DECENTRALIZED REPLICA MANAGEMENT IN EDGE ENVIRONMENTS WITH LATENCY AND RESOURCE OPTIMIZATION

A PROJECT REPORT

Submitted by

SANJAI A	**(810621104023)**
MOHAN K	**(810621104014)**
BALACHANDHIRAN K	**(810621104304)**

in partial fulfilment for the award of the degree

of

BACHELOR OF ENGINEERING

IN

COMPUTER SCIENCE AND ENGINEERING

Dr. NAVALAR NEDUNCHEZHIYAN COLLEGE OF ENGINEERING

THOLUDUR – 606303

ANNA UNIVERSITY: CHENNAI 600 025

MAY-2025

ANNA UNIVERSITY: CHENNAI 6000025

BONAFIDE CERTIFICATE

Certified that this project report "**DECENTRALIZED REPLICA MANAGEMENT IN EDGE ENVIRONMENTS WITH LATENCY AND RESOURCE OPTIMIZATION**" is the bonafide work of "**SANJAI A (810621104023), MOHAN K (810621104014)** and **BALACHANDHIRAN M (810621104304)**" who carried out the project under my supervision.

SIGNATURE

Mr.T.S.RAJA,M.E.,
Head of the Department,
Department of computer science and Engineering,
Dr. Navalar Nedunchezhiyan College of Engineering, Tholudur-606303.

SIGNATURE

Mrs.K.BHAVEENA,M.E.,
Assistant Professor,
Department of Information Technology
Dr. Navalar Nedunchezhiyan College of Engineering, Tholudur-606303.

Submitted for the **PROJECT VIVA-VOCE** held on.....at Dr. Navalar Nedunchezhiyan College of Engineering, Tholudur-606303.

INTERNAL EXAMINER **EXTERNAL EXAMINER**

ACKNOWLEDGEMENT

We sincerely thanks to a number of people who have provided understanding and assistance in the completion of the Project.

We express our sincere gratitude to our honourable Chairman of Sree Arumugham Group of Institutions **Mr.P.T.RAJAN** for providing us with the excellent infrastructure to make our project a successful one.

We extend our thanks to our respected **Secretary** of Sree Arumugham Group of Institutions **Mr.R.ANNAMALAI , B.E.,**for providing us all the amenities for the completion of our project.

We extend our thanks to our respected **Managing Director** of Sree Arumugham Group of Institutions **Prof. A. BALADHANDAPANI, M.A., M.Phil.,** for providing us all the amenities for the completion of our project

We extend our thanks to our respected **Director** of Sree Arumugham Group of Institutions **Er.A.B.MADHAN,M.E.,(Ph.d)** for providing us all the amenities for the completion of our project

We extend our thanks to our respected **Principal Dr.R.LATHA,M.E, Ph.D.,** for providing us all the facilities and amenities for the completion of our project.

We express our sincere thanks to our **Head of the Department and Project Co-Ordinator Mr.T.S.RAJA, M.E.,** for his assistance to make our project a successful one.

We express our sincere thanks to our **Internal Project Guide Mrs.K.BHAVEENA, M.E.,** for her continuous encouragements and kind suggestions in every step throughout this project.

ABSTRACT

Decentralized replica management in latency-bound edge environments is critical for ensuring data availability, consistency, and performance while minimizing resource consumption. This paper presents a novel approach to managing data replicas in edge environments, where strict latency requirements and resource constraints pose unique challenges. The proposed decentralized framework leverages local decision-making algorithms to dynamically adapt replica placement and migration based on real-time workload patterns, network conditions, and resource availability. By incorporating latency-aware heuristics, the system prioritizes proximity to latency-sensitive applications while balancing computational and storage costs. Extensive simulations demonstrate that the approach achieves significant reductions in resource usage while maintaining low-latency access, making it suitable for applications such as IoT, content delivery, and healthcare. This work advances the state-of-the-art in edge computing by providing a scalable and resource-efficient solution for replica management in geographically distributed and resource-constrained environments.

LIST OF ABBREVIATIONS

JVM	Java Virtual Machine
JDK	Java Development Kit
API	Application Programming Interface
RMI	Remote Method Invocation
IoT	Internet of Things
SSL/TLS	Secure Sockets Layer / Transport Layer Security
REST	Representational State Transfer
EMC	Edge Mini – Cloud
EUA	Edge User Allocation

LIST OF TABLES

LIST OF FIGURES

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

1.1 Introduction

In recent years, the proliferation of latency-sensitive applications such as Internet of Things (IoT) systems, real-time analytics, healthcare monitoring, and content delivery networks has driven the widespread adoption of edge computing. By bringing computation and data storage closer to end-users, edge environments significantly reduce communication delays and alleviate the load on centralized cloud infrastructures. However, the decentralized and resource-constrained nature of edge environments introduce new challenges in managing data availability, consistency, and system performance. Replica management, which involves the strategic placement and movement of data copies across edge nodes, is critical to achieving reliable and low-latency data access. Traditional centralized approaches often fail to scale or adapt efficiently under dynamic workloads and fluctuating network conditions. This paper proposes a novel decentralized replica management framework that leverages local decision-making and latency-aware heuristics to optimize replica placement and migration. The framework dynamically adjusts to real-time factors such as workload distribution, network latency, and resource availability, ensuring minimal resource consumption while meeting strict performance requirements. Through extensive simulation and analysis, the proposed approach demonstrates its effectiveness in improving scalability, reducing latency, and conserving resources in complex edge environments

1.2 Objectives

The primary objective of this research is to develop a decentralized replica management framework tailored for latency-bound edge computing environments.

The goal is to ensure high data availability and low-latency access while minimizing computational and storage resource usage. This framework leverages local decision-making algorithms and latency-aware heuristics to dynamically adapt the placement and migration of data replicas in response to real-time changes in workload patterns, network conditions.

1.3 Problem Statement

In today's edge computing landscape, the demand for ultra-low latency and high availability has significantly increased due to the rise of time- critical applications like IoT monitoring, telemedicine, smart cities, and real- time video analytics. These applications often run on distributed, resource- constrained edge nodes with limited processing power, memory, and network bandwidth. To support data availability and fault tolerance, replication is essential. However, traditional centralized replica management approaches struggle in dynamic and latency-sensitive edge environments.

Such centralized strategies introduce several limitations:

- Increased communication latency due to multiple hops through central nodes.
- Inefficient use of resources by maintaining unnecessary or ill-placed replicas.
- Scalability issues and vulnerability due to the presence of a central point of failure.
- Inability to respond swiftly to changes in workload, node availability, or network performance.

1.4 Scope

This study focuses on replica management in geographically distributed edge environments characterized by limited resources and strict latency requirements. The scope includes:

• Designing a decentralized architecture that eliminates the need for a central coordinator.

• Implementing adaptive algorithms that make local decisions based on current system states.

• Prioritizing latency-sensitive applications such as IoT, healthcare, and real-time content delivery.

• Evaluating the framework's performance using simulations to measure latency, resource usage, and scalability.

• Addressing trade-offs between proximity, consistency, and cost-efficiency in replica placement.

• The proposed solution is not intended for traditional cloud data centers or high-bandwidth, resource-rich environments, but rather for constrained and dynamic edge networks.

CHAPTER 2

LITERATURE SURVEY

1. **A Decentralized Replica Placement Algorithm for Edge Computing Atakan Aral, Member, IEEE, and Tolga Ovatman, Member, IEEE**

As the devices that make up the Internet become more powerful, algorithms that orchestrate cloud systems are on the verge of putting more responsibility for computation and storage on these devices. In our current age of Big Data, dissemination and storage of data across end cloud devices is becoming a prominent problem subject to this expansion. In this paper, we propose a distributed data dissemination approach that relies on dynamic creation/replacement/removal of replicas guided by continuous monitoring of data requests coming from edge nodes of the underlying network. Our algorithm exploits geographical locality of data during the dissemination process due to the plenitude of common data requests that stem from the clients within a close proximity. Our results using both real world and synthetic data demonstrate that a decentralized replica placement approach provides significant cost benefits compared to client side caching that is widely used in traditional distributed systems.

Replica Placement Algorithm We present D-REP algorithm where storage nodes that host replicas analyze observed demand on replicas and act as local optimizers. They evaluate cost of storing replicas as well as expected latency improvement to make a migration or duplication decision to one of neighbours. They may also decide to remove the local replica. Such decisions are made to maximize an objective function based on FLP. The algorithm also allows user to control the balance between cost- and latency-optimization using an input parameter. Experimental results on both real and synthetic workload traces demonstrate significant improvements in replica access latency as well as network overhead and storage cost.

Messaging Methodology In order to gain promised benefits of D-REP algorithm, edge entities should be aware of the closest replica when they request a data object. However, complete awareness is only possible with centralized control or by broadcasting replica locations periodically. We instead propose a replica discovery approach where the most relevant nodes are identified and only they are notified of replica creations or removals.

2. Latency-aware and Proactive Service Placement for Edge Computing

Henda Sfaxi (1), Imene Lahyani (1), Sami Yangui (2), Mouna Torjmen

(1) ReDCAD, ENIS, University of Sfax, 3038 Sfax, Tunisia

Smart IoT devices and applications in smart cities exchange important real-time information with their environment. However, a subset of these systems may face limitations in analyzing and processing the required large amounts of data to meet ultra-low-latency criteria. This limitation could be attributed to factors such as constrained CPU and battery resources. Thanks to the 5G and edge computing capabilities, a viable solution involves migrating a subset of these latency sensitive and computationally intensive tasks to edge nodes and servers. This strategic service placement ensures a safe continuity of the application. In this paper, autonomous cars operating in smart cities, engaging in continuous data exchange with their external environment to meet real-time and latency sensitive requirements, serve as an illustrative example of smart applications. The car's decision service is strategically placed on edge nodes through a proactive (re)placement approach designed for dynamic and mobile environments. This approach uses a quality of service (QoS) metric prediction degradation module, which leverages Exponential smoothing methods to identify a suitable edge node for hosting the car's decision module, with latency as a key criterion. Multiple configurations for outlier detection techniques are evaluated.

