physical distribution, the system is designed to offer smooth access, data management, and consistent information, as if everything were stored in a single, centralized location. Distributed databases have unique design and operational challenges that aren't typically encountered in centralized systems. Issues such as network latency, partitioning, concurrency control, fault tolerance, and data replication must be carefully addressed to maintain system integrity and performance. Various models of distributed databases exist, including homogeneous systems, where all nodes use the same database management system, and heterogeneous systems, where different nodes may operate using different technologies. However, the advantages of distributed databases come with increased complexity in maintaining consistency and synchronization. Managing consistent updates across multiple nodes, handling simultaneous data access smoothly, and recovering effectively from partial failures are fundamental challenges that distributed database systems need to tackle. The evolution of distributed databases has been closely linked with advancements in networking technologies, cloud computing, and large-scale data processing frameworks. As modern applications demand higher levels of availability, scalability, and responsiveness, the role of distributed databases and the mechanisms for their synchronization have become increasingly critical to system design.

## 1.3 Problem Statement

As distributed database systems grow in scale and complexity, traditional synchronization methods encounter limitations. Replicating the entire dataset might seem simple in theory, but in practice, it results in network bandwidth consumption, higher CPU and memory demands, and longer recovery times when systems encounter failures. These inefficiencies become problematic in systems requiring high availability, low latency, and efficient resource utilization. In cases like real-time analytics, cloud platforms, and distributed transactions, simple synchronization adds delays and reduces system responsiveness. Restoring a database often involves sending large amounts of data again, which adds pressure to the network and slows the operation. There is a need for lighter synchronization approaches that reduce data transfer, limit processing time, and support recovery without using heavy coordination tools. This thesis introduces a synchronization system based on batching, redundancy, and coding theory to improve how data is kept up to date across distributed environments.

## 1.4 Objectives and Research Questions

The objective of this research is to design, implement, and evaluate an efficient and fault-tolerant database synchronization system for distributed environments. The system is designed to overcome the previous synchronization methods by minimizing network traffic, making better use of CPU and memory resources, and allowing for quick recovery after database failures. The study primarily explores creating a synchronization method that combines caching with carefully managed redundancy to cut down on unnecessary data transfers. It also uses XOR-based packet generation

techniques, enabling full database reconstruction and more efficient recovery processes. The research also aims to develop a flexible adjustment approach for caching and data placement, personalized to the size of the dataset. This ensures the system stays scalable and responsive as data volumes grow. This study seeks to answer several key research questions.

† First, the focus is on exploring how combining caching, redundancy, and XOR-based methods can improve synchronization speed and reliability in distributed databases.

† Second, it investigates how the proposed system affects network load and computational resource usage during synchronization processes.

† Third, it evaluates the effectiveness of the recovery mechanism in reconstructing missing or corrupted data following database crashes.

† Finally, it explores how the dynamic adjustment of batches and nodes affects the system's ability to scale efficiently and make the most effective use of resources as the dataset expands.

Through addressing these objectives and research questions, the study aims to contribute to the advancement of lightweight, scalable, and resilient database synchronization techniques that are applicable to real-world distributed systems.

## 1.5 Contribution

This research introduces a synchronization system for distributed databases that reduces data transfer, supports recovery operation, and adapts to different data volumes. The system uses caching, redundancy, and coding theory methods to avoid

sending data unnecessary amount of time while keeping information consistent across nodes. By breaking data into smaller parts and adding redundancy, it avoids overloading system resources. A key part of the design is the use of coding methods that helps to restore lost or needed data. These methods allow missing data to be recovered without sending the dataset again. The system also adjusts caching and data placement. Tests were run on real database setups to compare performance in different conditions. The results show that the proposed method performs better in both synchronization and recovery than basic approaches. This work offers a practical method for keeping distributed data in sync, dealing with failures, and scaling to larger systems without needing complex infrastructure.

# Chapter 2

# Literature Review

A comprehensive review of existing literature is essential to identify the limitations of current data synchronization techniques and to justify the development of a more efficient and fault-tolerant system. This chapter explores various approaches that have been proposed in previous research, focusing on foundational synchronization methods, cloud-native and middleware frameworks, domain-specific synchronization challenges, and innovations in fault tolerance and redundancy. The review highlights the gaps and trade-offs in current solutions.

## 2.1 Foundational Techniques

Data synchronization in distributed systems has been widely studied, particularly with the growing need for real-time consistency, fault tolerance, and scalability. Despite considerable research, many current synchronization systems continue to struggle with maintaining consistent performance and reliability when nodes or databases experience failures.

Several foundational approaches have aimed to optimize synchronization by minimizing the volume of transmitted data. Log-based synchronization methods, for instance, transmit only the updated records since the last cycle, thereby reducing bandwidth usage and computational overhead. While effective under stable conditions, these methods are highly dependent on the availability of the primary database.Foundational replication models like optimistic replication aimed to improve availability by allowing updates to occur independently on different replicas, resolving conflicts later through reconciliation [1]. This model laid the groundwork for many modern distributed systems that prioritize responsiveness over immediate consistency.

In the event of system failures or interruptions, the synchronization process can break down entirely, leading to data loss or extended downtime [2]. Early research on mobile data synchronization highlighted core issues like network instability, resource constraints, and consistency trade-offs in asynchronous environments [3]. Differential synchronization has also been proposed as a lightweight, near real-time strategy that supports continuous updates with minimal bandwidth, making it suitable for collaborative environments [4].

The Bayou system introduced a pioneering approach to data synchronization in weakly connected and mobile environments. It supported eventual consistency, conflict resolution, and offline updates, laying the groundwork for later distributed systems that prioritize availability over strict consistency [5]. Synchronization challenges have also been extensively explored in high-performance computing, where efficient coordination among parallel threads or processes remains critical for minimizing overhead and ensuring correctness [6]. A structured taxonomy of evaluation indicators has been shown to improve the consistency and clarity of system assessments, especially in complex, multi-dimensional environments [7]. Early approaches to synchronization in heterogeneous database systems employed XML parsing and middleware layers to bridge platform differences, offering baseline interoperability at the cost of increased

latency and structural rigidity [8].

Although such methods established early efficiency benchmarks, they are increasingly viewed as inadequate for modern distributed systems that require both responsiveness and resilience. This creates a need for synchronization frameworks that incorporate failure recovery mechanisms without undermining the benefits of incremental data transfer.

## 2.2 Cloud-Native and Middleware Approaches

As distributed architectures evolved, synchronization mechanisms expanded into cloud-native and middleware-based solutions. Cloud-native orchestration platforms such as Kubernetes have contributed significantly to the scalability and automation of distributed applications. However, despite these benefits, Kubernetes lacks built-in support for redundancy or failure-resilient synchronization, rendering systems vulnerable during database crashes, network partitions, or node outages [2] [9] [10].

Global data synchronization (GDS) serves as a foundational mechanism in supply-chain systems, enabling consistent, real-time updating of product information across business partners. The evolution of standards and platforms like the Global Data Synchronization Network (GDSN) has enhanced data integrity and streamlined e-collaboration—though challenges remain in implementation and organizational alignment. These insights open research opportunities into GDS adoption dynamics and ecosystem network effects [11]. A sampled-data synchronization scheme has been developed for stochastic Markovian jump neural networks subject to time-varying delays. By applying LMI-based design under aperiodic sampling, this method guarantees mean-square exponential synchronization—demonstrating robustness against

randomness and varying delays [12].

Non-invasive EEG synchronization measures—specifically the Phase Lag Index (PLI) and the Weighted Phase Lag Index (WPLI)—were shown to reliably detect preictal states several minutes before the onset of the seizure. Applied to a 14-patient scalp-EEG dataset, the threshold-based classifier achieved high seizure prediction accuracy with few false alarms, demonstrating its practical potential for portable monitoring systems [13]. A low-overhead hardware synchronization mechanism, SynCron, has been proposed specifically to support fine-grained synchronization in near-data processing (NDP) architectures. The design incorporates multiple hardware primitives to accelerate barrier and mutex operations directly within the memory subsystem, significantly reducing latency compared to traditional CPU-centric techniques [14]. A resource allocation framework was introduced to dynamically synchronize virtual Metaverse environments with real-time IoT-generated data. This system enables IoT service instances to adaptively allocate computing, storage, and communication resources, maintaining consistency and low latency between the simulated virtual world and its physical counterpart [15].

Middleware solutions, particularly message brokers like RabbitMQ, offer a modular and decoupled approach to data transfer. By separating producers and consumers through asynchronous queues, such systems support optimized communication and easier system decomposition [16] [17] [18]. Still, they rely heavily on supplementary recovery plans and backup strategies to manage data loss during failure scenarios.

Machine learning infrastructures such as PyTorch Distributed further illustrate these challenges. While these systems enable high-throughput parallel training across multiple nodes, they also reveal the limitations of current synchronization practices in maintaining consistency and throughput under distributed computation [19]. These observations highlight the pressing need for synchronization solutions that can maintain

robustness without impeding the benefits of scalability and decentralization [20] [21]

In practice, synchronization failures often stem from common but overlooked issues such as clock drift, network partitions, and inconsistent state reconciliation. ClearInsights [22] highlights the importance of addressing such root causes through well-structured data modeling, consistent event ordering, and clear conflict resolution strategies. In enterprise applications like Salesforce CRM, synchronization becomes even more complex due to bidirectional data flows, strict API limits, and asynchronous communication patterns. As noted by GeeksforGeeks [23], developers must account for edge cases such as rate throttling, latency-induced state mismatch, and partial failures. Ensuring consistent and predictable packet delivery is crucial for reliable synchronization in distributed environments.

## 2.3 Domain-Specific Synchronization

Synchronization requirements become more complex in domain-specific environments where timing precision and continuous availability are critical. In Industry 4.0 contexts, the fusion of production and logistics (PiL) systems demands real-time synchronization to maintain continuous process flow. Although many early frameworks focused on system modeling and finished-product data integration, fewer addressed runtime disruptions or uncertainties that affect production-logistics coherence. Recent research has introduced stochastic control mechanisms such as stochastic mixed jump neural networks (SMJNNs), recurrent delayed neural networks (RDNNs), and fuzzy sampled-data control to achieve synchronization under irregular sampling and variable system conditions [24] [25].

Parallel to these advances, innovative control models have emerged to tackle the challenge of fault tolerance in uncertain environments. Stochastic control strategies and neural synchronization models enable systems to operate reliably even with incomplete or delayed data. However, while these solutions improve stability, they tend to be complex and computationally heavy, limiting their scalability and applicability to broader distributed systems [24] [25]. To reduce synchronization latency in time-sensitive IoT applications, fog computing has emerged as a decentralized alternative to cloud-based synchronization. By offloading processing to edge devices, fog frameworks help mitigate delays and improve data consistency near the source [26].

A digital twin–enabled real-time synchronization system (DT-SYNC) was developed to enhance planning, scheduling, and execution (PSE) for precast on-site assembly. By leveraging high-fidelity digital twins, the system provides cyber-physical visibility and traceability, enabling dynamic allocation of resources and coordination of operations via a ticket-based model. Numerical experiments and a robotic testbed demonstrate DT-SYNC's ability to maintain resilience and just-in-time synchronization in constrained, urban construction environments [27].

Healthcare applications also place high importance on accurate synchronization, especially in electroencephalogram (EEG) signal analysis where precision timing is vital for diagnostics. Frameworks like SynCron demonstrate how synchronization near the data source can significantly improve performance in environments with intensive computation demands [28] [29]. However, these solutions often lack mechanisms for effective failure recovery, which is crucial in medical and time-sensitive systems.

In the UAV domain, synchronization frameworks are used to support real-time video streaming in rapidly changing network conditions. These frameworks often employ caching strategies to improve transmission efficiency. Deep learning models have also been explored for synchronization tasks involving temporal data, particularly

in healthcare and sensor networks. Such models can automatically learn complex time-dependent patterns, offering potential for adaptive synchronization under dynamic and noisy conditions [30]. In telemedicine, synchronization delays due to poor connectivity can critically affect diagnostic accuracy and real-time responsiveness [31]

## 2.4 Innovations in Fault Tolerance and Redundancy

Recent efforts have shifted toward integrating fault tolerance and redundancy into synchronization frameworks through advanced architectural strategies.The fusion of digital twin and blockchain technologies has introduced new possibilities for maintaining synchronization across heterogeneous systems. In construction and industrial monitoring, for example, blockchain ensures traceability and secure information exchange, while digital twins offer real-time modeling and feedback loops [32]. These methods promote consistency and transparency but often prioritize data security over synchronization recovery efficiency. Blockchain technologies have also been applied to maintain data integrity in IoT environments, offering tamper-resistant logs and decentralized verification mechanisms for synchronization frameworks [33] [34]. Several systems have addressed fault tolerance in distributed synchronization by combining redundancy techniques with stable transmission mechanisms, demonstrating the ongoing relevance of resilient data coordination strategies [35].

Chrome's synchronization feature enables automatic syncing of browser data—such as bookmarks, extensions, history, and passwords—across desktop and mobile devices. Users can switch between syncing everything or customize the data types to sync via the settings "Manage sync", allowing selective data control across devices [36]. MultCloud's "Google Cloud Sync" feature enables automated synchronization between

Google Drive and other cloud storage accounts, such as the device's desktop, another Google profile, or third-party clouds, without local downloads. It supports multiple sync modes, including one-way, two-way, real-time, incremental, and scheduled synchronization, and enhances data management via filters, notifications, and security options like 256-bit encryption [37].

Despite these innovations, a consistent gap remains: many existing systems either optimize for speed without sufficient fault recovery, or introduce robustness at the expense of performance. There is still a need for lightweight synchronization frameworks that balance redundancy, efficiency, and resilience, especially for systems operating under constrained resources or volatile conditions. One emerging direction from industry is the Event-Carried State Transfer (ECST) paradigm, which embeds complete state information within each event to ensure self-contained synchronization and eliminate dependence on central sources [38].

Maddah-Ali and Niesen [39] proposed a foundational coded caching model that minimizes transmission load through prefetching and centralized multicast delivery. Building upon their framework, this thesis applies similar redundancy principles in a decentralized synchronization context. Unlike their broadcast-oriented model, the proposed system enables peer-to-peer linear-coded recovery, allowing nodes to reconstruct missing data even when the central source is unavailable. This shift supports fault-tolerant synchronization in distributed database environments rather than content delivery networks.

While this work proposes a new approach to synchronization that combines batch placement, caching, and linear-coded recovery, it is important to note that it builds on concepts introduced in prior literature. Existing systems, such as those based on

coded caching for content delivery [39] and native cloud orchestration [40]—offer valuable foundations, but they do not explicitly address fault-tolerant bidirectional synchronization across distributed hubs in the same configuration explored here. Given the novelty of the proposed framework, a direct quantitative comparison with existing implementations was not feasible within the scope of this study. However, the system design was conceptually aligned with well-established models to ensure consistency with proven practices.

# Chapter 3

# System Design

## 3.1 Coding and Caching Techniques

This section presents the coding and caching strategies integrated into the proposed system to support efficient, fault-tolerant data synchronization across distributed nodes. These techniques are designed to minimize communication overhead, enhance data availability during failures, and maintain system consistency even when parts of the infrastructure become temporarily inaccessible. By drawing on concepts from recent advances in coded caching theory, the system combines proactive data placement with coded multicasting to address the core challenges of redundancy and recovery in real-time environments.

Ensuring fault tolerance in distributed synchronization systems is a critical requirement, particularly in environments where node failures, database outages, or unstable network conditions can disrupt the flow and integrity of data. To address these challenges, the proposed system adopts a coded caching framework, which introduced

the concept of combining caching and coding to improve efficiency and robustness in data delivery. This work applies a theoretical framework [39], which formalizes the use of coded caching to minimize transmission load in distributed environments. The methodology adopted from the paper includes two core components: content placement based on redundancy-aware caching, and coded delivery through group-based multicasting. These techniques are used to optimize communication efficiency and ensure recoverability in distributed synchronization settings.