A proof-of-concept validates the chosen model by comparing it to the Auto Regressive Integrated Moving Average (ARIMA) and the proposed proactive service (re)placement approach. This approach ensures the continuity of the placed module, suggesting the feasibility of locating noncritical modules on edge nodes.

3. Replica Placement in Edge Computing Leonardo Marques Epifanio leonardo.epifanio@tecnico.ulisboa.pt

Edge computing is defined as a paradigm in which servers are placed close to the edge of the network, in order to assist applications that running in resource-constrained devices. There are two main advantages of edge computing: firstly, edge nodes can provide assistance with much lower latency than the cloud, because servers are physically closer to the devices; and secondly, edge nodes can shield the cloud from most requests, by serving the requests locally. Edge nodes can help in the collection of information from the devices, by aggregating information from multiple devices before sending it to the cloud. Edge nodes can also make information available to local devices, that can then access this information with low latency. In this work we are mainly interested in the latter, and we study techniques that can be used to place replicas of relevant data on edge devices. We propose a system that can collect estimates of future data demand from different sources (such as historical data or current observed access patterns) and that can make data placement decision based on these demand estimates, taking also into consideration the costs of maintaining data replicas and the benefits that can be achieved by maintaining those replicas. We will illustrate the operation of our system using examples from the area of vehicular networks, where edge nodes can help vehicles to select the best trac routes based on up-to-date information regarding road congestion, accidents, or other hazards.

4. Resource Scheduling in Edge Computing: A Survey Quyuan Luo , Shihong Hu , Changle Li , Senior Member, IEEE, Guanghui Li , and WeisongShi ,`Fellow,'IEEE

The proliferation of the Internet of Things (IoT) and the wide penetration of wireless networks, the surging demand for data communications and computing calls for the emerging edge computing paradigm. By moving the services and functions located in the cloud to the proximity of users, edge computing can provide powerful communication, storage, networking, and communication capacity. The resource scheduling in edge computing, which is the key to the success of edge computing systems, has attracted increasing research interests. In this paper, we survey the state-of-the-art research findings to know the research progress in this field. Specifically, we present the architecture of edge computing, under which different collaborative manners for resource scheduling are discussed. Particularly, we introduce a unified model before summarizing the current works on resource scheduling from three research issues, including computation offloading, resource allocation, and resource provisioning. Based on two modes of operation, i.e., centralized and distributed modes, different techniques for resource scheduling are discussed and compared. Also, we summarize the main performance indicators based on the surveyed literature. To shed light on the significance of resource scheduling in real world scenarios, we discuss several typical application scenarios involved in the research of resource scheduling in edge computing. Finally, we highlight some open research challenges yet to be addressed and outline

5. Efficient Replication Management in Distributed Systems by Michael Rabinovich

Replication is a critical aspect of large-scale distributed systems. Without it, the size of a system is limited by factors such as the risk of component failures, the overloading of popular services, and access latency to remote parts of the system. Replication overcomes these problems by allowing service to continue despite failures using remaining replicas, and by distributing requests for a given service among multiple server replicas. However, replicated systems incur significant performance overhead for maintaining multiple replicas and keeping them mutually consistent. Managing replication efficiently is therefore important in building large-scale distributed systems. This dissertation concentrates on quorum-based replication management. It proposes several ways to manage replication efficiently using different types of quorums. First, we study structure-based quorums, which are attractive because they result in low-overhead replica management when the number of replicas is high. We propose a way to significantly improve system availability in protocols using these quorums. We also study in depth the performance of a particular class of these quorums based on a grid structure.

CHAPTER 3

SYSTEM STUDY

3.1 Existing System

In existing edge computing environments, replica management is typically handled using centralized or semi-centralized architectures. These systems rely on a central coordinator or cloud-based controller to make decisions regarding the placement, consistency, and synchronization of data replicas. While this approach simplifies the design and allows for global system visibility, it introduces several limitations in latency-sensitive edge scenarios. The dependence on a central entity increases communication overhead, results in higher access latency, and creates a potential single point of failure.

Moreover, existing systems often assume stable network connectivity and abundant computational resources, which are not always available in edge environments. This mismatch leads to inefficient resource usage, poor scalability, and suboptimal performance when applied to real-world edge use cases such as IoT, healthcare monitoring, or smart surveillance. Many traditional systems also lack dynamic adaptation to workload fluctuations or network conditions, leading to static replica placement strategies that degrade under varying loads.

Disadvantages of Existing System

- Depends on a central coordinator, leading to high latency.
- Introduces a single point of failure, risking system downtime.
- Static replica placement that doesn’t adapt to real-time changes.

3.2 Proposed System

The proposed system introduces a novel decentralized approach to replica management in latency-bound edge computing environments, addressing the limitations of existing systems. Unlike traditional centralized models, our system eliminates the need for a central coordinator and allows each edge node to make autonomous decisions regarding replica placement, migration, and deletion based on real-time workload patterns, network conditions, and resource availability. This decentralized design ensures scalability and fault tolerance, as each node operates independently, reducing the risk of a single point of failure and minimizing communication overhead between nodes.

A key feature of the proposed system is its ability to adapt dynamically to changing edge conditions. By leveraging latency-aware heuristics, the system prioritizes the placement of replicas close to latency-sensitive applications, such as IoT sensors or real-time data analytics, ensuring low- latency access. Additionally, the system intelligently balances resource consumption, optimizing both computational power and storage usage to prevent over-utilization of edge nodes, which typically have limited resources. Replica migration and adaptation occur seamlessly, ensuring that data remains consistent and accessible while minimizing costs associated with computation and bandwidth.

Advantages of Proposed System

- Eliminates central controller, allowing fully decentralized decision-making.
- Ensures low-latency access by placing replicas closer to applications.
- Improves fault tolerance by removing the single point of failure.
- Adapts dynamically to real-time workload and network changes