The strategy is based on storing subsets of data redundantly across multiple nodes, following a predefined combinatorial structure. Although the original framework was designed for content delivery networks, its principles are extended here to support distributed database synchronization. Additionally, the theoretical recovery rate introduced in [39] is used to estimate the number of transmissions required to fulfill all missing data across the network. This rate is based on a normalized communication cost derived from the cache size and redundancy level, and serves as a guide for evaluating the efficiency of the applied approach.

Although the original paper [39] assumes a shared-link broadcast model, the methods were adapted to suit peer-to-peer and multi-hop communication contexts. However, the mathematical foundation for group selection, redundancy, and coded packet construction is preserved. The selection of this methodology was motivated by its provable communication efficiency under storage constraints and its adaptability to network failure conditions. The paper's results serve as a foundation for the system's redundancy logic and fault tolerance strategy, allowing the practical implementation to benefit from theoretically optimized communication bounds.

## 3.2 Architecture Overview

The proposed synchronization system is structured around a hub-distributed architecture, designed to enable efficient data synchronization, ensure fault tolerance, and support system scalability [27]. The architecture comprises three primary Hubs, Hub B, Hub A, and Hub C, each with a distinct role in the synchronization pipeline. Hub B serves as the primary sender and hosts the source database. Hub A functions as the orchestrator, responsible for health monitoring and coordination. Hub C acts as the receiver, storing synchronized data in a second database. The synchronization process begins at Hub B and progresses through Hub A to its final destination at Hub C.

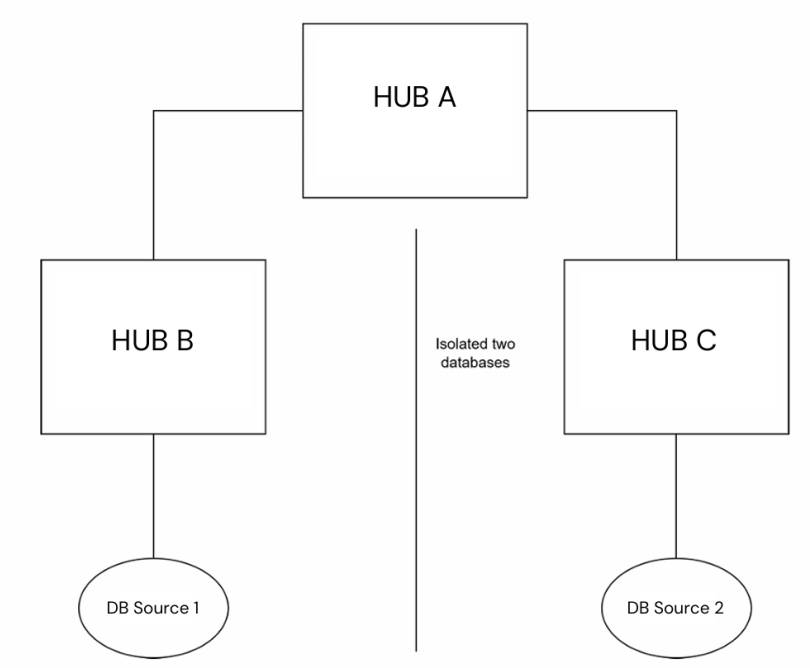



**Figure 3.1:** Distributed Architecture

Hub B and Hub C serve as active participants in bidirectional data exchange. Hub

B originates the dataset and sends it to Hub C via Hub A. Hub C, after processing and storing the data in its local instance, can also send data back to Hub B. This two-way communication enables the system to support both forward synchronization, ensuring that both hubs can contribute to maintaining data consistency.

## 3.3 Data Placement Strategy

The process starts with pulling data from the primary database source (e.g., MySQL), which contains the full dataset to be distributed across a set of nodes. The full dataset is divided into smaller units called *batches*, where each batch represents a subset of the data that can be transmitted and stored on different nodes.

**Note:** 'Batch count' ($N$) refers to the number of data partitions, while 'batch size' denotes the number of records in each partition. Instead of dividing the data arbitrarily, the system follows a structured approach. To organize the data placement strategy, we define the following parameters:

- $K$: the total number of nodes in the distributed system.
- $r$: the redundancy level, representing how many nodes each batch should be stored on.
- $N$: the total number of batches required to distribute the data with redundancy.
- $M$: the number of batches stored on each node (i.e., batches per node).
- $\eta$: a positive integer that scales the number of possible unique combinations. It allows for more batches than the base combinatorial minimum.

Each batch is placed on a unique subset of $r$ nodes out of the total $K$. Specifically, if each batch is stored on $r$ different nodes, the system can tolerate up to $r-1$ node failures without data loss.

To determine how many unique batches $N$ are needed, we calculate how many distinct combinations of $r$ nodes can be selected from the $K$ total nodes. This is given by the binomial coefficient [39]:

$$N = \eta \cdot \binom{K}{r} = \eta \cdot \frac{K!}{r!(K-r)!}. \tag{3.1}$$

**Note:** $\eta$ is a scaling factor that allows increasing the number of distinct batches beyond the strict minimum required for redundancy.

As shown in Equation (3.1)., this combinatorial approach ensures full coverage of all redundancy subsets. Since each batch is stored on $r$ different nodes to provide redundancy, the total number of batch placements in the system is:

$$\text{Total placements} = N \cdot r. \tag{3.2}$$

As shown in Equation (3.2)., this represents the total number of times batches are stored across the nodes. To distribute the load evenly, we divide the total number of placements by the number of nodes $K$. As a result, the number of batches that each node must store is given by [39]:

$$M = \frac{N \cdot r}{K}. \tag{3.3}$$

This ensures a balanced and uniform distribution of storage responsibilities across the system, as shown in Equation (3.3). Every node holds the same number of batches, which helps optimize memory usage and simplifies retrieval logic during reconstruction or synchronization.

In summary, this strategy guarantees the following:

† Each batch is replicated across exactly $r$ nodes.

† Each node stores exactly $\frac{N \cdot r}{K}$ batches.

† The system can tolerate up to $r - 1$ node failures without losing any data.

### 3.3.1 Example: Calculating Batch Storage with $K = 4$ and $r = 2$

Let us consider a system with $\eta = 1$, $K = 4$ nodes, and a redundancy level of $r = 2$, meaning each batch must be stored on 2 different nodes.

**Step 1: Total Number of Batches**

The number of unique batches $N$ is determined using the binomial coefficient:

$$N = \binom{4}{2} = \frac{4!}{2!(4-2)!} = \frac{24}{4} = 6.$$

In this scheme, 6 unique batches are created.

**Step 2: Total Placements and Batches per Node**

Each batch is stored on $r = 2$ nodes:

$$\text{Total placements} = N \cdot r = 6 \cdot 2 = 12.$$

These 12 total batch placements are spread evenly across the $K = 4$ nodes:

$$M = \frac{N \cdot r}{K} = \frac{12 \cdot 1}{4} = 3.$$

**Step 3: Node Assignments**

Each node will store exactly 3 batches. For example, the 6 batches can be assigned to the following node pairs:

$$\{(1,2),\ (1,3),\ (1,4),\ (2,3),\ (2,4),\ (3,4)\}.$$

Thus:

- † Node 1 stores: Batches 1, 2, 3.
- † Node 2 stores: Batches 1, 4, 5.
- † Node 3 stores: Batches 2, 4, 6.
- † Node 4 stores: Batches 3, 5, 6.

This confirms that each node holds 3 batches, and each batch is stored on 2 nodes — satisfying both the redundancy and balance criteria.

## 3.4 Caching and XOR

To reduce network load and improve synchronization speed, the system applies coding techniques in the form of encoded XOR-packets. These packets are generated by MySQL and sent to intermediate hubs (e.g., Hub B). Since the full dataset is not sent to every node, each node receives only a subset of the data (i.e., selected batches). Direct transmission of all missing batches to each node would result in considerable network overhead. To avoid this, XOR-packets are introduced as an efficient and compact method for reconstruction.

### 3.4.1 Definition of XOR Packets

In the coded caching scheme, an XOR packet is a coded transmission that simultaneously serves multiple nodes. Each XOR packet is constructed by combining multiple batches—one intended for each node in a group—using the bitwise XOR operation. Instead of sending these batches separately, the server transmits their XOR sum. Each node can then decode its intended batch by removing the others from the sum using its local cache.

### 3.4.2 Number of XOR Packets

Let $K$ be the number of nodes, $N$ the number of batches, and $M$ the number of batches per node. The normalized coded caching rate is given by [39]:

$$R_C(M) = K \cdot \left(1 - \frac{M}{N}\right) \cdot \frac{1}{1 + \frac{KM}{N}} \quad . \tag{3.4}$$

Since each XOR packet carries one batch of information, the total number of XOR-coded packets that must be transmitted is [39]:

$$\text{Total packets} = R_C(M) \cdot N = K \cdot \left(1 - \frac{M}{N}\right) \cdot \frac{N}{1 + \frac{KM}{N}} \quad . \tag{3.5}$$

This expression accounts for the redundancy introduced during placement and ensures minimal transmission volume while satisfying all node demands [39].

### 3.4.3 Number of Batches per XOR Packet

Each XOR-coded packet is intended to serve a group of $r + 1$ nodes, where the redundancy parameter $r$ is defined by:

$$r = \frac{KM}{N}. \tag{3.6}$$

Therefore, each XOR packet must include $r + 1$ batches—one for each node in the group. These batches are chosen such that every node in the group is able to decode its missing batch using the XOR packet and its locally cached content.

### 3.4.4 Group Formation

To prepare for efficient coded delivery, the server constructs all possible combinations of $r + 1$ distinct nodes from the total set of $K$ nodes. Each of these combinations defines a group $\mathcal{G}$, where [39]:

$$\binom{K}{r+1} \quad . \tag{3.7}$$

is the total number of such groups.

Each group $\mathcal{G} = \{i_1, i_2, \ldots, i_{r+1}\} \subseteq \{1, \ldots, K\}$ consists of $r+1$ nodes, and serves as a target for a single coded transmission. Specifically, the server will use this grouping to construct an XOR-coded packet that contains a combination of batches such that each node in the group can decode its missing batch using the data it already holds. These groups are precomputed and independent of any specific data requests, relying solely on the redundancy-based placement strategy defined during the caching phase. This group-based structure is fundamental to minimizing transmission overhead, as it enables a single message to satisfy multiple nodes simultaneously.

### 3.4.5 Batch Assignment Within Groups

Once groups of $r+1$ nodes are formed, the server assigns a specific batch to each node within the group $\mathcal{G}$. Let the group be denoted as $\mathcal{G} = \{i_1, i_2, \ldots, i_{r+1}\}$, where each $i_j$ represents the identifier of the $j$-th node in the group (for $j = 1, 2, \ldots, r+1$).

The goal is to construct a coded packet that enables each node to recover exactly one batch it does not have, using information available from the other nodes in the group.

For each node $i_j \in \mathcal{G}$, the server selects a batch $b_{i_j}$ such that:

$$b_{i_j} \notin Z_{i_j}, \quad b_{i_j} \in \bigcap_{k \in \mathcal{G} \setminus \{i_j\}} Z_k,$$

where $Z_k$ represents the set of batches cached at node $k$.

This condition ensures two things:

- † **Missing at Target Node:** The batch $b_{i_j}$ is not stored in node $i_j$'s cache, meaning it is a required batch for that node.
- † **Recoverable from Peers:** The batch $b_{i_j}$ is present in all other nodes in the group, allowing node $i_j$ to decode it from the coded packet using its local cache.

By satisfying this constraint for each node in the group, the server guarantees that a single coded transmission—containing the XOR of all assigned batches—will allow every node in the group to recover its missing data using only the information already stored in its cache.

### 3.4.6 Placement into XOR Packets

Once the group and batch assignments are determined, the server constructs the XOR-coded packet for group $\mathcal{G}$ as:

$$P_{\mathcal{G}} = b_{i_1} \oplus b_{i_2} \oplus \cdots \oplus b_{i_{r+1}}.$$

The packet $P_{\mathcal{G}}$ is then broadcast to all nodes in the group. Each node $i_j$ decodes its assigned batch using:

$$b_{i_j} = P_{\mathcal{G}} \oplus \bigoplus_{k \in \mathcal{G} \setminus \{i_j\}} b_{i_k}.$$

This decoding is enabled by the fact that node $i_j$ already has all other $r$ batches involved in the packet. This construction ensures that each transmission is simultaneously useful for all participating nodes $r + 1$.

## 3.5 Example: XOR Packet Formation for 4 Nodes and 6 Batches

We consider a system with the following parameters:

- Number of nodes: $K = 4$.
- Number of batches: $N = 6$.
- Redundancy: $r = 2$, meaning each batch is stored on exactly 2 nodes.
- Group size for XOR packets: $r + 1 = 3$.

The total number of XOR-coded packets (i.e., the number of unique groups of 3 nodes from 4) is calculated as:

$$\text{Total packets} = \binom{K}{r+1} = \binom{4}{3} = 4.$$

This corresponds to the group count defined in Equation (3.1). The batch placement is predefined so that each batch is placed on 2 nodes. Let $\mathcal{N}_i$ represent the set of batches stored in node $i$. Then:

$$\mathcal{N}_1 = \{B_1, B_2, B_3\}$$
$$\mathcal{N}_2 = \{B_1, B_4, B_5\}$$
$$\mathcal{N}_3 = \{B_2, B_4, B_6\}$$
$$\mathcal{N}_4 = \{B_3, B_5, B_6\}$$

Each batch appears in two different node caches as required:

$$B_1 \in \mathcal{N}_1 \cap \mathcal{N}_2$$

$$B_2 \in \mathcal{N}_1 \cap \mathcal{N}_3$$

$$B_3 \in \mathcal{N}_1 \cap \mathcal{N}_4$$

$$B_4 \in \mathcal{N}_2 \cap \mathcal{N}_3$$

$$B_5 \in \mathcal{N}_2 \cap \mathcal{N}_4$$

$$B_6 \in \mathcal{N}_3 \cap \mathcal{N}_4$$

### 3.5.1 Group Formation

We form all possible combinations of 3 nodes out of 4. These are:

$$\mathcal{G}_1 = \{1, 2, 3\}$$

$$\mathcal{G}_2 = \{1, 2, 4\}$$

$$\mathcal{G}_3 = \{1, 3, 4\}$$

$$\mathcal{G}_4 = \{2, 3, 4\}$$

Each of these groups will be assigned a packet $P_i$, which consists of the XOR of one batch for each node in the group. The batch assigned to a node must:

- be missing from that node's cache.

- be present in the caches of the other two nodes in the group.

### 3.5.2 Batch Assignment and Packet Construction

We now construct the coded packets $P_1, P_2, P_3, P_4$ based on the groups $\mathcal{G}_1, \mathcal{G}_2, \mathcal{G}_3, \mathcal{G}_4$ formed above. For each group, we identify one batch per node that satisfies two conditions: (1) the batch is missing from that node, and (2) it is available in the caches of the other two nodes in the group.

Each packet $P_i$ is then generated by computing the XOR of the selected batches from all three nodes in the corresponding group.

**Group {1, 2, 3}.**

$$\begin{aligned}
&\text{Node 1: missing } B_4, \quad B_4 \in \mathcal{N}_2 \cap \mathcal{N}_3 \\
&\text{Node 2: missing } B_2, \quad B_2 \in \mathcal{N}_1 \cap \mathcal{N}_3 \\
&\text{Node 3: missing } B_1, \quad B_1 \in \mathcal{N}_1 \cap \mathcal{N}_2 \\
&\Rightarrow P_1 = B_1 \oplus B_2 \oplus B_4
\end{aligned}$$

**Group {1, 2, 4}.**

$$\begin{aligned}
&\text{Node 1: missing } B_5, \quad B_5 \in \mathcal{N}_2 \cap \mathcal{N}_4 \\
&\text{Node 2: missing } B_3, \quad B_3 \in \mathcal{N}_1 \cap \mathcal{N}_4 \\
&\text{Node 4: missing } B_1, \quad B_1 \in \mathcal{N}_1 \cap \mathcal{N}_2 \\
&\Rightarrow P_2 = B_1 \oplus B_3 \oplus B_5
\end{aligned}$$

**Group {1, 3, 4}.**

$$\begin{aligned}
&\text{Node 1: missing } B_6, \quad B_6 \in \mathcal{N}_3 \cap \mathcal{N}_4 \\
&\text{Node 3: missing } B_3, \quad B_3 \in \mathcal{N}_1 \cap \mathcal{N}_4 \\
&\text{Node 4: missing } B_2, \quad B_2 \in \mathcal{N}_1 \cap \mathcal{N}_3 \\
&\Rightarrow P_3 = B_2 \oplus B_3 \oplus B_6
\end{aligned}$$

**Group {2, 3, 4}.**

$$\begin{aligned}
&\text{Node 2: missing } B_6, \quad B_6 \in \mathcal{N}_3 \cap \mathcal{N}_4 \\
&\text{Node 3: missing } B_5, \quad B_5 \in \mathcal{N}_2 \cap \mathcal{N}_4 \\
&\text{Node 4: missing } B_4, \quad B_4 \in \mathcal{N}_2 \cap \mathcal{N}_3 \\
&\Rightarrow P_4 = B_4 \oplus B_5 \oplus B_6
\end{aligned}$$

### 3.5.3 Transmission Phase

Once all XOR-coded packets $P_1, P_2, P_3, P_4$ are constructed, the server transmits them to their corresponding groups of nodes. Each packet is sent only once, but it serves all $r + 1$ nodes in its group simultaneously.

$$\begin{aligned}
P_1 &= B_1 \oplus B_2 \oplus B_4 \quad \text{sent to Nodes 1, 2, 3} \\
P_2 &= B_1 \oplus B_3 \oplus B_5 \quad \text{sent to Nodes 1, 2, 4} \\
P_3 &= B_2 \oplus B_3 \oplus B_6 \quad \text{sent to Nodes 1, 3, 4} \\
P_4 &= B_4 \oplus B_5 \oplus B_6 \quad \text{sent to Nodes 2, 3, 4}
\end{aligned}$$

Each node listens to the packets relevant to its group and applies the decoding rule to retrieve the batch it is missing. Since every node has the other two batch values involved in its group packet, the XOR operation allows full recovery.

This method ensures minimal network usage because each transmission benefits multiple receivers at once. The server only sends a total of 4 coded packets to achieve full synchronization across all nodes.

### 3.5.4 Decoding Rule

Every node receives the XOR packet for its group. Since each node knows the other two batches in the packet, it can decode its missing batch using:

$$b_i = P - \sum_{j \in G \setminus \{i\}} b_j$$

In this context, the subtraction is interpreted as XOR. For example, Node 1 in group 1 computes:

$$B_4 = P_1 \oplus B_1 \oplus B_2$$

This logic is applied identically across all four groups. To illustrate the decoding at the receiver side more clearly, consider the following example:

**Example: Decoding at Node 1 in Group $\mathcal{G}_1 = \{1, 2, 3\}$**

The server transmits packet $P_1 = B_1 \oplus B_2 \oplus B_4$ to all nodes in $\mathcal{G}_1$. Node 1 already stores $B_1$ and $B_2$ in its local cache $\mathcal{N}_1$. Upon receiving $P_1$, it performs the XOR operation:

$$B_4 = P_1 \oplus B_1 \oplus B_2$$

Thus, Node 1 successfully recovers the missing batch $B_4$. Similarly, Node 2 in the same group knows $B_1$ and $B_4$ and decodes:

$$B_2 = P_1 \oplus B_1 \oplus B_4$$

Node 3 knows $B_2$ and $B_4$ and decodes:

$$B_1 = P_1 \oplus B_2 \oplus B_4$$

This demonstrates that all nodes in a group can independently recover their missing batch using the shared XOR-coded packet and their local cache. The same procedure applies to packets $P_2$, $P_3$, and $P_4$ for the remaining groups.

## 3.6 Hub A as an Orchestrator

Hub A is responsible for performing database validation and system health monitoring. Hub A relies on multiple distributed nodes to verify the operational status of both the MySQL source and the Oracle storage. Each monitoring node in Hub A checks different aspects of database health and synchronization consistency. These

distributed verification processes enhance system robustness by eliminating single points of failure in the health monitoring mechanism.

Hub A serves two primary functions: continuous database health monitoring and coordination of data forwarding. This health monitoring is ongoing and is not directly linked to the synchronization cycle or the forwarding of datasets. The objective of the monitoring nodes is to detect anomalies or failures as they appear, supporting overall system resilience, but not to block or control the flow of synchronized data. In parallel with ongoing health monitoring, one designated node within Hub A is responsible for transmitting the complete synchronized dataset. This node functions independently of monitoring outcomes and is designated for transmission tasks according to the system setup, allowing synchronization to happen smoothly.

In scenarios where the MySQL server becomes unavailable, Hub A activates peer-initiated recovery mode. In this mode, Hub A takes the role of coordinator, enabling nodes to recover missing batches using XOR-coded packets constructed from their cached data.

## 3.7 Failure detection and Recovery Design

In the event of a node failure, the system does not rely fully on the database source to resend the lost data. Instead, peer nodes collaborate using cached data and node's own local batches to reconstruct and restore the lost information. This offloads the burden from MySQL and ensures fault tolerance and network efficiency.

### 3.7.1 Linear-Coded Recovery

The MySQL server's unavailability triggers the start of a Linear-Coded Recovery. The proposed system enables peer-based recovery using linearly encoded messages transmitted among the nodes. This section presents the theoretical framework for recovering multiple missing batches per node using a minimal number of coded transmissions. The proposed recovery strategy is based on principles of linear coding over finite fields, allowing each node in a distributed system to reconstruct multiple missing data batches solely through peer-to-peer exchange.

Consider a distributed system with $K$ nodes and $N$ data batches $\{B_1, B_2, \ldots, B_N\}$. Each node stores a subset of $r$ batches. In the event of a central source failure, each node aims to reconstruct the $N-r$ batches it is missing, relying only on coded packets received from other nodes.

Each of the $K$ nodes constructs a coded packet by forming a linear combination of locally stored batches. These combinations are constructed over a finite field $\mathbb{F}_{2^m}$, and the coded packets are shared among the other $K-1$ nodes.

The architecture ensures the following:

- † Each node constructs one coded packet based solely on the batches in its own local cache.
- † Each coded packet is sent to the other nodes.
- † Each node receives coded packets.
- † Upon receiving the coded packets, each node subtracts the known components

from the received values and isolates a system of linear equations involving its unknown batches.

Assume that Node $i$ is missing three data batches, denoted as $B_{x_i}, B_{y_i}$, and $B_{z_i}$. To facilitate recovery, the system transmits three coded packets, $Q_1, Q_2$, and $Q_3$, to this node. Each coded packet is constructed as a linear combination of three batches over the finite field $\mathbb{F}_{2^m}$, such that:

$$Q_j = B_{a_j} + B_{b_j} + B_{c_j}, \quad \text{for } j \in \{1, 2, 3\}.$$

Here, $B_{a_j}, B_{b_j}$, and $B_{c_j}$ represent the specific batches included in coded packet $Q_j$. The subscripts $a_j, b_j, c_j$ refer to the indices of these batches and vary with $j$ to ensure that each packet contains a distinct combination of batches. The system is designed so that, from Node $i$'s perspective, each packet contains exactly two batches that are missing (from the set $\{B_{x_i}, B_{y_i}, B_{z_i}\}$) and one batch that is already present in its local cache.

Upon receiving each coded packet, Node $i$ subtracts the known (cached) batch from the linear combination, resulting in an equation that involves two unknowns. This process yields a system of three equations:

$$\begin{aligned} B_{x_i} + B_{y_i} &= A_1, \\ B_{x_i} + B_{z_i} &= A_2, \\ B_{y_i} + B_{z_i} &= A_3, \end{aligned}$$

where $A_1, A_2, A_3 \in \mathbb{F}_{2^m}$ are intermediate values computed after removing the known batch from each coded packet.

This system of equations is solvable over the finite field using standard algebraic

methods such as substitution or Gaussian elimination. Once solved, Node $i$ successfully recovers all three of its missing batches using only the received coded packets and its locally cached data. This strategy enables fully decentralized recovery and demonstrates the effectiveness of the linear-coded recovery mechanism employed in the system.

### 3.7.2 Solving the System

This system admits a unique solution over any finite field $\mathbb{F}_{2^m}$ in which division by 2 is defined (i.e., the field has characteristic not equal to 2). Using standard algebraic manipulations, the missing batches at Node $i$, specifically $B_{x_i}, B_{y_i}$, and $B_{z_i}$, can be recovered as follows:

$$B_{y_i} = \frac{A_1 + A_3 - A_2}{2},$$
$$B_{z_i} = A_3 - B_{y_i},$$
$$B_{x_i} = A_1 - B_{y_i}.$$

Here, $A_1, A_2, A_3 \in \mathbb{F}_{2^m}$ are values obtained by subtracting the known (cached) batch from each coded packet. The solution illustrates how Node $i$ can locally reconstruct all three of its missing batches using linear-coded cooperation without any additional communication with the source node.

Each operation is performed over the finite field $\mathbb{F}_{2^m}$, such as GF($2^8$), ensuring that byte-sized data values are compatible with field operations.

### 3.7.3 Example – Linear-Coded Recovery at Node 1

Consider a distributed system consisting of $K = 4$ nodes: Node 1, Node 2, Node 3, and Node 4, and $N = 6$ unique data batches $\{B_1, B_2, B_3, B_4, B_5, B_6\}$. Each batch is stored in exactly $r = 2$ nodes, according to the following placement:

† Node 1 stores $\{B_1, B_2, B_3\}$

† Node 2 stores $\{B_1, B_4, B_5\}$

† Node 3 stores $\{B_2, B_4, B_6\}$

† Node 4 stores $\{B_3, B_5, B_6\}$

Suppose the central source database is temporarily unavailable. Node 1, which holds $B_1, B_2, B_3$, must reconstruct its missing batches $B_4, B_5, B_6$ using only peer-coded transmissions.

Each peer node constructs and transmits a single coded packet by linearly combining its locally cached batches over a finite field $\mathbb{F}_{2^m}$. The packets received at Node 1 are:

$$\begin{aligned} Q_2 &= B_1 + B_4 + B_5 \quad \text{(from Node 2)} \\ Q_3 &= B_2 + B_4 + B_6 \quad \text{(from Node 3)} \\ Q_4 &= B_3 + B_5 + B_6 \quad \text{(from Node 4)} \end{aligned}$$

Node 1 subtracts its known batch from each received coded packet to isolate the

unknown terms, resulting in the following equations:

$$Q_2 - B_1 = B_4 + B_5 = A_1,$$
$$Q_3 - B_2 = B_4 + B_6 = A_2,$$
$$Q_4 - B_3 = B_5 + B_6 = A_3.$$

This yields the following system of linear equations over a finite field $\mathbb{F}_{2^m}$:

$$B_4 + B_5 = A_1,$$
$$B_4 + B_6 = A_2,$$
$$B_5 + B_6 = A_3.$$

Assuming the field supports division by 2 (i.e., its characteristic is not equal to 2), the solution is obtained using basic algebraic operations:

$$B_5 = \frac{A_1 + A_3 - A_2}{2},$$
$$B_6 = A_3 - B_5,$$
$$B_4 = A_1 - B_5.$$

This decoding process allows Node 1 to recover all three of its missing batches—$B_4$, $B_5$, and $B_6$—using only the received coded packets and its locally stored data, without querying the original data source.

This allows Node 1 to fully reconstruct its missing batches using peer transmissions only, without accessing the source. The method generalizes to any node with three missing batches, given appropriate overlap and linear independence in received coded packets.

### 3.7.4 Solving the Linear System at Node 1

Let the known batch values at Node 1 be:

$$B_1 = 10, \quad B_2 = 20, \quad B_3 = 30$$

Let the true values of the missing batches be:

$$B_4 = 40, \quad B_5 = 50, \quad B_6 = 60$$

Then the coded packets transmitted from the peers are:

$$\begin{aligned} Q_2 &= B_1 + B_4 + B_5 = 10 + 40 + 50 = 100, \\ Q_3 &= B_2 + B_4 + B_6 = 20 + 40 + 60 = 120, \\ Q_4 &= B_3 + B_5 + B_6 = 30 + 50 + 60 = 140. \end{aligned}$$

Node 1 subtracts its cached values from the received packets to isolate the unknown components:

$$\begin{aligned} A_1 &= Q_2 - B_1 = 100 - 10 = 90, \\ A_2 &= Q_3 - B_2 = 120 - 20 = 100, \\ A_3 &= Q_4 - B_3 = 140 - 30 = 110. \end{aligned}$$

This yields the following system of linear equations:

$$\begin{aligned} B_4 + B_5 &= A_1 = 90, \\ B_4 + B_6 &= A_2 = 100, \\ B_5 + B_6 &= A_3 = 110. \end{aligned}$$

Solving the system gives:

$$\begin{aligned} B_5 &= \frac{A_3 + A_1 - A_2}{2} = \frac{110 + 90 - 100}{2} = \frac{100}{2} = 50, \\ B_6 &= A_3 - B_5 = 110 - 50 = 60, \\ B_4 &= A_1 - B_5 = 90 - 50 = 40. \end{aligned}$$

Thus, Node 1 successfully reconstructs its missing data:

$$B_4 = 40, \quad B_5 = 50, \quad B_6 = 60$$

This confirms that linear-coded peer transmissions enable complete and accurate recovery without reliance on the failed source. The same decoding strategy can be applied symmetrically at the remaining nodes.

## 3.8 Efficiency of Linear-Coded Recovery

This section presents a formal comparison between two approaches: uncoded recovery and linear-coded peer recovery and proves that the linear-coded strategy is more

efficient under full source failure conditions.

Let the system be defined as follows:

- † $K$: number of nodes in the system.
- † $N$: total number of unique data batches.
- † $r$: redundancy factor (each batch is stored on exactly $r$ nodes).
- † $s$: size of each batch in bytes (or symbols over a finite field).

Each node holds $\frac{r \cdot N}{K}$ batches. Therefore, each node is missing:

$$m = N - \frac{r \cdot N}{K}.$$

batches that must be recovered from peers when the source is offline.

### 3.8.1 Uncoded Peer Recovery Cost

In uncoded peer recovery, each missing batch is transmitted individually to the nodes that require it. Let $s$ denote the size of a single data batch (e.g., in bytes). Since each of the $N$ batches is stored on $r$ nodes and needed by the remaining $K - r$ nodes, each batch must be sent $K - r$ times during the recovery process. Therefore, the total communication cost incurred by the uncoded scheme is given by:

$$C_{\text{uncoded}} = s \cdot N \cdot (K - r),$$

where:

- $s$ is the size of each batch,

- $N$ is the total number of batches,

- $K$ is the total number of nodes,

- $r$ is the replication factor (i.e., the number of nodes storing each batch).

### 3.8.2 Linear-Coded Peer Recovery Cost

In the linear-coded method, each node sends exactly one linearly coded packet to the rest of the system. This packet is constructed as a linear combination of its locally stored batches and has the same size $s$. Since each of the $K$ nodes sends one coded packet, the total communication cost is:

$$C_{\text{coded}} = s \cdot K.$$

Each node receives 3 such packets from its peers and solves a linear system (as shown in prior decoding sections) to reconstruct its missing batches.