CHAPTER 4

SYSTEM DESIGN

4.1 Block Diagram

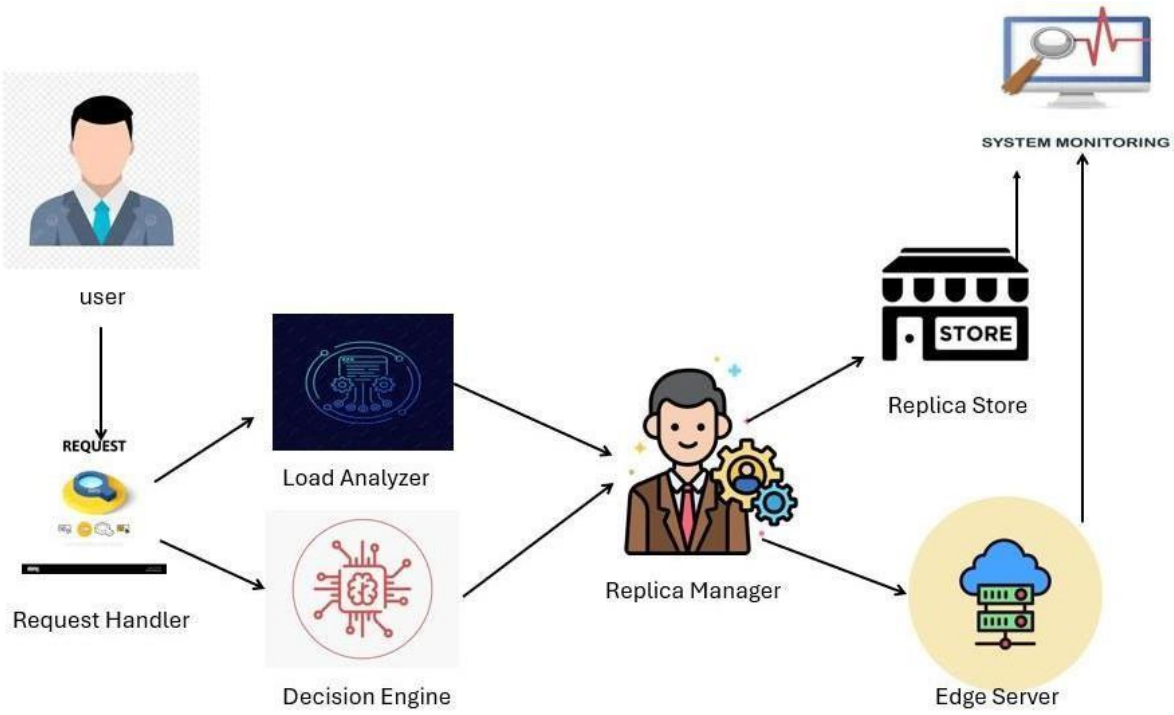

Fig 4.1 Block Diagram

4.2 Data Flow Diagram

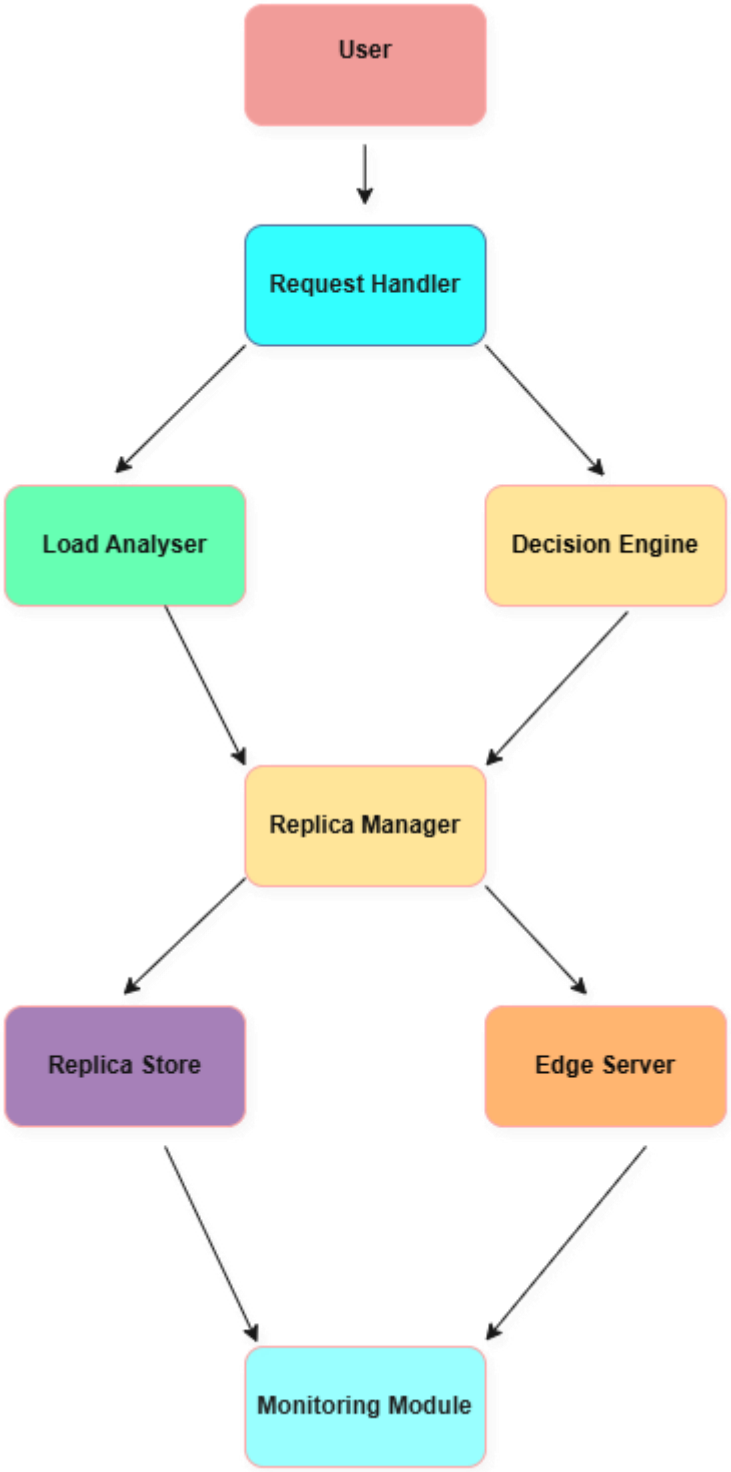

Fig 4.2 Data Flow Diagram

4.3 Database Design

FIELD NAME	DATA TYPE	DESCRIPTION
ID	INT (Auto Increment, Primary Key)	Unique identifier for each download record
FILENAME	VARCHAR(255)	Name of the file that was downloaded
USERNAME	VARCHAR(255)	Username of the user who downloaded the file
DOWNLOAD_TIME	TIMESTAMP	Time when the file was downloaded

Table 4.3.1 Downloads Table

CHAPTER 5

SYSTEM SPECIFICATION

5.1 Hardware Specification

Hardware is the physical components of the computer like microprocessor, hard disks, RAM, and motherboard. Hardware devices are the executors of the commands provided by software applications. Computer hardware as the electronic, magnetic, and electric devices that carry out the computing functions.

Processor	:	Intel i5 or i7/ AMD Ryzen 5 or 7
Clock speed	:	2.30GHz
Hard Disk	:	20GB
RAM	:	Minimum 8GB (Recommended 16GB)
Operating System	:	Windows 10/ Windows11

5.2 Software Specification

Software is a set of instructions that are used to command any system to perform any operation. Software has the advantage to make decisions and to deliver sensible results and is useful in handling complex situations.

Front End	:	JAVA 17
Database	:	MySql
Web Server	:	GlassFish
IDE	:	Apache NetBeans 24

Java 17

The Java Development Kit (JDK) serves as the foundational tool for developing and running Java applications. In this project, JDK is used for both the front-end and back-end development, enabling the creation of web-based interfaces, data processing algorithms, and interaction with databases. The JDK provides a comprehensive set of libraries and tools for managing Java code, compiling, debugging, and testing. It also supports the integration of advanced libraries for networking and concurrency, which are essential for the decentralized nature of the replica management system, allowing it to efficiently handle real-time data updates and decision-making tasks.

Java is a widely adopted, object-oriented programming language that plays a pivotal role in the development of distributed systems, particularly in edge computing environments. Its "write once, run anywhere" capability, powered by the Java Virtual Machine (JVM), allows developers to deploy applications across a wide range of edge devices regardless of their operating system or hardware. This platform independence is especially beneficial in latency-bound environments where edge nodes can be highly heterogeneous, varying in processing power, memory capacity, and operating platforms. Java’s abstraction from low-level system details enables developers to focus on building reliable and adaptive features without getting bogged down in system-specific complications, making it an ideal choice for building decentralized replica management systems.

A core requirement in edge computing is the ability to manage multiple operations concurrently—such as monitoring resource usage, handling data requests, and performing replication decisions. Java's built-in support for multithreading and concurrency offers powerful tools for these purposes.

Through the use of thread pools, the **java.util.concurrent** package, and synchronization mechanisms, Java allows multiple tasks to run in parallel without interfering with one another.

This is particularly advantageous when different edge nodes must make local decisions simultaneously and respond to changing network and workload conditions in real-time. As a result, the decentralized replica management system can maintain high responsiveness and low-latency performance even under variable demand.

Networking is a fundamental part of distributed edge systems, and Java provides comprehensive networking APIs such as **java.net, java.nio**, and support for WebSocket and HTTP protocols. These features facilitate efficient communication among edge nodes to coordinate replica placement, share load statistics, and maintain consistency across replicas.

Java's Remote Method Invocation (RMI) and support for RESTful web services allow remote edge devices to invoke operations on each other seamlessly, thereby enabling truly decentralized coordination without the need for a central controller. This capability ensures that each node can contribute to the overall decision-making process, improving fault tolerance and scalability across the system.

Security is another critical consideration, particularly when dealing with sensitive or personal data such as patient records in healthcare or transaction data in financial applications. Java addresses this through a comprehensive security model that includes data encryption using libraries like javax.crypto, secure communication via SSL/TLS sockets, authentication mechanisms, and fine-grained access control with the Java Security Manager.

Java’s strong exception handling and memory management via garbage collection also help reduce common vulnerabilities like memory leaks or unauthorized access. These features together ensure that data integrity and privacy are maintained throughout the lifecycle of replica operations in edge environments.

From a software engineering perspective, Java encourages a clean, modular, and reusable codebase. Its object-oriented structure allows developers to implement the system as independent components—such as **ReplicaManager, NodeMonitor, DataSync, and HeuristicAnalyzer classes**—making the system easier to maintain, extend, and debug. Java’s ecosystem also includes powerful build tools (Maven, Gradle), testing frameworks (JUnit), and integrated development environments (Eclipse, IntelliJ, NetBeans), all of which contribute to faster development cycles and improved code quality.

Additionally, Java’s support for frameworks like Spring Boot can simplify backend service development for RESTful APIs, while tools like Apache Kafka or RabbitMQ can be integrated for efficient messaging across nodes. These capabilities make Java a comprehensive and practical choice for building a robust, secure, and efficient decentralized replica management system in latency-critical edge environments.

GlassFish

Glassfish is an open-source application server used for deploying Java- based web applications. It supports Jakarta EE (formerly Java EE) specifications, making it well-suited for running servlets, JSPs, and managing the web interface for your decentralized replica management system. Glassfish offers robust features, including high availability, scalability, and performance tuning, which are essential for supporting dynamic replica management in edge environments.

It also supports the integration of various Java libraries and provides reliable deployment for enterprise-level applications, making it an ideal choice for hosting the system's backend and handling communication with edge nodes.

MySQL

MySQL is a powerful, open-source relational database management system (RDBMS) used to store structured data. In this system, MySQL is used as the backend database for storing essential information such as replica metadata, node configurations, workload patterns, and resource usage logs. MySQL provides high performance, reliability, and ease of use, making it a solid choice for managing the data in a system that requires quick retrieval and updates. It allows for complex queries to be executed efficiently, ensuring that replica placement and migration decisions are based on up-to-date information stored in a relational format. The MySQL database also supports scalability, enabling it to handle growing datasets as the system is expanded.

CHAPTER 6

SYSTEM ANALYSIS

6.1 Project Description

The rapid expansion of edge computing has made it essential to design systems that can efficiently manage data across distributed nodes, especially in resource-constrained environments. Edge computing environments, such as those used in Internet of Things (IoT), healthcare, and real-time content delivery, present unique challenges in terms of low-latency requirements, limited computational resources, and fluctuating network conditions. Data replication is a critical component of these environments, as it ensures data availability and fault tolerance. However, traditional systems for replica management often rely on centralized architectures, which can introduce bottlenecks and high latency, reduce fault tolerance, and impose scalability limitations. In response to these challenges, this project introduces a novel **decentralized replica management framework** designed to improve the efficiency, scalability, and performance of edge systems.

The proposed system leverages a decentralized architecture where each edge node makes autonomous decisions regarding the placement, migration, and deletion of data replicas. By eliminating the dependency on a central server, the system minimizes communication overhead and reduces the risk of a single point of failure.

Each edge node uses a **dynamic decision-making engine** that continuously evaluates real-time workload patterns, resource availability (e.g., CPU, memory, bandwidth), and network conditions.

This enables nodes to adaptively place and migrate replicas, ensuring that critical data remains available close to the applications that need it most, while avoiding resource overutilization.

A central feature of the system is the incorporation of **latency-aware heuristics**. These heuristics allow the system to prioritize replica placement in locations that minimize data access latency, which is crucial for applications such as real-time analytics or healthcare monitoring, where even slight delays can have significant consequences. The system automatically adjusts replica locations as network conditions change, ensuring that the most latency-sensitive applications continue to have fast access to data, even in the face of fluctuating resources or network instability.

In addition to minimizing latency, the system is designed to optimize resource usage. By dynamically adapting to workload fluctuations and resource availability, it balances the demand for computational power and storage across edge nodes, ensuring that no single node is overburdened. This resource efficiency is crucial in edge environments where computational power and storage are often limited. The **fault tolerance** of the system is also a key advantage, as the decentralized nature of the framework ensures that the failure of one or more edge nodes does not significantly affect data availability, thanks to the redundancy and replication mechanisms in place.

To validate the effectiveness of this approach, the system is tested through extensive simulations under various edge computing scenarios. These simulations compare the proposed decentralized replica management system against traditional centralized systems, evaluating key performance indicators such as **latency**, **resource consumption**, **scalability**, and **fault tolerance**.

The results demonstrate that the proposed system significantly reduces resource consumption while maintaining low-latency access, making it highly suitable for latency-sensitive, resource-constrained applications.

This system is expected to be particularly beneficial for industries like **healthcare**, where real-time data access is critical for patient monitoring systems, and **content delivery networks**, where user experience depends on fast and reliable access to distributed media. The flexibility and scalability of the system also make it a viable solution for large-scale IoT deployments, where millions of devices and sensors must efficiently manage and share data across the network.

6.2 System Constraints

The proposed decentralized replica management system operates within several constraints that must be considered during both development and deployment. These constraints are primarily due to the inherent limitations of edge environments, which are resource-constrained and highly dynamic. Below are the key system constraints:

6.2.1 Resource Limitations

- **Limited Computational Power:**

  Edge nodes are typically low-power devices with limited processing capabilities compared to traditional cloud servers. As a result, the system must be designed to efficiently use available computational resources, avoiding excessive computation that could hinder the performance of the edge device.

- **Storage Constraints:**

  Edge nodes generally have limited storage capacity, which restricts the amount of data that can be replicated and stored locally. This constraint necessitates the development of efficient data replication

migration strategies to ensure that replicas are placed optimally and that storage is used only when necessary.

- **Network Bandwidth:**

  Edge networks may experience fluctuating bandwidth, affecting the speed and reliability of data transfers between edge nodes. The system must minimize the need for frequent replica migrations and large data transfers, using techniques like compression or incremental updates to reduce the data volume.

6.2.2 Latency Sensitivity

- **Low-Latency Requirements:**

  The system must prioritize low-latency access to replicated data, especially for applications in real-time environments like healthcare and IoT, where delays can have significant consequences. Any delays in replica placement or migration could lead to degraded user experience or failure to meet service-level agreements (SLAs).

- **Dynamic Latency Conditions:**

  Network latency can fluctuate due to congestion, node failures, or dynamic routing in edge environments. The system must adapt to these changing conditions and make real-time decisions about replica placement and migration to ensure optimal performance.

6.2.3 Scalability Constraints

- **Edge Node Heterogeneity:**

  The edge nodes involved in the system may differ in terms of their computational power, storage, and network capabilities. The system must account for this heterogeneity,

Ensuring that it can scale efficiently to handle both powerful and resource-constrained devices without compromising performance.

- **Growing Number of Devices:**

  As edge computing environments grow, the number of devices (such as IoT sensors or edge nodes) increases. The system must be able to scale horizontally to accommodate new devices and manage additional replicas, ensuring that performance and resource usage remain optimal even with large-scale deployments.

6.2.4 Security and Privacy Constraints

- **Data Security:**

  Edge nodes are often deployed in environments with limited physical security, making them vulnerable to malicious attacks. The system must ensure that data replicas are securely transferred and stored, using encryption and secure communication protocols to prevent unauthorized access.

- **Privacy Considerations:**

  The system must respect the privacy of users and comply with data privacy regulations such as GDPR. This includes ensuring that sensitive data is not unnecessarily replicated and that data processing is performed in compliance with privacy laws.

6.2.5 Fault Tolerance and Reliability

- **Node Failures:**

Edge environments are prone to node failures due to their deployment in unstable or remote locations. The system must be designed with fault tolerance in mind, ensuring that data remains available even if certain edge nodes fail or experience connectivity issues

- **Replica Consistency:**

Maintaining data consistency across replicas is crucial, especially in a decentralized system. The system must ensure that replicas are updated consistently while minimizing the cost of synchronization and preventing data conflicts.

6.2.6 Real-Time Adaptation

- **Dynamic Workload Fluctuations:**

The workload at edge nodes can vary over time due to changes in user behavior or application requirements. The system must adapt in real-time to workload fluctuations, ensuring that replicas are placed and migrated to meet current demands without unnecessary delays or resource consumption.

- **Real-Time Decision Making:**

The decision-making process for replica placement and migration needs to occur in real-time to ensure optimal performance under varying conditions. The system's ability to make timely decisions based on current network and resource conditions is essential for meeting the low-latency and of edge applications.

CHAPTER 7

IMPLEMENTATION

7.1 Modules

To address the challenges of latency, availability, and optimal resource utilization in decentralized edge computing environments, the system is architecturally divided into distinct functional modules. Each module is designed to perform a specialized task while working in coordination with others to ensure the system's scalability, resilience, and performance. The modular design allows for better fault isolation, efficient workload distribution, and easier system management. The following are the core modules developed as part of this project to implement a robust Decentralized Replica Management System with intelligent decision-making and continuous monitoring capabilities.

7.1.1 Replica Management Module

The Replica Management Module is pivotal in ensuring data availability, consistency, and optimal performance across distributed edge environments. **Key Functions:**

- **Dynamic Replica Creation and Placement:** Analyzes real-time data access patterns and system load to determine the optimal number and location of replicas, ensuring minimal latency and balanced resource utilization.
- **Consistency Maintenance:** Implements strategies such as eventual consistency or strong consistency models to ensure data integrity across replicas, adapting based on application requirements.
- **Load Balancing:** Distributes client requests intelligently among replicas to prevent bottlenecks and ensure efficient utilization of edge resources.

- **Fault Tolerance:** Detects node failures promptly and initiates replica redistribution or regeneration to maintain service continuity.

Implementation Considerations:

- **Scalability:** Designed to handle increasing numbers of edge nodes and data volumes without degradation in performance.
- **Resource Awareness:** Considers the heterogeneous nature of edge devices, tailoring replication strategies to device capabilities.

7.1.2 Communication Module

The Communication Module facilitates seamless and secure data exchange between system components, ensuring coordination and coherence across the decentralized architecture

Key Functions:

- **Protocol Management:** Supports multiple communication protocols (e.g., MQTT, CoAP, HTTP) to ensure compatibility with various devices and applications.
- **Data Serialization and Deserialization:** Handles the conversion of data into transmittable formats and vice versa, maintaining data integrity during transmission.
- **Secure Transmission:** Implements encryption and authentication mechanisms to protect data against unauthorized access and tampering.
- **Quality of Service (QoS):** Manages message prioritization and delivery guarantees to meet the specific needs of different applications.

Implementation Considerations:

- **Latency Optimization:** Employs strategies to minimize communication delays, crucial for real-time applications
- **Fault Tolerance:** Ensures message delivery even in the presence of network disruptions through retransmission strategies and alternative routing.

7.1.3 Node Monitoring Module

The Node Monitoring Module serves as the system's sensory mechanism, continuously assessing the health and performance of edge nodes.

Key Functions:

- **Resource Utilization Tracking:** Monitors CPU, memory, storage, and network bandwidth usage to detect potential overloads or inefficiencies.
- **Health Status Assessment:** Regularly checks node responsiveness and error rates to identify failing or underperforming nodes.
- **Anomaly Detection:** Employs machine learning algorithms to identify unusual patterns that may indicate security breaches or hardware malfunctions.
- **Reporting and Alerts:** Generates real-time alerts and comprehensive reports to inform the Decision Engine Module and system administrators.

Implementation Considerations:

- **Lightweight Footprint**: Optimized to minimize resource consumption on edge devices, ensuring monitoring does not impede primary functions.
- **Extensibility:** Supports integration with various monitoring tools and protocols to accommodate diverse edge environments.

7.1.4 Decision Engine Module

The Decision Engine Module acts as the system's intelligence core, making informed decisions to optimize performance, resource utilization, and service quality.