### 3.8.3 Communication Efficiency Gain

To evaluate the benefit of linear coding, we define the communication efficiency gain as:

$$\text{Gain} = \frac{C_{\text{uncoded}}}{C_{\text{coded}}} = \frac{s \cdot N \cdot (K - r)}{s \cdot K} = \frac{N(K - r)}{K}.$$

This gain becomes more pronounced as $N$ grows relative to $K$, especially in systems with high batch redundancy and source unavailability.

### 3.8.4 Example: $K = 4, N = 6, r = 2$

Each batch is stored on 2 nodes. Therefore, each node holds 3 unique batches and is missing 3 others:

† Node 1: $B_1, B_2, B_3 \rightarrow$ needs $B_4, B_5, B_6$

† Node 2: $B_1, B_4, B_5 \rightarrow$ needs $B_2, B_3, B_6$

† Node 3: $B_2, B_4, B_6 \rightarrow$ needs $B_1, B_3, B_5$

† Node 4: $B_3, B_5, B_6 \rightarrow$ needs $B_1, B_2, B_4$

† **Uncoded recovery:** Each of the 6 batches must be sent to 2 other nodes:

$$C_{\text{uncoded}} = 6 \cdot 2 \cdot s = 12s.$$

† **Linear-coded recovery:** Each of the 4 nodes sends one coded packet:

$$C_{\text{coded}} = 4s.$$

† **Gain:**

$$\text{Gain} = \frac{12s}{4s} = 3 \quad \Rightarrow \quad 66.7\% \text{ reduction in communication cost}$$

Each node reconstructs its 3 missing batches by solving a linear system using the 3

coded packets received from its peers. This enables full recovery in a communication-efficient, decentralized manner without any access to the source database.

## 3.9 Node Failure with MySQL Online

In the event that a node (e.g., Node 1) loses its local cache but the MySQL server remains operational, the system performs a direct batch recovery. MySQL identifies which batches are missing at the failed node (e.g., $B_1, B_2, B_3$) and locates peer nodes that still hold them due to the redundancy level $r = 2$.

Instead of sending these batches itself, MySQL delegates the responsibility to the appropriate nodes:

$$\text{Node } 2 \rightarrow \text{send } B_1 \text{ to Node } 1$$

$$\text{Node } 3 \rightarrow \text{send } B_2 \text{ to Node } 1$$

$$\text{Node } 4 \rightarrow \text{send } B_3 \text{ to Node } 1$$

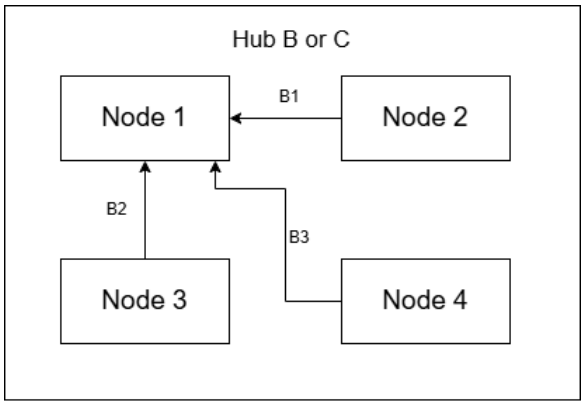



**Figure 3.2:** Nodes Communicating With Each Other in order to Get Needed Batches

This method avoids unnecessary transmissions from the server and leverages existing cached data for quick recovery.

# Chapter 4

# Implementation

The implementation of the proposed synchronization framework was carried out using a carefully selected set of technologies and tools, chosen for their performance, simplicity, and compatibility with distributed system simulation.

## 4.1 Technologies Used

Python served as the core programming language for the development of the entire synchronization framework due to its simplicity, versatility, and wide adoption in both academic and industrial settings. Its strong standard library, extensive ecosystem of third-party modules, and support for multiple programming paradigms made it particularly well-suited for modeling a modular, distributed system that required real-time data processing, communication across multiple hubs, and built-in fault tolerance mechanisms. The synchronization system was built using Python modules and packages, which helped keep different parts of the program organized and focused on their specific tasks. Each core task—such as data extraction, batch processing,

XOR-coded packet generation, caching logic, and node-to-node communication was implemented as a self-contained function or class. This modularity made it easier to simulate failure scenarios, implement fallback logic, and adjust components independently during testing and development.

Python offered solid tools for working with exact calculations and managing timestamps effectively. The decimal module was used to preserve numerical accuracy during data serialization and deserialization, particularly for values involving currency or computation-sensitive figures. Python's datetime module played a key role in accurately tracking time during synchronization processes. This helped keep timestamp-based sorting and updates consistent, whether handling batch jobs or managing data exchanges between hubs. Python was also essential in coordinating communication between the hubs. Using Flask, a lightweight Python web framework, enabled quick and efficient development of RESTful API endpoints. These endpoints made it smooth to exchange data between Hub B, Hub A, and Hub C. Combined with the requests module, this setup enabled bidirectional communication across hubs using HTTP protocols, simulating the data routing and synchronization that would occur in a larger-scale distributed deployment.

## 4.2 Databases

Two relational database management systems were employed in the implementation of the synchronization framework to simulate real-world data flow across distributed hubs and to support evaluation of both transmission logic and recovery behavior. The choice of MySQL and SQLite was based on their core advantages: MySQL for managing structured data reliably and offering flexible integration options, and SQLite for its efficiency in running simulations smoothly. MySQL was used as the

primary source database in Hub B, serving as the central repository from which the original dataset was extracted. It was selected because of its reliable performance, solid support for transactions, and its popularity across both enterprise settings and open-source communities. MySQL's structured query language (SQL) capabilities allowed for precise selection and filtering of records based on attributes such as creation timestamps, which proved essential for batch segmentation and incremental synchronization. Its ability to effectively manage relational data structures makes it well-suited for representing real-world datasets, especially when maintaining referential integrity and clear schema definitions are essential. The system integrated MySQL with the Python backend through the mysql.connector interface, allowing for smooth execution of queries and straightforward retrieval of results. After extracting records from the database, Python processed and partitioned them into batches—each representing a subset of the total dataset—to be transmitted to distributed storage nodes. These batches were stored in cache and encoded using redundancy techniques, with MySQL serving as the primary source of truth throughout the entire simulation process. In failure simulation scenarios, MySQL's absence triggered recovery operations based on cached data and coded packets, allowing for practical assessment of the system's resilience to source unavailability. Although Oracle was initially considered for its industry relevance in enterprise systems, SQLite was selected as a practical alternative during implementation due to its ease of integration, zero-configuration nature, and compatibility with the Python environment. SQLite, by contrast, was used exclusively in Hub C as a lightweight destination database. Although the original architectural intent was to synchronize data into an Oracle environment, SQLite was selected as a substitute due to its zero-configuration setup, native support in Python, and adequacy for simulating data reception and storage on the receiving end. Hub C uses SQLite to receive and store the entire dataset sent from Hub A. Upon reception, the data was reorganized into batches in preparation for future synchronization with downstream systems. The simplicity of SQLite made it particularly suitable for

localized testing and validation, especially in single-node environments where ease of deployment and minimal overhead were prioritized.

## 4.3 Dataset Description

The dataset used for evaluation was sourced from Kaggle's *Banking Dataset for Marketing Targets*[41]. It contains 4,521 records and 17 fields, including attributes such as age, occupation, marital status, loan status, contact duration, and subscription outcome. Its structured format reflects typical relational database schemas and supports simulation of distributed synchronization tasks.

Despite its moderate size, the dataset was chosen to validate system logic under controlled conditions. The architecture is scalable: batch count $N$, node count $K$, and redundancy level $r$ can be increased proportionally to support larger datasets. These experiments function as minimal working examples, confirming the system's scalability, fault tolerance, and recovery behavior under varying parameters.

## 4.4 Web Framework and Communication

The communication layer is an important part of the synchronization framework, allowing efficient data sharing and coordination across distributed hubs. The system used Flask. Flask was selected for its simplicity, modular structure, and its suitability for building RESTful APIs, all of which aligned well with the project's goal of simulating loosely coupled, service-based communication between distributed components. Flask was used to expose API endpoints at each hub to handle data reception,

validation, and forwarding. Endpoints such as /receive.from.b, /send.to.c, and /process were implemented to represent the various stages of the synchronization pipeline. These endpoints allowed hubs to transmit data payloads, confirm receipt, initiate validation routines, and coordinate transfer logic, all using simple HTTP requests. By simulating inter-hub operations over HTTP, Flask enabled the system to replicate the asynchronous, decentralized behavior common in real-world microservice architectures and distributed database systems. The web communication layer was built using Python's requests library, offering an easy-to-use interface for sending outbound HTTP requests. This was essential for simulating push-based synchronization, where one hub proactively sends data to another, rather than relying on polling or manual data pulls. Each transmission involved packaging the batch data or full dataset as a JSON-formatted HTTP request body, complete with serialization of timestamped records, decimal values, and metadata.

## 4.5 System Design Features

To replicate more realistic distributed conditions, the system integrated basic communication protocols such as request retries, error detection, and acknowledgment responses. Each HTTP interaction between hubs involved a quick status check to confirm the data was received correctly before moving on to the next step. In the case of failed requests—due to simulated downtime, timeouts, or data corruption—the sending hub could retry the operation, or in certain scenarios, activate recovery logic using cached or redundant data. This behavior aligns closely with real-world distributed systems, where ensuring fault tolerance and reliable message delivery are essential priorities. The Flask-based architecture also enabled clean modular separation between components. Each hub operated its own Flask server instance, complete

with a specific set of routes and logic designed to handle its part in the synchronization process. Hub B, for instance, exposed endpoints specifically for sending initial batch data and coded packets, while Hub A's routes managed system health checks, database forwarding, and node status monitoring. Hub C processed and reorganized the incoming data to set it up properly for the next steps in synchronization. This separation of responsibilities made it easier to test each hub independently and ensured that system behavior remained predictable and scalable.

The combination of Flask and HTTP-based communication offered a optimized, flexible, and scalable foundation for the synchronization framework. It enabled reliable, testable, and asynchronous interaction between distributed hubs, reflecting real-world practices in microservices and distributed database environments. Combined with Python's built-in networking and serialization capabilities, this web framework allowed for clean orchestration of inter-node communication while preserving system flexibility, simplicity, and robustness.

**Algorithm 1** Hub B Sync and Recovery Distribution Procedure

Fetch data from MySQL: `data`, `mysql_failed` ← `fetch_data()`
**if** no data and not in recovery **then**
    Wait and retry
**else if** in recovery **then**
    **if** MySQL is down **then**
        **for** each Node $i$ **do**
            Identify missing batches $M_i$
            Receive $K-1$ coded packets $Q_j = B_{a_j} + B_{b_j} + B_{c_j}$ over $\mathbb{F}_{2^m}$
            Subtract known terms, solve for $M_i$
        **end for**
    **else**
        Identify all failed nodes (e.g., Node 1, Node 2, Node 3)and their missing batches
        **for** each missing batch **do**
            Request it from a peer holding it
        **end for**
    **end if**
**end if**
Split data into $N$ batches and distribute to $K$ nodes
Generate and assign XOR-coded packets
Transmit full DB from Node 1 to Hub A
Wait for next sync cycle

**Description.** The Hub B algorithm is responsible for pulling data from the primary MySQL database, splitting it into batches, and distributing both raw and XOR-coded data across the storage nodes. It ensures distribution during normal operation and activates fault-tolerant mechanisms during failure scenarios.

**Algorithm 2** Hub A Sync and Monitoring Procedure

```
1: while True do
2:     Check status of MySQL, Hub B/C (/ping), and local SQLite
3:     if MySQL down and recovery not active then
4:         Trigger recovery mode on Hub B
5:     else if MySQL restored and recovery was active then
6:         Reset recovery mode on Hub B
7:     end if
8:     Wait before next status check
9: end while
10: function RECEIVE_FROM_B(PAYLOAD)
11:     Log and forward payload to Hub C
12: end function
13: function RECEIVE_FROM_C(PAYLOAD)
14:     Log and forward payload to Hub B
15: end function
16: function SEND_TO_HUB_B_FROM_C(DATA)
17:     Forward full dataset from Hub C to Hub B
18: end function
19: function SEND_TO_HUB_C(DATA)
20:     if Hub C and SQLite are alive then
21:         Build and send payload with sync metadata
22:     else
23:         Skip transmission
24:     end if
25: end function
```

**Description.** Hub A serves as an intermediary relay and system monitor. It forwards

synchronization data between Hubs B and C, ensuring consistent propagation of batch updates. Additionally, it continuously monitors the availability of MySQL, Hub B, Hub C, and the simulated SQLite database. When MySQL becomes unavailable, Hub A triggers recovery mode in Hub B. Once MySQL is restored, it resets Hub B to resume normal operations.

**Algorithm 3** Hub C Synchronization and Recovery Procedure

**while** True **do**
    Fetch records from local SQLite
    **if** data is available **then**
        Split into $N$ batches and distribute to $K$ nodes
        Generate and assign XOR-coded packets
        Extract full data from one node (e.g., Node k)
        Package with sync metadata and send to Hub A
    **else if** SQLite is down **then**
        **for** each Node $i$ **do**
            Identify missing batches $M_i$
            Receive $K-1$ coded packets $Q_j = B_{a_j} + B_{b_j} + B_{c_j}$ over $\mathbb{F}_{2^m}$
            Subtract known components to isolate unknowns
            Solve the linear system over $\mathbb{F}_{2^m}$
        **end for**
    **end if**
**end while**
**if** a node fails but SQLite is online **then**
    Identify missing batches
    **for** each batch **do**
        Request it from a peer node
    **end for**
**end if**
**function** UPLOAD_FROM_A(`payload`)
    Extract data and metadata
    Store into local SQLite
    Log sync metrics
**end function**

**Description.** The Hub C algorithm handles synchronization and storage of data using a local SQLite database. It acts as a secondary source and supports bidirectional sync with Hub A. Hub C retrieves records, splits them into batches, generates XOR-coded packets, and distributes both raw and encoded data across its nodes.

## 4.6 Synchronization Modes: Direct, Partial, and Full

To evaluate the efficiency of the proposed system, three synchronization strategies are compared: direct synchronization, partial synchronization, and full synchronization via coded caching.

**Direct synchronization** refers to transmitting the entire dataset to each node without batch strategy and caching. This approach incurs the highest communication and memory overhead, as every node receives and stores a full copy of the data.

**Partial synchronization** transmits data in batches using logical placement but without caching or coded recovery. Nodes request missing data on demand, and those are sent directly from peers or the source, without redundancy or optimization.

**Full synchronization** is implemented using the proposed coded caching method. Data is divided into batches, placed redundantly using combinatorial logic, and transmitted as XOR-coded packets. This approach reduces network traffic and enables recovery without requiring full retransmissions.

# Chapter 5

# Results and Evaluations

This section presents the evaluation of the proposed synchronization system under varying parameters. The goal is to assess how well the system performs as the number of nodes and redundancy levels change, particularly in terms of transmission time and overhead. Through a series of experiments, we analyze the scalability and efficiency of the coded synchronization mechanism, highlighting its robustness under different configurations. The system is tested under varying configurations, including changes in node count $K$, redundancy $r$, and batch count $N$. Three synchronization modes are compared: direct, partial, and full coded synchronization. Evaluation metrics include transmission time, CPU and memory usage, and recovery efficiency under failure scenarios.