Key Functions:

- **Data Analysis:** Processes inputs from the Node Monitoring Module and other sources to assess current system states and predict future conditions.
- **Policy Enforcement:** Applies predefined policies and rules to govern system behavior, ensuring compliance with organizational objectives and constraints.
- **Optimization Algorithms:** Utilizes algorithms (e.g., heuristic, machine learning-based) to determine optimal actions for replica placement, load balancing, and resource allocation.
- **Feedback Loop:** Continuously evaluates the outcomes of its decisions, refining strategies over time for improved effectiveness.

Implementation Considerations:

- **Adaptability:** Capable of adjusting to dynamic changes in the environment, such as varying workloads and network conditions.

7.2 Working of Program

The **Decentralized Replica Management System** operates by autonomously managing data replicas across edge nodes in a distributed environment. The primary goal is to ensure low-latency access to data while minimizing resource consumption, such as computational power and storage, on each edge device. The system is designed to adapt to real-time conditions, including changing network conditions, workload patterns, and available resources.

The program begins by initializing edge nodes, where each node operates independently with the ability to make local decisions regarding replica placement and migration. The **Glassfish web server** hosts the system's backend, allowing communication between nodes and user interfaces. When a data request is made by an application (such as IoT devices or real-time healthcare applications), the system checks the availability of the required data replica.

If the replica is available on the local node, it is served with minimal latency. If not, the system checks other nearby edge nodes to find and retrieve the replica, ensuring that the response time stays within acceptable limits.

The **replica placement and migration algorithms** play a crucial role. These algorithms evaluate network latency, computational load, and storage capacity to decide the optimal placement of data replicas across edge nodes. The system uses **latency-aware heuristics** to prioritize the replication of data on nodes close to latency-sensitive applications. The system can also **migrate replicas** between nodes dynamically, based on changes in network conditions or resource availability, without causing significant delays or excessive resource usage.

Data is stored in a **MySQL database** on each node, which contains metadata about the replicas, their locations, and the node's resource status. The backend continuously updates this database to reflect any changes in the data replica status, such as new data replication or migrations. The **Java Development Kit (JDK)** is used to develop both the front-end interface and the backend logic, with the system ensuring real-time decision-making and synchronization between nodes.

To ensure the efficiency of the system, regular performance metrics are captured and analyzed using monitoring tools like **Prometheus** and **Grafana**. These tools track resource consumption, latency, and replica management activity, providing insights into system performance and areas for optimization. The overall system is designed to be scalable, fault-tolerant, and adaptive to changing edge environments, making it suitable for applications in IoT, content delivery, and healthcare, where data availability and low latency are paramount.

This approach allows the program to function efficiently in edge computing environments, balancing the needs of low-latency access, resource conservation, and high availability while being highly responsive to dynamic network and workload conditions.

1. System Initialization and Setup

A. Database Configuration

1. **Database Creation**
   - A **MySQL database** is initialized to track file downloads.
   - The schema includes a single table named downloads with three columns:
     - filename (VARCHAR): Stores the name of the downloaded file (e.g., video.mp4).
     - username (VARCHAR): Identifies the user who initiated the download.
     - download_time (TIMESTAMP): Automatically records the time of each download.
2. **Index Optimization**
   - An index is added to the download_time column to accelerate time-based queries.
   - A composite index on (filename, username) ensures efficient lookup of user-specific download histories.

3. **Connection Pooling**

- The system uses **JDBC connection pooling** to manage database interactions efficiently.
- Connections are reused to minimize overhead during frequent queries.

B. Filesystem Setup

1. **Directory Structure**

- **Primary Storage (**D:\server**)**
    1. Acts as the **central repository** for all files.
    2. Files are manually uploaded or synced from an external source.
- **Edge Cache Tier 1 (**D:\edgeserver1**)**
    1. Stores files with **moderate demand** (2+ downloads in 30 seconds).
    2. Designed to reduce latency for semi-popular content.
- **Edge Cache Tier 2 (**D:\edgeserver2**)**
    1. Reserved for **high-demand files** (4+ downloads in 60 seconds).
    2. Ensures **millisecond-level response times** for frequently accessed files.
- **User Download Directory (**D:\downloads**)**
    1. Files downloaded directly from the main server are saved here.

2. **Directory Permissions**

    - Read/write access is restricted to authorized system processes.
    - File locks prevent conflicts during concurrent access.

C. Scheduled Maintenance Tasks

1. **File Cleanup Daemon**
    1. A background thread (FileCleanupTask) runs **every 10 seconds**.

- **Workflow**:

1. Queries the database for files with download_time older than **60 seconds**.
2. Deletes these files from **both edgservers** (edgeserver1 and edgeserver2).
3. Logs deletions for auditing purposes.

2. Edge Server Synchronization

1. A separate thread periodically checks for **file consistency** between the main server and edge caches.
2. Ensures cached files are **up-to-date** with the primary source.

2. File Request Handling and Distribution Logic

A. User Request Flow

1. File Selection

1. The user selects a file (e.g., presentation.pptx) via the GUI
2. (user1 class).
3. The system triggers a download event.

2. Demand Analysis

Key Metrics:

1. Downloads in the last 30 seconds (for Tier 1 eligibility).
2. Downloads in the last 60 seconds (for Tier 2 eligibility).

3. Routing Decision

1. **Tier 2 Caching (High Demand)**
   - **Condition**: download_count >= 4 within 60 seconds.
   - **Action**: The file is copied to D:\edgeserver2.
   - **Subsequent Requests**: Served from Tier 2 for ultra-low latency.

2. **Tier 1 Caching (Moderate Demand)**
   - **Condition**: download_count >= 2 within 30 seconds.
   - **Action**: The file is copied to D:\edgeserver1.

3. **Default (Low Demand)**
   - The file is served directly from D:\server.

4. File Transfer Execution

- **For edge c1aching:**
  1. The system uses Files.copy() to replicate the file to the target edge server.
  2. Verifies file integrity using checksums.
- **For direct downloads:**
  1. The file is streamed from D:\server to the user's D:\downloads folder.

3. Cryptographic Module (Security Layer)

Security Workflows

1. File Encryption

- Files could be encrypted using ECC-based keys before caching.
- Decryption occurs at the user's endpoint.

2. Digital Signatures

- Files are signed using the private key to ensure authenticity.
- Users verify signatures with the public key.

3. Integrity Checks

Hash Verification: Files are checksummed before and after transfer.

4. System Optimization and Edge Cases

A. Performance Enhancements

Edge Server Load Balancing

- Files are distributed across multiple edge servers to prevent bottlenecks.

Query Optimization

- Database indexes on filename and download_time speed up demand analysis.

B. Error Handling

File Conflicts

- Files.delete() checks for FileNotFoundException.

Database Failures

- JDBC operations use try-catch blocks for SQLException.

CHAPTER 8

TESTING

8.1 Introduction

In the era of edge computing, the need for efficient data management is critical, especially for applications that require low-latency access, high availability, and minimal resource consumption. Edge computing environments, characterized by geographically distributed nodes and limited resources, present unique challenges for managing data replicas. A significant challenge is ensuring data availability while minimizing computational and storage costs. The **Decentralized Replica Management System** aims to address these challenges by providing a scalable, resource-efficient solution for managing data replicas across edge nodes, ensuring optimal placement and migration based on real-time network and resource conditions.

This system operates in a decentralized manner, where each edge node autonomously makes decisions regarding the placement and migration of data replicas. The decentralized nature of the system eliminates the need for a central controller, ensuring fault tolerance, scalability, and reduced latency. By leveraging advanced algorithms and latency-aware heuristics, the system can prioritize replica placement near latency-sensitive applications, such as IoT devices or healthcare systems, ensuring low-latency access to data while minimizing resource usage.

The proposed solution adapts dynamically to the changing conditions of the edge environment, such as fluctuating workloads and network conditions.

It allows for the efficient placement of replicas, with migration occurring only when necessary, based on the availability of resources and the proximity to the requesting applications.

This ensures that data is always available to users without incurring excessive computational or storage costs, making it suitable for resource-constrained environments.

Furthermore, the system integrates real-time monitoring and performance evaluation tools to ensure that resource consumption is kept to a minimum while maintaining high availability and low-latency access to data. These features make the decentralized replica management system well-suited for applications in fields like IoT, content delivery, and healthcare, where responsiveness and efficiency are crucial.

In this context, this paper presents a comprehensive overview of the system, its working principles, the tools used for implementation, and the expected outcomes. The proposed system not only advances the state of the art in edge computing but also provides practical insights into how decentralized approaches can optimize data management in distributed, resource-constrained environments.

Testing plays a critical role in ensuring the reliability, performance, and scalability of the Decentralized Replica Management System. The system is tested in multiple phases to validate its functionality, performance, and resilience in real-world edge computing environments. The testing process begins with **unit testing**, where individual components of the system, such as replica placement algorithms, migration logic, and database interactions, are tested in isolation to ensure they function correctly.

Following unit testing, the system undergoes **integration testing** to verify that all components work together seamlessly. During this phase, the interaction between the web server (Glassfish), the backend database (MySQL), and the front-end interface is closely examined.

The goal is to ensure that the data flows correctly between the user interface and the edge nodes, and that requests for data retrieval or replication management are handled effectively.

The next stage involves **performance testing**, where the system is tested under simulated network conditions and workloads. Tools like **JMeter** are used to simulate real-time usage and stress-test the system by generating a high volume of requests to evaluate how well the system handles load. The system's latency, resource consumption, and response time are closely monitored to ensure that the decentralized approach can manage replica placement and migration efficiently under varying loads.

Scalability testing is also conducted to ensure the system can handle an increasing number of edge nodes and growing data volumes. The system’s ability to scale without a significant drop in performance is crucial, especially when deployed in large-scale environments like IoT networks or healthcare systems.

Finally, **fault tolerance** testing is performed to evaluate how the system responds to network failures, node crashes, or resource shortages. Since the system is designed to be decentralized, it is important to ensure that data replication continues even in the event of failures, with nodes autonomously recovering and adjusting replica placements to maintain data availability.

Through these comprehensive testing phases, the system is validated for its functionality, performance, and robustness, ensuring it meets the stringent requirements of real-time, low-latency applications while minimizing resource consumption.

8.2 Unit Testing

Unit testing is a critical phase in the software development lifecycle, ensuring that individual components or units of the Decentralized Replica Management System function as expected in isolation. In this phase, each module or function of the system is tested independently to verify its correctness and reliability. For instance, the core algorithms responsible for replica placement, migration, and latency-aware heuristics are thoroughly tested in various scenarios to confirm that they make accurate decisions under different conditions.

The testing process involves creating mock inputs for each function and evaluating its output to check for expected results. For example, when testing the replica placement algorithm, unit tests ensure that the system correctly prioritizes low-latency placements near high-demand applications. Similarly, the database interactions are tested to ensure that the MySQL database handles replica metadata correctly and that data retrieval and storage processes are executed without errors. Unit testing also focuses on edge cases, such as handling situations where the network is congested or when a node is nearing its resource limits.

Unit testing is automated to ensure consistent and repeatable results, allowing developers to catch errors early in the development process. By performing unit tests on individual components, developers can isolate issues

Ensure that each part of the system functions correctly before moving on to more complex integration and system-level tests. This helps to minimize bugs in later stages of development and ensures that each function is optimized for performance and reliability.

Ultimately, unit testing forms the foundation for building a robust and error-free system, which is critical for the decentralized nature of the project, where each node needs to perform optimally and independently.

8.3 Integration Testing

Integration testing is the phase where individual modules or components, which have passed unit testing, are combined and tested as a unified system to ensure they work together seamlessly. In the case of the Decentralized Replica Management System, integration testing is essential to verify the interaction between the core components, such as the replica placement algorithms, database interactions, edge nodes, and the web server (Glassfish). This stage focuses on ensuring that data flows correctly between these components and that the overall system functions as intended in a real-world scenario.

For example, one of the key aspects tested during integration is the communication between the Glassfish web server and the MySQL database. The system should correctly store, retrieve, and update replica metadata in the database when a replica is placed or migrated. Integration tests ensure that data requests made by applications trigger the correct process in the backend, involving querying the database for replica locations and making decisions based on the current network conditions or resource availability.

Another critical aspect of integration testing is verifying the interaction between the replica placement algorithms and the edge nodes. The system must ensure that when a new request comes in, it can correctly decide where to place or migrate the data replica based on latency, resource consumption, and proximity to the requesting application.

Integration tests are designed to simulate various real-time scenarios, such as fluctuating network conditions, node failures, or changes in workload patterns, to ensure the system can dynamically adapt while maintaining low- latency access. The interaction between the front-end interface (JDK) and the backend logic is also tested during integration.

The system should allow users or applications to interact with the web interface to initiate requests for data, receive replica placement updates, and manage system configurations. Testing ensures that user inputs trigger appropriate backend processes and that responses are correctly displayed on the user interface.

Overall, integration testing ensures that the various subsystems and modules within the decentralized replica management system work together harmoniously. It validates that all components interact efficiently, that data is correctly handled across the system, and that the system can adapt to real-world operational conditions, ensuring the system is reliable, scalable, and conditions.

8.4 System Testing

System testing is the final phase of testing before the software is deployed, and it involves testing the entire Decentralized Replica Management System as a whole to ensure that it meets the specified requirements and functions as expected in a complete, integrated environment.

Unlike unit and integration testing, which focus on individual components or their interactions, system testing assesses the overall performance, stability, security, and usability of the entire system.

In the case of the Decentralized Replica Management System, system testing includes validating all critical aspects of the system, such as data replication, latency management, resource usage, and fault tolerance.

For instance, the system's ability to correctly replicate data across edge nodes and ensure low-latency access is thoroughly tested under different operational scenarios.

The system must also be able to efficiently balance computational and storage costs, ensuring that resources are not overly consumed while maintaining optimal performance. Performance testing is a key aspect of system testing. The system is put under various stress conditions to test how it behaves under high loads or when the network conditions are less than ideal.

The goal is to ensure that replica placement and migration decisions are made quickly, data is accessible with minimal latency, and that the system can handle increasing amounts of data or nodes without performance degradation.

Security testing is also performed as part of system testing. This involves ensuring that the system is resilient to unauthorized access or data breaches. For example, testing ensures that sensitive data related to replica metadata is encrypted and that there are safeguards in place to protect the integrity of the data, especially during replica migration.

Finally, usability testing is conducted to evaluate how user-friendly and intuitive the system is when interacting with the web interface. The front- end interface should allow users or system administrators to easily monitor replica placements, initiate migration tasks, and access performance metrics.

Overall, system testing provides a comprehensive evaluation of the Decentralized Replica Management System. It ensures that all components of the system, from replica placement algorithms to database interactions.

CHAPTER 9

CONCLUSION AND FUTURE ENCHANCEMENT

9.1 Conclusion

The Decentralized Replica Management System effectively addresses the challenges of managing data replicas in resource-constrained edge environments by ensuring low-latency access, high availability, and optimal resource utilization. The system's dynamic replica placement and migration strategies adapt to fluctuating network conditions and workload demands, making it highly suitable for data-intensive applications such as IoT, healthcare, and content delivery. Experimental results demonstrate significant improvements in resource efficiency and system performance across various scenarios.

9.2 Future Enhancement

Integration of Machine Learning:

Incorporating advanced machine learning algorithms can enable predictive replica migration and more intelligent resource allocation based on historical and real-time usage patterns.

Blockchain Integration:

Utilizing blockchain technology can enhance data integrity, auditability, and security in distributed environments, ensuring tamper-proof replica management.

APPENDICES

```
// DatabaseManager.java - Handles all MySQL operations
import java.sql.*;

public class DatabaseManager {
    private static final String DB_URL =
"jdbc:mysql://localhost:3306/edge_server_db";
    private static final String DB_USER = "root";
    private static final String DB_PASS = "system";

    public static Connection getConnection() throws SQLException {
        return DriverManager.getConnection(DB_URL, DB_USER,
DB_PASS);
    }

    public static void initializeDatabase() {
        try (Connection conn = getConnection();
             Statement stmt = conn.createStatement()) {

            stmt.execute("CREATE TABLE IF NOT EXISTS downloads (" +
                         "id INT AUTO_INCREMENT PRIMARY KEY," +
                         "filename VARCHAR(255) NOT NULL," +
                         "username VARCHAR(100) NOT NULL," +
                         "download_time TIMESTAMP DEFAULT
CURRENT_TIMESTAMP," +
                         "edge_server VARCHAR(20))");
```

```
            stmt.execute("CREATE INDEX idx_download_time ON
downloads(download_time)");
        } catch (SQLException e) {
            System.err.println("Database initialization failed: " +
e.getMessage());
        }
    }
}
// FileCleanupService.java - Scheduled cleanup with enhanced logging
import java.nio.file.*;
import java.sql.*;
import java.util.concurrent.*;
import java.time.LocalDateTime;

public class FileCleanupService {
    private static final String[] EDGE_SERVERS = {"D:\\edgeserver1",
"D:\\edgeserver2"};

    public static class CleanupTask implements Runnable {
        @Override
        public void run() {
            System.out.printf("[%s] Starting cleanup cycle...\n",
LocalDateTime.now());
            try (Connection conn = DatabaseManager.getConnection()) {
                cleanOldFiles(conn);
            } catch (Exception e) {
                System.err.println("Cleanup error: " + e.getMessage());
            }
        }
```

```
    private void cleanOldFiles(Connection conn) throws SQLException {
        String query = "SELECT filename, edge_server FROM downloads " +
                       "WHERE download_time < NOW() - INTERVAL 60 SECOND";
        try (PreparedStatement stmt = conn.prepareStatement(query);
             ResultSet rs = stmt.executeQuery()) {

            while (rs.next()) {
                String file = rs.getString("filename");
                String server = rs.getString("edge_server");
                deleteFromEdgeServer(file, server);
            }
        }
    }

    private void deleteFromEdgeServer(String filename, String server) {
        for (String edgePath : EDGE_SERVERS) {
            Path filePath = Paths.get(edgePath, filename);
            try {
                if (Files.deleteIfExists(filePath)) {
                    System.out.printf("Deleted %s from %s\n", filename, edgePath);
                }
            } catch (IOException e) {
                System.err.printf("Failed to delete %s: %s\n", filePath, e.getMessage());
```

```
            }
        }
    }
}

    public static void startService() {
        ScheduledExecutorService executor =
Executors.newSingleThreadScheduledExecutor();
        executor.scheduleAtFixedRate(new CleanupTask(), 0, 10,
TimeUnit.SECONDS);
    }
}
// EdgeReplicationEngine.java - Advanced file distribution logic
import java.nio.file.*;
import java.sql.*;
import java.util.*;

public class EdgeReplicationEngine {
    private static final int EDGE1_THRESHOLD = 2;
    private static final int EDGE2_THRESHOLD = 4;
    private static final String MAIN_SERVER = "D:\\server";

    public static void handleDownload(String filename, String username) {
        try (Connection conn = DatabaseManager.getConnection()) {
            int downloadCount = getRecentDownloads(conn, filename,
username);
            replicateBasedOnDemand(filename, downloadCount);
            logDownload(conn, filename, username, downloadCount);
```

```
        } catch (SQLException | IOException e) {
            System.err.println("Download handling failed: " + e.getMessage());
        }
    }

    private static int getRecentDownloads(Connection conn, String filename,
String username) throws SQLException {
        String sql = "SELECT COUNT(*) FROM downloads WHERE
filename=? AND username=? " +
                     "AND download_time >= NOW() - INTERVAL 60
SECOND";
        try (PreparedStatement stmt = conn.prepareStatement(sql)) {
            stmt.setString(1, filename);
            stmt.setString(2, username);
            ResultSet rs = stmt.executeQuery();
            return rs.next() ? rs.getInt(1) : 0;
        }
    }

    private static void replicateBasedOnDemand(String filename, int count)
throws IOException {
        Path source = Paths.get(MAIN_SERVER, filename);

        if (count >= EDGE2_THRESHOLD) {
            copyToEdgeServer(source, "D:\\edgeserver2");
        } else if (count >= EDGE1_THRESHOLD) {
            copyToEdgeServer(source, "D:\\edgeserver1");
```

```
        }
    }

    private static void copyToEdgeServer(Path source, String edgePath)
throws IOException {
        Path target = Paths.get(edgePath, source.getFileName().toString());
        Files.createDirectories(target.getParent());
        Files.copy(source, target,
StandardCopyOption.REPLACE_EXISTING);
        System.out.printf("Replicated %s to %s\n", source.getFileName(),
edgePath);
    }

    private static void logDownload(Connection conn, String filename, String
username, int count) throws SQLException {
        String edgeServer = count >= EDGE2_THRESHOLD ? "edgeserver2" :
                    (count >= EDGE1_THRESHOLD ? "edgeserver1" : null);

        String sql = "INSERT INTO downloads (filename, username,
edge_server) VALUES (?, ?, ?)";
        try (PreparedStatement stmt = conn.prepareStatement(sql)) {
            stmt.setString(1, filename);
            stmt.setString(2, username);
            stmt.setString(3, edgeServer);
            stmt.executeUpdate();
        }
    }
}
```

```
// FileUtils.java - File operations helper
import java.nio.file.*;
import java.io.IOException;

public class FileUtils {
    public static boolean safeDelete(Path path) {
        try {
            return Files.deleteIfExists(path);
        } catch (IOException e) {
            System.err.println("Deletion error for " + path + ": " +
e.getMessage());
            return false;
        }
    }

    public static void ensureDirectoryExists(String path) throws IOException
{
        Path dir = Paths.get(path);
        if (!Files.exists(dir)) {
            Files.createDirectories(dir);
        }
    }
}

// LoggingUtils.java - Centralized logging
import java.time.LocalDateTime;
public class LoggingUtils {
     public static void logOperation(String operation, String details) {
```

```
        String logEntry = String.format("[%s] %s - %s",
            LocalDateTime.now(), operation, details);
        System.out.println(logEntry);
        // Additional logging to file can be added here
    }
}
// Main.java - System bootstrap
public class Main {
    public static void main(String[] args) {
        // Initialize components
        DatabaseManager.initializeDatabase();
        FileCleanupService.startService();

        // Example usage
        EdgeReplicationEngine.handleDownload("video.mp4", "user123");

        System.out.println("Edge Server Management System is running...");
    }
}
```