## 5.1 Evaluation Across Varying Node Counts and Redundancy Levels

Figure 5.1 captures how the system responds to increasing batch volumes in a configuration with $K = 4$ nodes and a redundancy level of $r = 2$. As the number of batches increases from 6 to 24, the transmission time shows only a modest rise, from approximately 3.05 to 3.66 seconds. This increase highlights the system's ability to handle higher data loads without compromising synchronization efficiency. The additional time is primarily attributed to the rise in XOR-coded packet generation and delivery, yet the performance curve remains smooth and controlled.

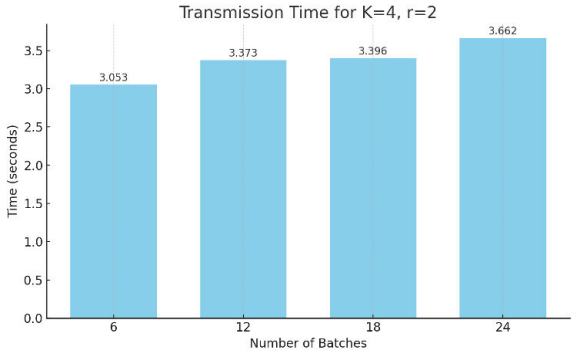



**Figure 5.1:** Transmission Time Comparison for $K = 4$ and $r = 2$ values

In Figure 5.2 the transmission shows a performance for a setup with $K = 5$ nodes and redundancy $r = 2$, across batch sizes ranging from 10 to 40. Transmission time increases slightly from 3.57 to 3.84 seconds, it's an incremental rise similar to that observed in the $K = 4$ case. As the number of nodes grows, the system continues to handle increased batch traffic with minimal overhead. The slight uptick reflects the expected cost of generating more XOR-coded packets, yet the synchronization process remains efficient and predictable.

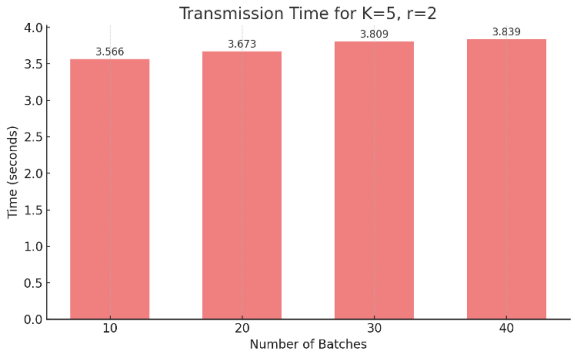



**Figure 5.2:** Transmission Time Comparison for $K = 5$ and $r = 2$ values

Figure 5.3 presents the transmission time for a larger configuration with $K = 7$ nodes and redundancy $r = 2$, evaluated over batch sizes from 21 to 84. Interestingly, the transmission time remains nearly flat, rising only slightly from 4.15 to 4.21 seconds. Compared to the $K = 4$ and $K = 5$ cases, this setup handles a significantly higher data volume with almost no added latency. Even as both node count and batch count increase, the system sustains its performance with impressive consistency.

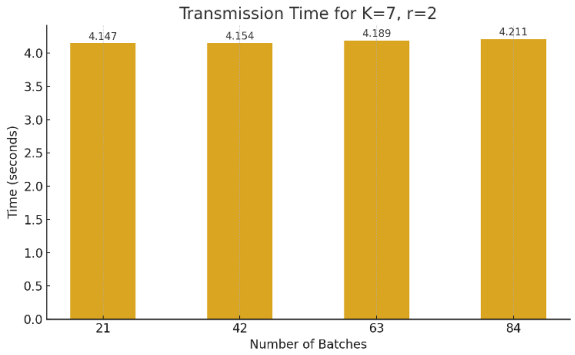



**Figure 5.3:** Transmission Time Comparison for $K = 7$ and $r = 2$ values

Figure 5.4 displays the transmission time for $K = 4$ nodes under a higher redundancy setting of $r = 3$, with batch counts ranging from 4 to 16. The increase in transmission time is rising from 3.10 to 3.20 seconds, demonstrating that the added redundancy

introduces only a negligible overhead. Compared to the $r = 2$ scenario in Figure 5.1 with the same node count, the system maintains a similarly stable trend, indicating that increased fault tolerance does not compromise synchronization efficiency.

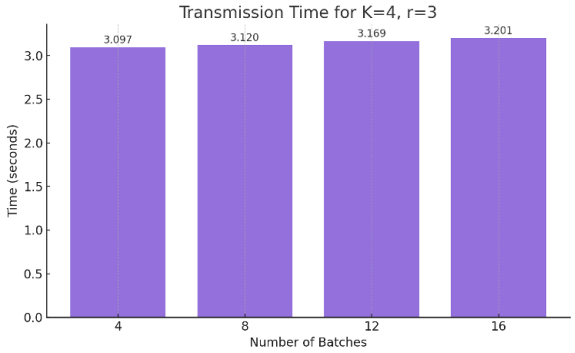



**Figure 5.4:** Transmission Time Comparison for $K = 4$ and $r = 3$ values

Figure 5.5 presents the transmission performance for a configuration with $K = 5$ nodes and redundancy level $r = 3$, across batch sizes from 10 to 40. The transmission time remains tightly clustered between 3.25 and 3.30 seconds, showing almost no upward trend. The results suggest that the system absorbs the cost of higher redundancy with minimal performance trade-off.

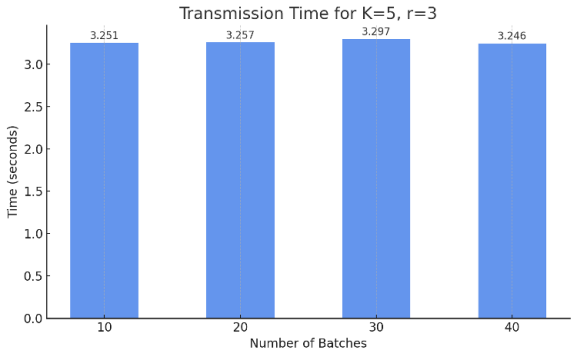



**Figure 5.5:** Transmission Time Comparison for $K = 5$ and $r = 3$ values

Figure 5.6 shows the transmission time for a system with $K = 6$ nodes and redundancy

level $r = 3$, tested across 20 to 80 batches. The synchronization time gradually increases from 3.22 to 3.48 seconds, reflecting modest growth. Although the overhead is slightly more noticeable than in the $K = 5$ case, the rise remains linear and controlled.

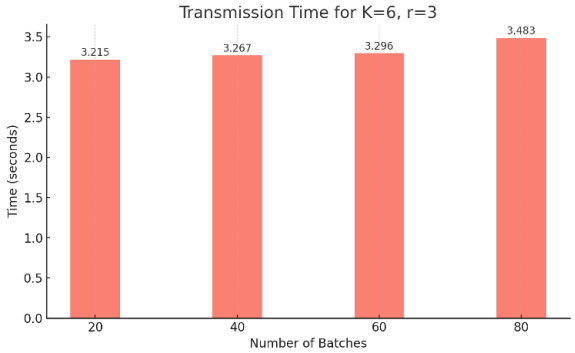



**Figure 5.6:** Transmission Time Comparison for $K = 6$ and $r = 3$ values

Figure 5.7 illustrates the transmission time for $K = 7$ nodes with redundancy level $r = 3$, evaluated over batch counts ranging from 35 to 140. The time increases steadily from 3.82 to 4.27 seconds, marking the most noticeable increase among the configurations tested. Despite the heavier workload, growth remains smooth and predictable, suggesting that the system scales reliably even at higher node and batch counts.

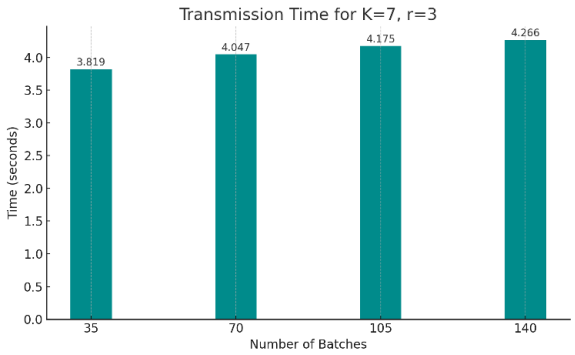



**Figure 5.7:** Transmission Time Comparison for $K = 7$ and $r = 3$ values

## 5.2 Recovery Time Simulation For Different Scenarios

Table 5.1 presents the peer-based recovery times under the linear-coded recovery strategy for a configuration with $K = 4$ nodes, $N = 6$ data batches, and a redundancy factor of $r = 2$. In this scenario, each node reconstructs its missing data by solving a system of linear equations derived from coded packets received from its peers.

**Table 5.1**
Linear-Coded Recovery Time for $K = 4$, $N = 6$, $r = 2$

| **Node** | **Recovery Time (s)** |
|---|---|
| Node 1 | 0.031 |
| Node 2 | 0.019 |
| Node 3 | 0.028 |
| Node 4 | 0.017 |

The recovery times show minor but notable variation: Node 1 completes in 0.031 seconds, whereas Node 2, Node 3, and Node 4 complete in 0.019, 0.028, and 0.017 seconds, respectively. The slightly higher recovery time for Node 1 may be attributed to a more complex decoding path or delayed arrival of the required coded packets due to its role in multiple XOR groupings. Node 4 consistently shows the fastest recovery, potentially indicating a more favorable packet overlap or less computational load.

These results confirm that the proposed system can reliably and efficiently recover lost data using peer-coded transmissions, without requiring reconnection to the original MySQL source. The sub-second recovery latency across all nodes demonstrates

the practicality of the coded approach in real-time or near-real-time distributed environments.

In Table 5.2, the system is evaluated with $K = 4$ nodes managing $N = 12$ data batches under a redundancy factor of $r = 2$. Recovery times range from 0.043 seconds (Node 4) to 0.054 seconds (Node 1), reflecting slight variations that may be attributed to packet arrival sequencing, decoding load, or node placement within the coded groupings.

**Table 5.2**
Linear-Coded Recovery Time for $K = 4$, $N = 12$, $r = 2$

| **Node** | **Recovery Time (s)** |
|---|---|
| Node 1 | 0.054 |
| Node 2 | 0.048 |
| Node 3 | 0.052 |
| Node 4 | 0.043 |

Despite the increased data volume compared to the $N = 6$ configuration (Table 5.1), all nodes successfully reconstruct their missing data using only peer transmissions, without any reliance on the MySQL source. This confirms that the system's linear-coded recovery mechanism maintains consistent performance and fault tolerance even as the batch count doubles, demonstrating its scalability under growing data loads.

Table 5.3 reports recovery times for a configuration involving $K = 4$ nodes and $N = 18$ data batches with redundancy $r = 2$. Nodes 1, 2, and 4 complete recovery within a tight range of 0.055 to 0.083 seconds. However, Node 3 exhibits a significantly longer recovery time of 0.168 seconds—nearly double that of its peers.

**Table 5.3**
Linear-Coded Recovery Time for $K = 4$, $N = 18$, $r = 2$

| **Node** | **Recovery Time (s)** |
|---|---|
| Node 1 | 0.058 |
| Node 2 | 0.055 |
| Node 3 | 0.095 |
| Node 4 | 0.083 |

This anomaly likely reflects external runtime factors such as unbalanced packet assignments, increased decoding complexity due to its position in multiple XOR groups, or temporary network latency during peer communication. While the system ultimately achieves full recovery for all nodes, this case highlights how batch volume and coded group dynamics can introduce load imbalances. Such behavior reinforces the importance of optimizing placement strategies and decoding schedules, particularly in dense or high-throughput deployments.

Table 5.4 presents recovery times under the linear-coded strategy for a configuration with $K = 4$ nodes handling $N = 24$ data batches and a redundancy level of $r = 2$. Recovery performance remains well balanced across nodes, with completion times ranging from 0.089 to 0.110 seconds.

**Table 5.4**
Linear-Coded Recovery Time for $K = 4$, $N = 24$, $r = 2$

| **Node** | **Recovery Time (s)** |
|---|---|
| Node 1 | 0.089 |
| Node 2 | 0.110 |
| Node 3 | 0.092 |
| Node 4 | 0.097 |

Node 2 exhibits the highest recovery time at 0.110 seconds—a modest deviation that may reflect temporary decoding overhead or slight variation in packet complexity. Importantly, no extreme outliers are observed, indicating that the system scales predictably even as batch volume continues to increase. These results reinforce the framework's stability and effectiveness in handling larger datasets without compromising fault-tolerant recovery.

The recovery results in Table 5.5 reveal consistently uniform performance across all five nodes, with recovery times tightly clustered between 0.057 and 0.065 seconds.

**Table 5.5**
Linear-Coded Recovery Time for $K = 5$, $N = 10$, $r = 2$

| **Node** | **Recovery Time (s)** |
|---|---|
| Node 1 | 0.061 |
| Node 2 | 0.058 |
| Node 3 | 0.061 |
| Node 4 | 0.065 |
| Node 5 | 0.057 |

This minimal variation—on the order of milliseconds—demonstrates that the peer-to-peer coded recovery mechanism scales effectively in a five-node configuration. Operating with $K = 5$, $N = 10$ data batches, and a redundancy factor of $r = 2$, the system maintains low-latency, balanced recovery across all participants. The absence of significant outliers confirms that increasing the number of nodes does not introduce instability or performance degradation. Instead, it reinforces the design's robustness and ability to preserve fault tolerance at scale.

**Table 5.6**
Linear-Coded Recovery Time for $K = 5$, $N = 20$, $r = 2$

| Node | Recovery Time (s) |
|---|---|
| Node 1 | 0.111 |
| Node 2 | 0.112 |
| Node 3 | 0.117 |
| Node 4 | 0.125 |
| Node 5 | 0.111 |

While overall recovery times remain closely grouped in Table 5.6, Node 4 stands out with a slightly elevated duration of 0.125 seconds — just above its peers, who fall between 0.111 and 0.119 seconds. This test, conducted with $K = 5$ nodes, $N = 20$ data batches, and redundancy level $r = 2$, hints at possible fluctuations in decoding load or encoded packet complexity for individual nodes. Even with this minor outlier, the results confirm the system's ability to maintain synchronized recovery across a growing dataset.

**Table 5.7**
Linear-Coded Recovery Time for $K = 5$, $N = 30$, $r = 2$

| Node | Recovery Time (s) |
|---|---|
| Node 1 | 0.161 |
| Node 2 | 0.150 |
| Node 3 | 0.172 |
| Node 4 | 0.181 |
| Node 5 | 0.157 |

This table 5.7 illustrates the peer-based recovery times using the proposed linear-coded strategy, assuming $K = 5$ nodes, $N = 30$ data batches, and a redundancy

factor of $r = 2$. As shown, Node 1 completes recovery in 0.168 seconds, Node 2 in 0.159 seconds, Node 3 in 0.172 seconds, Node 4 in 0.181 seconds, and Node 5 in 0.157 seconds. While the times are relatively close, Node 4 exhibits slightly higher recovery latency, which may be attributed to decoding overhead or network timing variation.

**Table 5.8**
Linear-Coded Recovery Time for $K = 5$, $N = 40$, $r = 2$

| **Node** | **Recovery Time (s)** |
|---|---|
| Node 1 | 0.211 |
| Node 2 | 0.205 |
| Node 3 | 0.219 |
| Node 4 | 0.224 |
| Node 5 | 0.203 |

As batch volume increases to $N = 40$, Table 5.8 shows a predictable rise in recovery duration across all five nodes. In this configuration with $K = 5$ and redundancy level $r = 2$, decoding times fall within a range from 0.203 to 0.224 seconds, representing approximately double the values observed in the $N = 20$ test. This increase is expected, as each node must process a greater number of missing batches and solve more complex linear systems.

## 5.3 Comparison of Direct, Partial, and Coded Transmission Methods ($K = 4$, $N = 6$ and $r = 2$ values)

Figure 5.8 compares the transmission performance of three synchronization strategies under a fixed configuration of $K = 4$ nodes, $N = 6$ data batches, and a redundancy factor of $r = 2$. The three methods—direct placement, partial send, and coded caching—differ in how they distribute data from the source to the nodes.