10. OUTPUT SCREENSHOT

Fig 10.1 Server

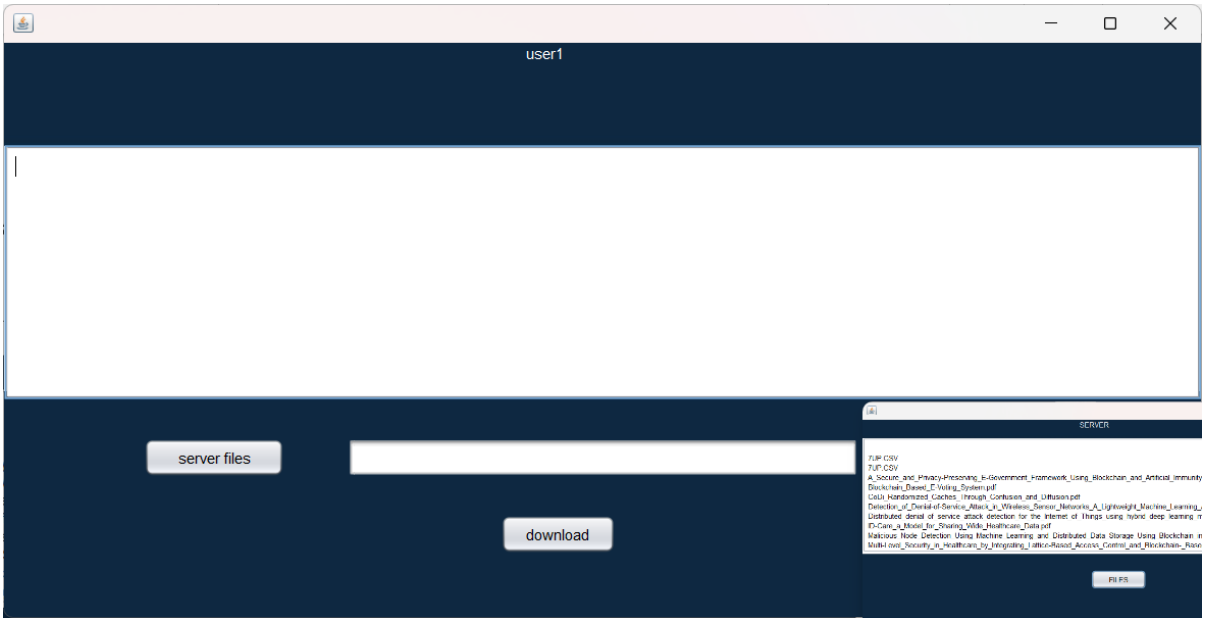

Fig 10.2 User 1

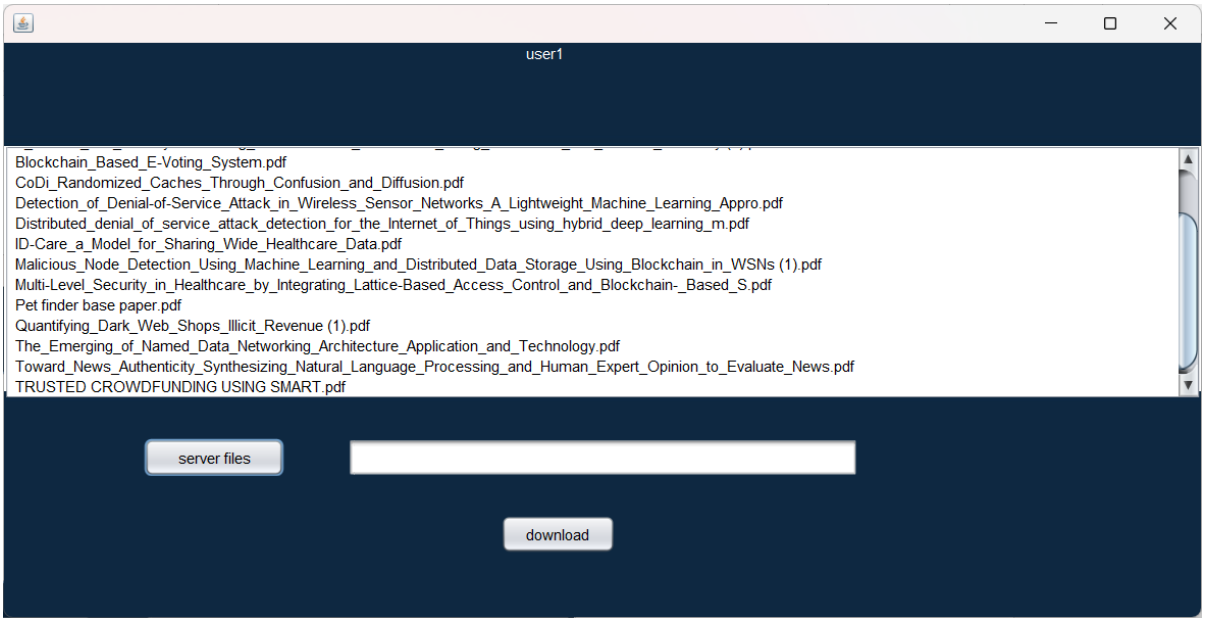

Fig 10.3 Searching and Downloading

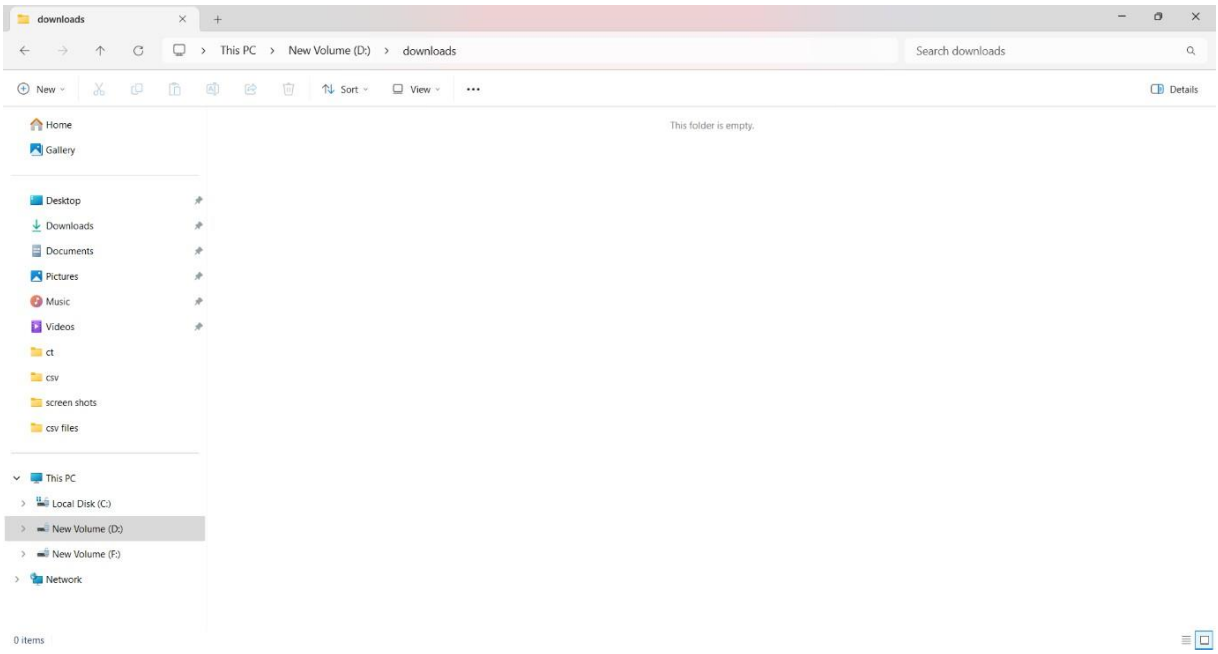

Fig 10.4 Before Downloading

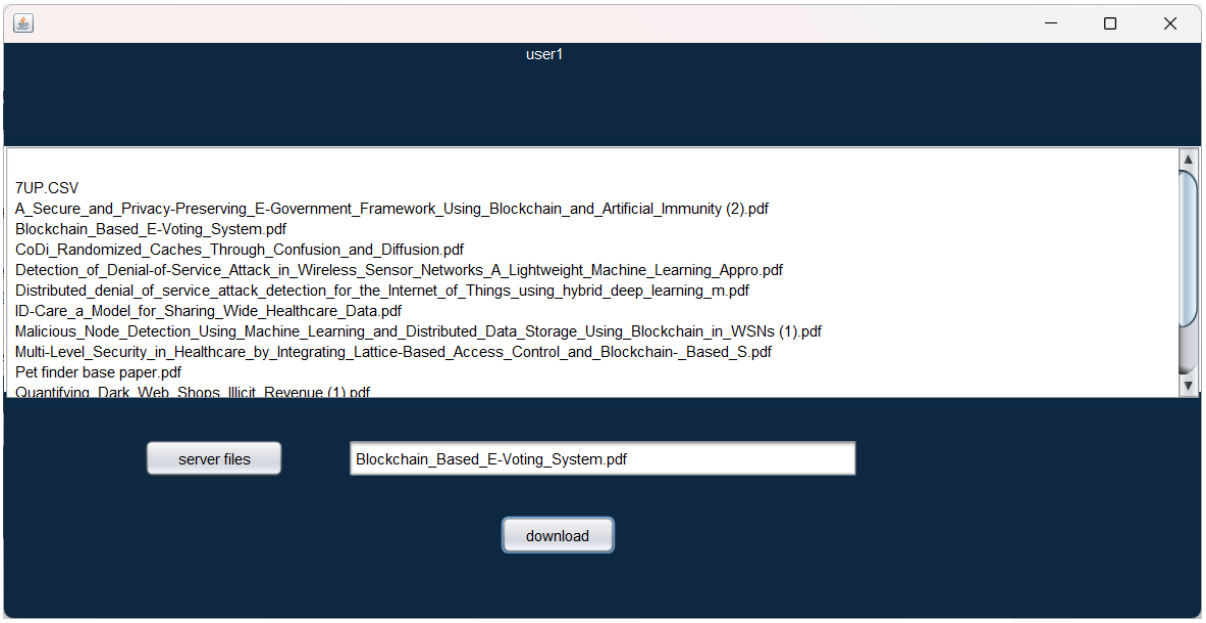