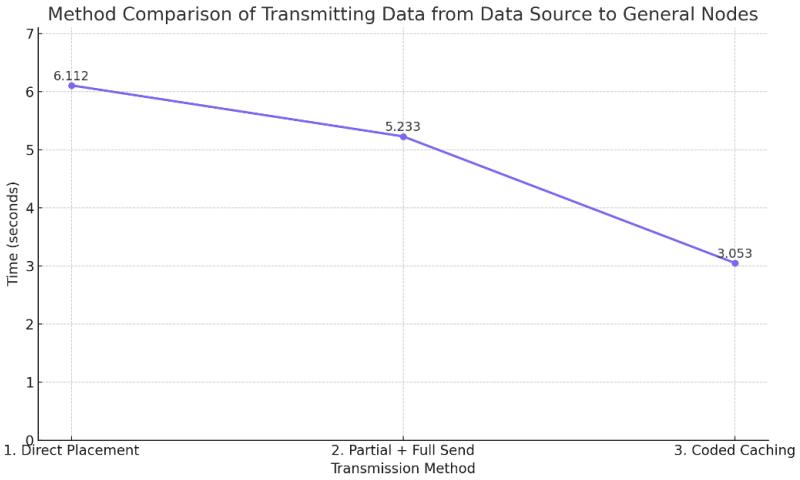



**Figure 5.8:** Method Comparison of Transmitting Data from Data Source to General Nodes

In the direct placement method, each node receives all required batches independently, resulting in the highest transmission cost due to redundant data transfers. The partial send approach improves this by dividing delivery into two steps, first assigning initial batches and then responding to missing batch requests, moderately reducing overhead. The coded caching strategy provides the most efficient outcome: it uses four XOR-coded packets to satisfy the data needs of all nodes, leveraging

combinatorial redundancy to reduce network load.

This comparison highlights the substantial communication savings introduced by coded synchronization, particularly in systems with high node overlap and shared batch dependencies. The result demonstrates the framework's ability to reduce synchronization time while maintaining fault tolerance and recovery flexibility.

### 5.3.1 Resource Usage (CPU, Memory) ($K = 4$, $N = 6$ and $r = 2$ values)

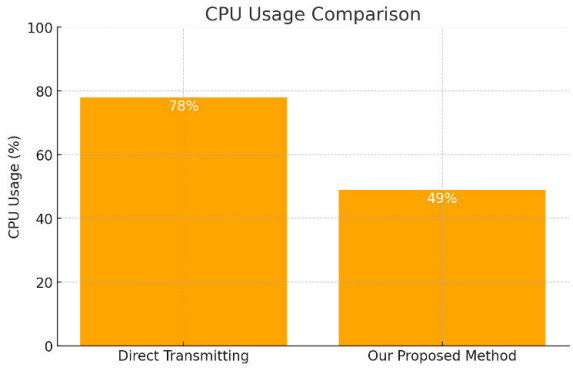



**Figure 5.9:** CPU Usage

Figure 5.9 compares the CPU usage percentage of the full synchronization system during one complete synchronization cycle. The first bar, representing "Direct Transmitting", shows a CPU usage of 78 percent. This approach involves sending the entire dataset to each node without batching, redundancy and caching. The high CPU usage reflects the big processing load required, transmission management, and memory operations during full-dataset replication across multiple nodes.

The second bar, labeled as "Our Proposed Method", demonstrates a lower CPU usage

of 49 percent. This reduction comes from transmitting data in batches, incorporating redundancy, and recovering locally by using cached batches. Since each node is only responsible for processing a subset of the data and can reconstruct the rest without additional transmissions, the method decreases CPU workload, especially under larger-scale deployments. The 29 percent reduction in CPU utilization illustrates the computational efficiency of the proposed design.

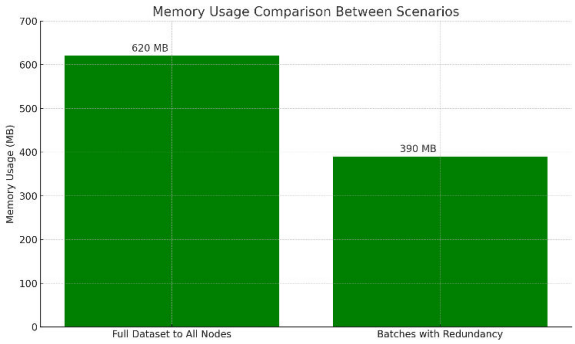



**Figure 5.10:** Memory Usage Comparison Between Scenarios($K = 4$, $N = 6$ and $r = 2$ values)

Figure 5.10 presents an analysis of memory consumption under two different synchronization strategies: a traditional direct approach versus the proposed method of the full synchronization system during one complete synchronization cycle. The first bar, "Full Dataset to All Nodes", indicates a memory usage of 620 MB. This approach involves transmitting the complete dataset to each node individually, leading to high memory consumption due to direct data duplication and repeated operations. System memory is heavily utilized.

In contrast, the second bar, "Batches with Redundancy", shows lower memory usage of 390 MB. This reduction is attributed to the system's design, which divides the dataset into smaller, manageable batches distributed across nodes with a redundancy factor.

The comparison demonstrates a 37 percent decrease in memory usage with the proposed method. This finding confirms that batching and redundancy mechanisms not only improve synchronization and fault tolerance but also optimize memory usage, making the system more suitable for environments where memory resources are limited or shared with other applications.

## 5.4 Network Performance

Figure 5.11 presents the effect of increasing network latency on total transmission time for two synchronization strategies: a conventional method without caching and the proposed coded caching-based method.

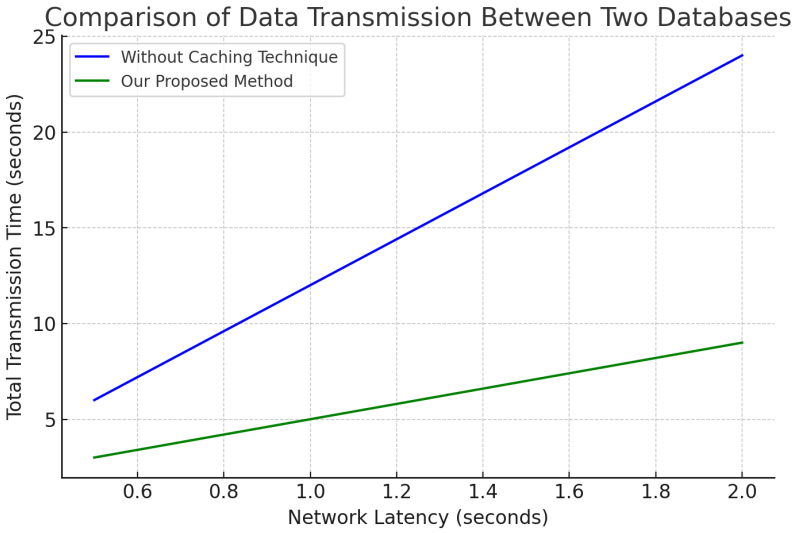



**Figure 5.11:** Comparison of Data Transmission Between Two Databases($K = 4$, $N = 6$ and $r = 2$ values)

The blue curve, labeled "Without Caching Technique," exhibits a linear increase in transmission time as network latency grows. This strategy transmits each dataset in its entirety to every destination, leading to cumulative delays across batches and

nodes—particularly in high-latency environments.

In contrast, the green curve, representing "Our Proposed Method," demonstrates significantly improved efficiency. While latency still impacts transmission time, the total duration remains consistently lower due to reduced communication overhead. This advantage stems from the use of cached placement, batching, and XOR-coded packets, which minimize the number of transmissions required for complete data distribution.

Quantitatively, for the configuration $K = 4$, $N = 6$, and $r = 2$, the proposed method achieves synchronization in 3.053 seconds, compared to 6.112 seconds using the uncached method—nearly a 50% improvement. These results confirm the method's resilience to network delay and its capacity to maintain efficient synchronization performance under variable network conditions.

# Chapter 6

# Conclusion

## 6.1 Summary of Findings

This research presents a practical database synchronization framework tailored for distributed environments. By combining batching, XOR-based coded redundancy, and decentralized recovery mechanisms, the system aims to reduce transmission overhead while maintaining data availability, even in the event of failures.

The experimental results indicate that the proposed approach can significantly reduce resource usage and transmission time compared to full-dataset replication. While the improvements are encouraging, such as a reduction in CPU and memory usage and consistent recovery performance, the system's design remains relatively adaptable. Recovery during simulated MySQL failures was handled effectively, and synchronization time scaled predictably with increasing batch volume and node count.

Although these outcomes are promising, this work represents a foundational step. Future extensions may explore more advanced coding schemes, dynamic reconfiguration under node churn, or integration with real-time consistency requirements. Collectively, these findings affirm that the framework not only achieves its design goals of efficiency and resilience, but also offers a scalable solution suitable for real-world distributed environments where performance and fault tolerance are critical.

## 6.2 Recommendations For Future Work

While the proposed synchronization framework achieved its objectives in terms of performance, fault tolerance, and resource efficiency, several opportunities remain for future exploration. Incorporating technologies such as Apache Kafka or RabbitMQ could enable continuous data updates rather than periodic batch transmissions, allowing the framework to support low-latency applications and live data processing scenarios.

Future work should focus on evaluating the proposed system using larger and more diverse datasets to better understand its scalability and robustness. Additionally, benchmarking against established synchronization frameworks would provide clearer insights into the system's relative performance, efficiency, and applicability in real-world environments. Such comparisons may also highlight areas for further optimization and refinement.

Implementing security measures like encrypted data transfer, strict access controls, and tamper detection within the synchronization process helps ensure the system is capable of handling sensitive or regulated environments effectively. This would involve implementing secure communication protocols and verification mechanisms.

Finally, the system could be extended into a containerized deployment model using Docker and Kubernetes, enabling scalability testing under cloud-native conditions. This kind of architecture makes it easier to deploy in multi-tenant setups and enables features like automatic scaling, health monitoring, and smooth recovery processes.

# References


[1] S. Browne, "Communication and synchronization issues in distributed multimedia database systems," in *Advanced Database Systems*, ser. Lecture Notes in Computer Science, N. R. Adam and B. K. Bhargava, Eds. Berlin, Heidelberg: Springer, 1993, vol. 759, pp. 259–277. [Online]. Available: https://doi.org/10.1007/3-540-57507-3_19

[2] N. J. George, "Optimizing hybrid and multi-cloud architectures for real-time data streaming and analytics: Strategies for scalability and integration," *World Journal of Advanced Engineering Technology and Sciences*, vol. 7, no. 1, pp. 174–185, 2022. [Online]. Available: https://doi.org/10.30574/wjaets.2022.7.1.0087

[3] "A database synchronization algorithm for mobile devices," *IEEE Journals & Magazine*, May 1 2010, accessed via IEEE Xplore. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/5505945

[4] M. Ahluwalia, R. Gupta, A. Gangopadhyay, Y. Yesha, and M. McAllister, "Target-based database synchronization," in *Proceedings of the 2010 ACM Symposium on Applied Computing*. New York, NY, USA: Association for Computing Machinery, 2010. [Online]. Available: https://doi.org/10.1145/1774088.1774443

[5] G. Schlageter, "Process synchronization in database systems," *ACM*

*Transactions on Database Systems*, vol. 3, no. 3, pp. 284–306, 1978. [Online]. Available: https://doi.org/10.1145/320263.320279

[6] K. Kottursamy, G. Raja, J. Padmanabhan, and V. Srinivasan, "An improved database synchronization mechanism for mobile data using software-defined networking control," *Computers & Electrical Engineering*, vol. 57, pp. 93–103, 2017. [Online]. Available: https://doi.org/10.1016/j.compeleceng.2016.01.019

[7] M. M. Pereira and E. M. Frazzon, "A data-driven approach to adaptive synchronization of demand and supply in omni-channel retail supply chains," *International Journal of Information Management*, vol. 57, p. 102165, 2020. [Online]. Available: https://doi.org/10.1016/j.ijinfomgt.2020.102165

[8] G. Deng, Y. Wang, Y. Shi, M. Zhang, J. Fang, and J. Zhang, "Research and implementation of heterogeneous database synchronization technology based on cloud-native architecture," *Proceedings of SPIE*, vol. 30, p. 43, 2024. [Online]. Available: https://doi.org/10.1117/12.3024750

[9] M. Li and G. Q. Huang, "Production-intralogistics synchronization of industry 4.0 flexible assembly lines under graduation intelligent manufacturing system," *International Journal of Production Economics*, vol. 241, p. 108272, 2021. [Online]. Available: https://doi.org/10.1016/j.ijpe.2021.108272

[10] R. Ayman, Suparlan, and J. Wahyudi, "Design of video transmission with wi-fi on runway inspection using jetson nano and data synchronization," *International Journal of Advanced Computer Science*, vol. 7, n.d.

[11] K. Nakatani, T. Chuang, and D. Zhou, "Data synchronization technology: Standards, business values and implications," *Communications of the Association for Information Systems*, vol. 17, pp. pp–pp, 2006. [Online]. Available: https://doi.org/10.17705/1CAIS.01744

[12] G. Chen, J. Xia, J. H. Park, H. Shen, and G. Zhuang, "Sampled-data synchronization of stochastic markovian jump neural networks with time-varying delay," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 33, no. 8, pp. 3829–3841, 2022. [Online]. Available: https://doi.org/10.1109/TNNLS.2021.3054615

[13] P. Detti, G. Vatti, and G. Zabalo Manrique de Lara, "Eeg synchronization analysis for seizure prediction: A study on data of noninvasive recordings," *Processes*, vol. 8, no. 7, p. 846, 2020. [Online]. Available: https://doi.org/10.3390/pr8070846

[14] C. Giannoula *et al.*, "Syncron: Efficient synchronization support for near-data-processing architectures," in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2021, pp. 263–276. [Online]. Available: https://doi.org/10.1109/HPCA51647.2021.00031

[15] Y. Han, D. Niyato, C. Leung, C. Miao, and D. I. Kim, "A dynamic resource allocation framework for synchronizing metaverse with iot service and data," in *ICC 2022 - IEEE International Conference on Communications*, 2022, pp. 1196–1201. [Online]. Available: https://doi.org/10.1109/ICC45855.2022.9838422

[16] S. Khriji, Y. Benbelgacem, R. Chéour, D. E. Houssaini, and O. Kanoun, "Design and implementation of a cloud-based event-driven architecture for real-time data processing in wireless sensor networks," *The Journal of Supercomputing*, vol. 78, no. 3, pp. 3374–3401, 2021. [Online]. Available: https://doi.org/10.1007/s11227-021-03955-6

[17] S. Butler and S. Butler, "How to use google backup and sync to backup your hard drive," February 1 2025. [Online]. Available: https://helpdeskgeek.com/how-to-use-google-backup-and-sync-to-backup-your-hard-drive/

[18] G. Yang, "Data synchronization for integration systems based on trigger," in *2010 2nd International Conference on Signal Processing Systems*. Dalian, China: IEEE, 2010, pp. V3–310–V3–312. [Online]. Available: https://doi.org/10.1109/ICSPS.2010.5555804

[19] R. Zhang, D. Zeng, J. H. Park, H.-K. Lam, and X. Xie, "Fuzzy sampled-data control for synchronization of t–s fuzzy reaction–diffusion neural networks with additive time-varying delays," *IEEE Transactions on Cybernetics*, vol. 51, no. 5, pp. 2384–2397, 2021. [Online]. Available: https://doi.org/10.1109/TCYB.2020.2996619

[20] D. of Video Transmission with Wi-Fi on Runway Inspection using Jetson Nano and D. Synchronization, *Indonesian Journal of Education Mathematical Science*, vol. 5, no. 1, pp. 1–11, 2024. [Online]. Available: https://doi.org/10.30596/ijems.v5i1.16154

[21] Z. Li, W. Qin, L. Zhang *et al.*, "Data heterogeneity-robust federated learning via group client selection in industrial iot," *IEEE Internet of Things Journal*, vol. 9, no. 18, pp. 17 844–17 857, 2022. [Online]. Available: https://doi.org/10.1109/JIOT.2022.3161943

[22] J. Ray, "A practical approach to solving data synchronization problems," May 16 2023. [Online]. Available: https://clearinsights.io/blog/a-practical-approach-to-solving-data-synchronization-problems/

[23] GeeksforGeeks, "Troubleshoot data synchronization with salesforce crm," December 17 2024. [Online]. Available: https://www.geeksforgeeks.org/software-engineering/troubleshoot-data-synchronization-with-salesforce-crm/

[24] D. Lee, S. H. Lee, N. Masoud, Krishnan, and V. C. Li, "Integrated digital twin and blockchain framework to support accountable information sharing in

construction projects," *Automation in Construction*, vol. 127, p. 103688, 2021. [Online]. Available: https://doi.org/10.1016/j.autcon.2021.103688

[25] G. Arnulfo, S. Wang, V. Myrov *et al.*, "Long-range phase synchronization of high-frequency oscillations in human cortex," *Nature Communications*, vol. 11, p. 5363, 2020. [Online]. Available: https://doi.org/10.1038/s41467-020-18975-8

[26] M. Zhang, V. Lehman, and L. Wang, "Scalable name-based data synchronization for named data networking," in *IEEE INFOCOM 2017 - IEEE Conference on Computer Communications*. Atlanta, GA, USA: IEEE, 2017, pp. 1–9. [Online]. Available: https://doi.org/10.1109/INFOCOM.2017.8057193

[27] Y. Jiang, M. Li, M. Li, X. Liu, R. Y. Zhong, W. Pan, and G. Q. Huang, "Digital twin-enabled real-time synchronization for planning, scheduling, and execution in precast on-site assembly," *Automation in Construction*, vol. 141, p. 104397, 2022. [Online]. Available: https://doi.org/10.1016/j.autcon.2022.104397

[28] J. C. Kirchhof, J. Michael, B. Rumpe, S. Varga, and A. Wortmann, "Model-driven digital twin construction," in *Proceedings of the ACM/IEEE 23rd International Conference on Model Driven Engineering Languages and Systems (MODELS)*, 2020, pp. 90–101. [Online]. Available: https://doi.org/10.1145/3365438.3410941

[29] S. Li, Y. Zhao, R. Varma, O. Salpekar, P. Noordhuis, T. Li, A. Paszke, J. Smith, B. Vaughan, P. Damania, and S. Chintala, "Pytorch distributed: Experiences on accelerating data parallel training," 2020, pyTorch.

[30] H.-B. Zeng, K. L. Teo, Y. He, H. Xu, and W. Wang, "Sampled-data synchronization control for chaotic neural networks subject to actuator saturation," *Neurocomputing*, vol. 260, pp. 25–31, 2017. [Online]. Available: https://doi.org/10.1016/j.neucom.2017.02.063

[31] S.-P. Xiao, H.-H. Lian, K. L. Teo, H.-B. Zeng, and X.-H. Zhang, "A new lyapunov functional approach to sampled-data synchronization control for delayed neural networks," *Journal of the Franklin Institute*, vol. 355, no. 17, pp. 8857–8873, 2018. [Online]. Available: https://doi.org/10.1016/j.jfranklin.2018.09.022

[32] H. Li, C. Li, D. Ouyang, and S. K. Nguang, "Impulsive synchronization of unbounded delayed inertial neural networks with actuator saturation and sampled-data control and its application to image encryption," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 32, no. 4, pp. 1460–1473, 2021. [Online]. Available: https://doi.org/10.1109/TNNLS.2020.2984770

[33] M. Faiz and U. Shanker, "Data synchronization in distributed client-server applications," in *2016 IEEE International Conference on Engineering and Technology (ICETECH)*. Coimbatore, India: IEEE, 2016, pp. 611–616. [Online]. Available: https://doi.org/10.1109/ICETECH.2016.7569323

[34] M. Dahlin, A. Brooke, M. Narasimhan, and B. Porter, "Data synchronization for distributed simulations," accessed via CiteSeerX. [Online]. Available: https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=97e26cc618e912ecc1f1f17fcced78ea68b1dff0

[35] N. Pushadapu and H. Hospital, "Real-time integration of data between different systems in healthcare: Implementing advanced interoperability solutions for seamless information flow," *Distributed Learning and Broad Applications in Scientific Research*, vol. 6, 2020.

[36] J. Sanhz, "How to enable, disable, and manage google sync," June 6 2019. [Online]. Available: https://www.technipages.com/manage-google-sync/

[37] Vera, "How to perform google cloud sync [3 solutions]," September 22 2022. [Online]. Available: https://www.multcloud.com/tutorials/google-cloud-sync-1234.html

[38] A. Midha, "Council post: How event-carried state transfer can create meaningful data synchronization," June 13 2023. [Online]. Available: https://www.forbes.com/councils/forbestechcouncil/2023/06/13/how-event-carried-state-transfer-can-create-meaningful-data-synchronization/

[39] M. A. Maddah-Ali and U. Niesen, "Fundamental limits of caching," *IEEE Transactions on Information Theory*, vol. 60, no. 5, pp. 2856–2867, 2014. [Online]. Available: https://doi.org/10.1109/TIT.2014.2306938

[40] G. Coviello, G. Avitabile, and A. Florio, "The importance of data synchronization in multiboard acquisition systems," in *2020 IEEE 20th Mediterranean Electrotechnical Conference (MELECON)*. Palermo, Italy: IEEE, 2020, pp. 293–297. [Online]. Available: https://doi.org/10.1109/MELECON48756.2020.9140622

[41] P. Rathi, "Banking dataset - marketing targets," https://www.kaggle.com/datasets/prakharrathi25/banking-dataset-marketing-targets?resource=download, 2020, accessed: 2025-06-24.

# Appendix A

# Proof of Existence

This thesis and all associated original work, including source code, experiments, evaluations, and figures, have been developed and completed by the author in accordance with the university's academic integrity standards.

Additionally, a signed PDF version of this document has been submitted to the university's institutional repository and shared via a trusted timestamping service to ensure authenticity.

## Code Snapshots

```python
def fetch_data():
        )
        cursor = connection.cursor()

        def has_created_at_column(table):
            cursor.execute(f"SHOW COLUMNS FROM {table};")
            return "created_at" in [column[0] for column in cursor.fetchall()]

        all_records = []
        cursor.execute("SHOW TABLES;")
        tables = [table[0] for table in cursor.fetchall()]
        print(f" Tables found: {tables}")

        for table in tables:
            try:
                cursor.execute(f"SELECT * FROM {table};")
                records = cursor.fetchall()
                columns = [desc[0] for desc in cursor.description]

                if not records:
                    print(f"Table `{table}` returned no records.")
                    continue

                for record in records:
                    formatted = {}
                    for col, val in zip(columns, record):
                        if isinstance(val, Decimal):
                            formatted[col] = float(val)
                        elif isinstance(val, (datetime, date)):
                            formatted[col] = val.strftime("%Y-%m-%d %H:%M:%S") if isinstance(val,
```

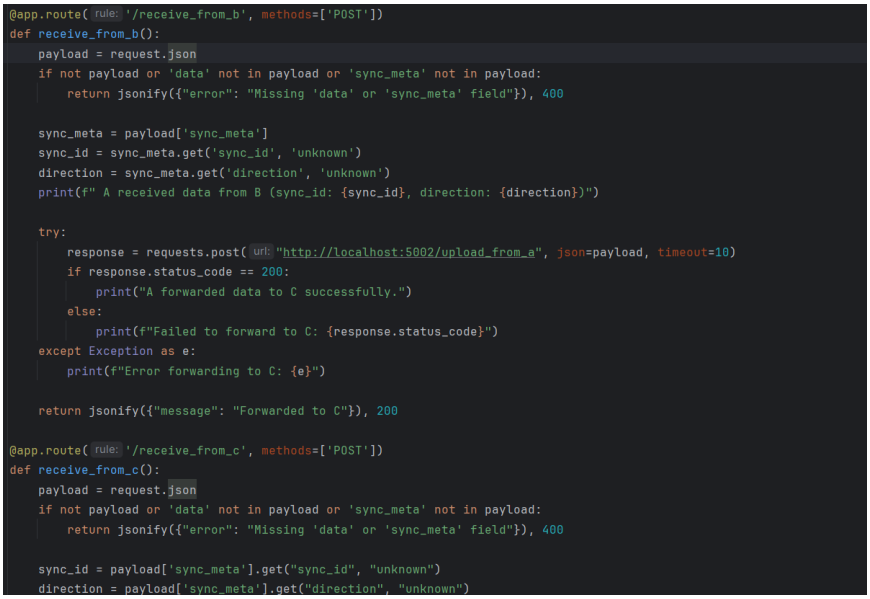
**Figure A.1:** Python function `fetch_data()` for extracting and formatting records from all tables in a MySQL database. This function queries all tables, retrieves their records, and converts special data types like decimals and timestamps for batch processing and transmission.

```python
def create_batches(data):  1 usage
    batch_size = max(1, len(data) // 6)
    batches = [data[i:i + batch_size] for i in range(0, len(data), batch_size)]
    return (batches + [[]] * 6)[:6]  # Ensure exactly 6 or another N batches
```

**Figure A.2:** Function `create_batches(data)` that splits the input dataset into evenly sized batches for distribution. This implementation ensures exactly 6 batches (or another fixed $N$), padding if necessary. It supports batch-based placement in the synchronization pipeline.

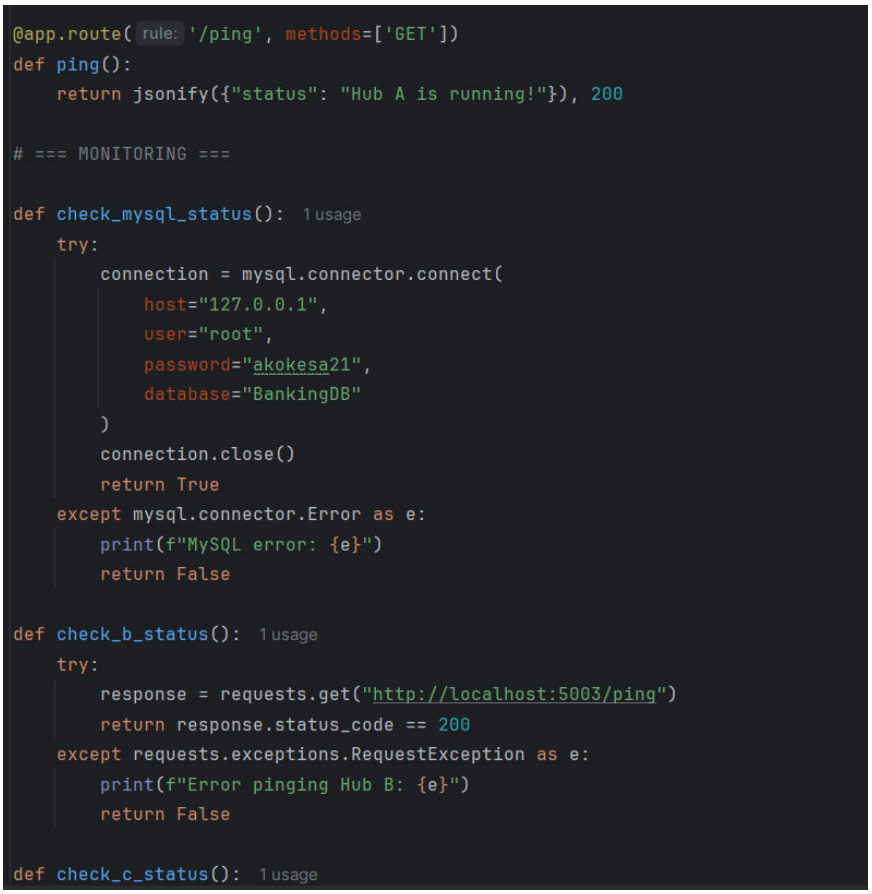
```python
def distribute_batches(batches):  1 usage
    return {
        "B1": batches[0] + batches[1] + batches[2],  # b1b2b3
        "B2": batches[0] + batches[3] + batches[4],  # b1b4b5
        "B3": batches[1] + batches[3] + batches[5],  # b2b4b6
        "B4": batches[2] + batches[4] + batches[5]  # b3b5b6
    }


def xor_batches(*batches):  12 usages
    combined = []
    max_len = max(len(b) for b in batches)
    for i in range(max_len):
        entry = []
        for batch in batches:
            if i < len(batch):
                entry.append(batch[i])
        combined.append({"xor": json.dumps(entry)})
    return combined

# Generate XOR parity packets
def generate_xor_packets(batches):
    return {
        "P1": xor_batches( *batches: batches[0], batches[1], batches[3]),  # b1 ⊕ b2 ⊕ b4
        "P2": xor_batches( *batches: batches[0], batches[2], batches[3]),  # b1 ⊕ b3 ⊕ b4
        "P3": xor_batches( *batches: batches[1], batches[2], batches[5]),  # b2 ⊕ b3 ⊕ b6
        "P4": xor_batches( *batches: batches[3], batches[4], batches[5])   # b4 ⊕ b5 ⊕ b6
    }
```

**Figure A.3:** Python functions for batch distribution and XOR-coded packet generation. The `distribute_batches()` method assigns batches to nodes based on a fixed redundancy pattern. The `xor_batches()` function performs element-wise encoding across batches to produce parity packets. These packets are generated using `generate_xor_packets()`



```python
@app.route( rule: '/receive_from_b', methods=['POST'])
def receive_from_b():
    payload = request.json
    if not payload or 'data' not in payload or 'sync_meta' not in payload:
        return jsonify({"error": "Missing 'data' or 'sync_meta' field"}), 400

    sync_meta = payload['sync_meta']
    sync_id = sync_meta.get('sync_id', 'unknown')
    direction = sync_meta.get('direction', 'unknown')
    print(f" A received data from B (sync_id: {sync_id}, direction: {direction})")

    try:
        response = requests.post( url: "http://localhost:5002/upload_from_a", json=payload, timeout=10)
        if response.status_code == 200:
            print("A forwarded data to C successfully.")
        else:
            print(f"Failed to forward to C: {response.status_code}")
    except Exception as e:
        print(f"Error forwarding to C: {e}")

    return jsonify({"message": "Forwarded to C"}), 200

@app.route( rule: '/receive_from_c', methods=['POST'])
def receive_from_c():
    payload = request.json
    if not payload or 'data' not in payload or 'sync_meta' not in payload:
        return jsonify({"error": "Missing 'data' or 'sync_meta' field"}), 400

    sync_id = payload['sync_meta'].get("sync_id", "unknown")
    direction = payload['sync_meta'].get("direction", "unknown")
```

**Figure A.4:** Flask routes implemented in Hub A for relaying synchronization data between Hubs B and C. The `receive_from_b()` function accepts JSON payloads from Hub B and attempts to forward them to Hub C via an HTTP POST request. If the response is successful, Hub A logs the confirmation; otherwise, it handles and logs forwarding failures. The `receive_from_c()` function similarly accepts incoming data from Hub C, making Hub A a bidirectional relay point for synchronization traffic.

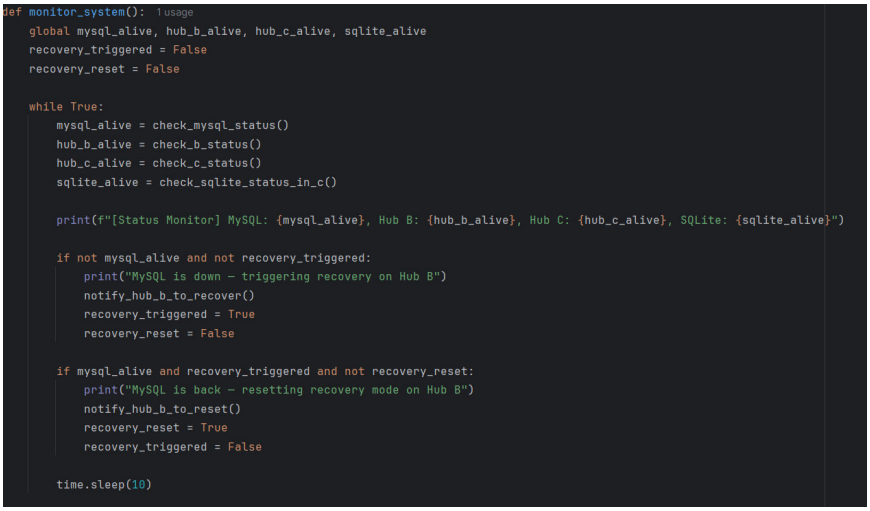


```python
@app.route( rule: '/ping', methods=['GET'])
def ping():
    return jsonify({"status": "Hub A is running!"}), 200

# === MONITORING ===

def check_mysql_status():  1 usage
    try:
        connection = mysql.connector.connect(
            host="127.0.0.1",
            user="root",
            password="akokesa21",
            database="BankingDB"
        )
        connection.close()
        return True
    except mysql.connector.Error as e:
        print(f"MySQL error: {e}")
        return False

def check_b_status():  1 usage
    try:
        response = requests.get("http://localhost:5003/ping")
        return response.status_code == 200
    except requests.exceptions.RequestException as e:
        print(f"Error pinging Hub B: {e}")
        return False

def check_c_status():  1 usage
```

**Figure A.5:** Flask-based status monitoring functions in Hub A. The `/ping` route allows external systems to confirm that Hub A is operational. The `check_mysql_status()` function attempts to establish a connection with the MySQL database and reports connection success or failure. Similarly, `check_b_status()` and `check_c_status()` verify connectivity to Hubs B and C, respectively, by issuing GET requests to their `/ping` endpoints. These functions support centralized health monitoring and error handling across the system.

```python
def check_c_status():  1 usage
    try:
        response = requests.get("http://localhost:5002/ping")
        return response.status_code == 200
    except requests.exceptions.RequestException as e:
        print(f"Error pinging Hub C: {e}")
        return False

def check_sqlite_status_in_c():  1 usage
    try:
        connection = sqlite3.connect('hub_c_simulated_oracle.db')
        connection.close()
        return True
    except sqlite3.Error as e:
        print(f"SQLite error in Hub C: {e}")
        return False

def notify_hub_b_to_recover():  1 usage
    try:
        response = requests.post("http://localhost:5003/trigger_recovery")
        if response.status_code == 200:
            print("Notified Hub B to start recovery mode.")
        else:
            print("Hub B recovery trigger failed.")
    except requests.exceptions.RequestException as e:
        print(f"Could not reach Hub B to trigger recovery: {e}")

def notify_hub_b_to_reset():  1 usage
    try:
        response = requests.post("http://localhost:5003/reset_recovery")
```

**Figure A.6:** Recovery and monitoring functions in Hub A. The `check_c_status()` function verifies the availability of Hub C via its `/ping` endpoint, while `check_sqlite_status_in_c()` attempts to connect to the simulated Oracle database at Hub C. The `notify_hub_b_to_recover()` and `notify_hub_b_to_reset()` functions are responsible for triggering or resetting the recovery mode in Hub B through HTTP POST requests. These functions enable Hub A to orchestrate fault detection and initiate appropriate recovery workflows.



```
def monitor_system():  1 usage
    global mysql_alive, hub_b_alive, hub_c_alive, sqlite_alive
    recovery_triggered = False
    recovery_reset = False

    while True:
        mysql_alive = check_mysql_status()
        hub_b_alive = check_b_status()
        hub_c_alive = check_c_status()
        sqlite_alive = check_sqlite_status_in_c()

        print(f"[Status Monitor] MySQL: {mysql_alive}, Hub B: {hub_b_alive}, Hub C: {hub_c_alive}, SQLite: {sqlite_alive}")

        if not mysql_alive and not recovery_triggered:
            print("MySQL is down — triggering recovery on Hub B")
            notify_hub_b_to_recover()
            recovery_triggered = True
            recovery_reset = False

        if mysql_alive and recovery_triggered and not recovery_reset:
            print("MySQL is back — resetting recovery mode on Hub B")
            notify_hub_b_to_reset()
            recovery_reset = True
            recovery_triggered = False

        time.sleep(10)
```

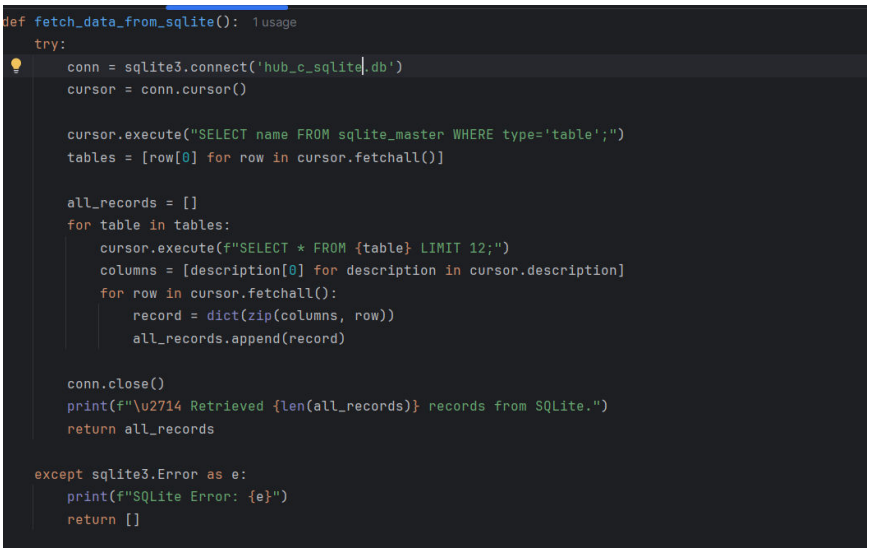
**Figure A.7:** Central monitoring loop in Hub A, implemented in `monitor_system()`. This function continuously checks the health status of MySQL (Hub B), SQLite (Hub C), and both hubs via their respective endpoints. If MySQL is found to be down, the system triggers a recovery request to Hub B. Once MySQL is restored, the loop detects its availability and sends a reset signal to disable recovery mode. This loop executes every 10 seconds and enables Hub A to act as a live orchestrator for failure detection and automated response.

```python
def send_to_hub_c(data):
    if not (hub_c_alive and sqlite_alive):
        print("Skipping send to Hub C: either Hub C or SQLite is down.")
        return

    sync_meta = {
        "sync_id": str(uuid4()),
        "start_time": time.time(),
        "direction": "sqlite_to_mysql"
    }

    payload = {
        "data": data,
        "sync_meta": sync_meta
    }

    try:
        c_url = "http://localhost:5002/process"
        response = requests.post(c_url, json=payload)
        if response.status_code == 200:
            print(f"Data sent to Hub C successfully (sync_id: {sync_meta['sync_id']})")
        else:
            print(f"Failed to send to Hub C: {response.status_code} - {response.text}")
    except requests.exceptions.RequestException as e:
        print(f"Error sending data to Hub C: {e}")
```

**Figure A.8:** Python function `send_to_hub_c(data)` for transmitting data from Hub A to Hub C. This function checks whether both Hub C and its local SQLite database are operational before proceeding. It builds a synchronization payload containing the data and metadata such as a UUID-based `sync_id`, a timestamp, and a direction tag. The payload is then sent via a POST request to Hub C's processing endpoint. Success or failure responses are logged, and error handling ensures that any connection issues are clearly reported.

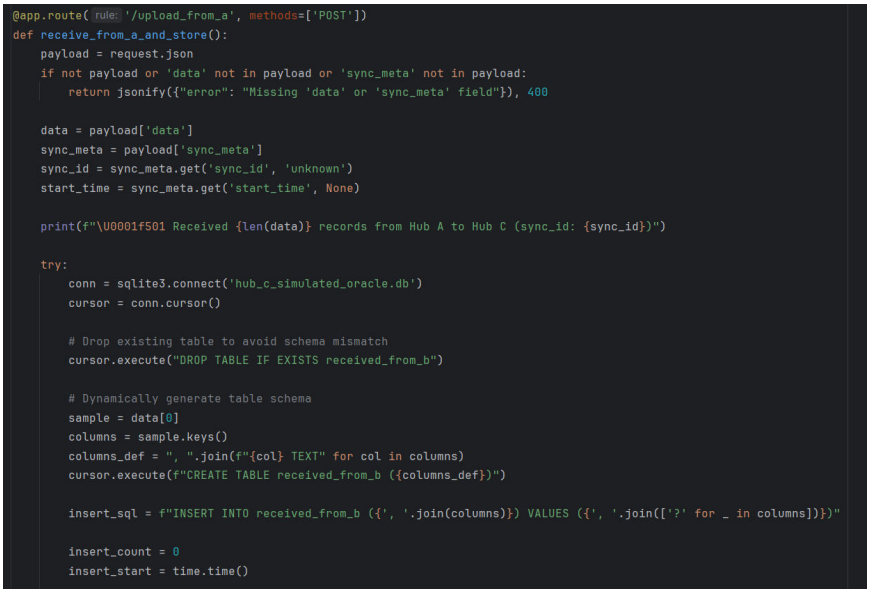
```python
def send_to_hub_b_from_c_node(data):
    try:
        url = "http://localhost:5003/receive_from_c"
        headers = {"Content-Type": "application/json"}
        payload = {"data": data}

        print("Forwarding full data from Hub C (via A) to Hub B...")
        response = requests.post(url, json=payload, headers=headers, timeout=10)

        if response.status_code == 200:
            print("✅ Data forwarded to Hub B successfully.")
        else:
            print(f"❌ Failed to forward to Hub B: {response.status_code} - {response.text}")
    except Exception as e:
        print(f"⚠️ Transmission error to Hub B: {e}")

# === App Entry Point ===

if __name__ == "__main__":
    monitor_thread = threading.Thread(target=monitor_system, daemon=True)
    monitor_thread.start()
    app.run(host='0.0.0.0', port=5001)
```

**Figure A.9:** Python function `send_to_hub_b_from_c_node(data)` is executed by Hub A to forward a complete dataset from Hub C to Hub B. This function constructs a POST request to Hub B's receiving endpoint, including a JSON payload. It logs success or failure messages depending on the response status and includes exception handling for transmission errors. Below the function, the application's entry point initializes a background monitoring thread and starts the Flask server on port 5001.



```
def fetch_data_from_sqlite():  1 usage
    try:
        conn = sqlite3.connect('hub_c_sqlite.db')
        cursor = conn.cursor()

        cursor.execute("SELECT name FROM sqlite_master WHERE type='table';")
        tables = [row[0] for row in cursor.fetchall()]

        all_records = []
        for table in tables:
            cursor.execute(f"SELECT * FROM {table} LIMIT 12;")
            columns = [description[0] for description in cursor.description]
            for row in cursor.fetchall():
                record = dict(zip(columns, row))
                all_records.append(record)

        conn.close()
        print(f"\u2714 Retrieved {len(all_records)} records from SQLite.")
        return all_records

    except sqlite3.Error as e:
        print(f"SQLite Error: {e}")
        return []
```

**Figure A.10:** Function `fetch_data_from_sqlite()` is executed in Hub C to extract records from all tables stored in the SQLite database. It connects to `hub_c_sqlite.db`, retrieves table names dynamically, executes a `SELECT * ` query for each table (limited to 12 rows per table), and formats the results into a list of dictionaries. This function enables downstream synchronization or transmission by packaging the retrieved data in a structured format.



```python
@app.route( rule: '/upload_from_a', methods=['POST'])
def receive_from_a_and_store():
    payload = request.json
    if not payload or 'data' not in payload or 'sync_meta' not in payload:
        return jsonify({"error": "Missing 'data' or 'sync_meta' field"}), 400

    data = payload['data']
    sync_meta = payload['sync_meta']
    sync_id = sync_meta.get('sync_id', 'unknown')
    start_time = sync_meta.get('start_time', None)

    print(f"\U0001f501 Received {len(data)} records from Hub A to Hub C (sync_id: {sync_id})")

    try:
        conn = sqlite3.connect('hub_c_simulated_oracle.db')
        cursor = conn.cursor()

        # Drop existing table to avoid schema mismatch
        cursor.execute("DROP TABLE IF EXISTS received_from_b")

        # Dynamically generate table schema
        sample = data[0]
        columns = sample.keys()
        columns_def = ", ".join(f"{col} TEXT" for col in columns)
        cursor.execute(f"CREATE TABLE received_from_b ({columns_def})")

        insert_sql = f"INSERT INTO received_from_b ({', '.join(columns)}) VALUES ({', '.join(['?' for _ in columns])})"

        insert_count = 0
        insert_start = time.time()
```

**Figure A.11:** Function `receive_from_a_and_store()` runs in Hub C and handles POST requests sent from Hub A. It validates the incoming JSON payload, extracts the data and synchronization metadata, and dynamically rebuilds a local SQLite table named `received_from_b`. To avoid schema mismatches, the function drops any existing table before generating a new one based on the structure of the incoming data. This mechanism ensures compatibility and seamless ingestion of synchronized content into Hub C's database.

```
def receive_from_a_and_store():

        insert_count = 0
        insert_start = time.time()

        def safe_str(val):
            try:
                return str(val).encode( encoding: 'utf-8', errors: 'replace').decode('utf-8')
            except Exception:
                return ""

        for entry in data:
            values = tuple(safe_str(entry.get(col, "")) for col in columns)
            cursor.execute(insert_sql, values)
            insert_count += 1

        conn.commit()
        insert_duration = round(time.time() - insert_start, 3)
        conn.close()

        if start_time:
            adjusted_duration = round(max(time.time() - start_time, 0), 3)
            print(f"FULL SYNC (MySQL → A → C → SQLite) completed in {adjusted_duration} seconds (sync_id: {sync_id})")

        print(f"Stored {insert_count} structured records into SQLite.")
        print(f"SQLite insert duration: {insert_duration} seconds")
        return jsonify({"message": "Data stored in SQLite successfully"}), 200

    except Exception as e:
        print(f"\u274c SQLite insert error: {e}")
        return jsonify({"error": str(e)}), 500
```

**Figure A.12:** Continuation of the `receive_from_a_and_store()` function. After preparing the SQLite table schema, this block inserts each data record using a safe string conversion method to avoid encoding issues. The function tracks insertion duration, prints sync completion messages, and returns a success status if all records are stored correctly. If any step fails, it logs the error and returns a 500 response to the caller. This implementation enables robust ingestion and timing validation for full sync operations (MySQL → Hub A → Hub C → SQLite).

```
@app.route( rule: '/ping', methods=['GET'])
def ping():
    return jsonify({"status": "Hub C is running!"}), 200

if __name__ == "__main__":
    threading.Thread(target=background_sync_to_a, daemon=True).start()
    app.run(host="0.0.0.0", port=5002)
```

**Figure A.13:** Ping route and entry point for Hub C. The `/ping` route provides a simple heartbeat check to confirm Hub C is operational. Upon execution, the script launches a background thread responsible for initiating synchronization to Hub A and runs the Flask application on port 5002.