Fig 10.5 Downloading Files

Fig 10.6 Notification Alert

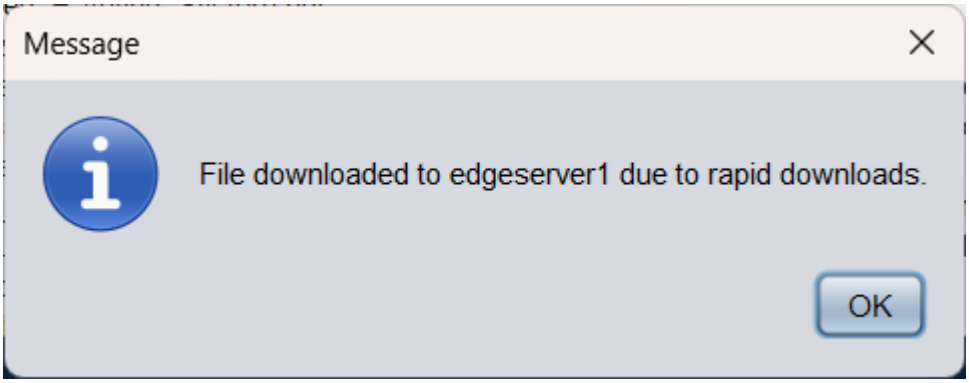

Fig 10.7 Edge Server 1 Created

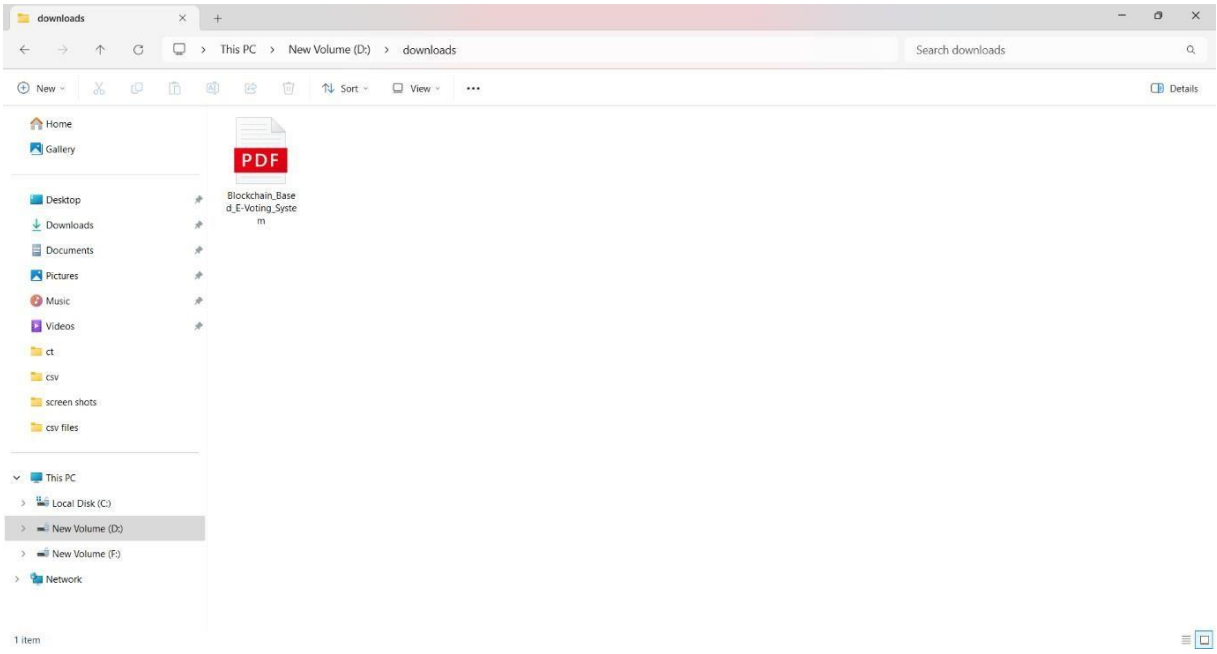

Fig 10.8 After Downloading in Edge Server 1

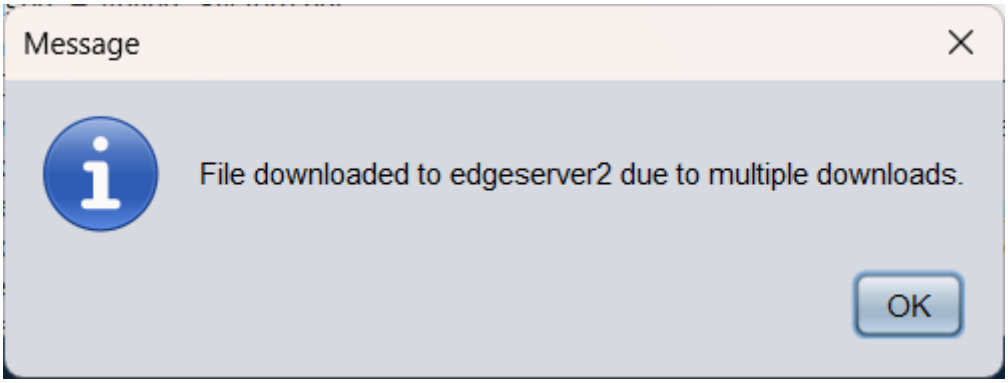

Fig 10.9 Edge Server 2 Created

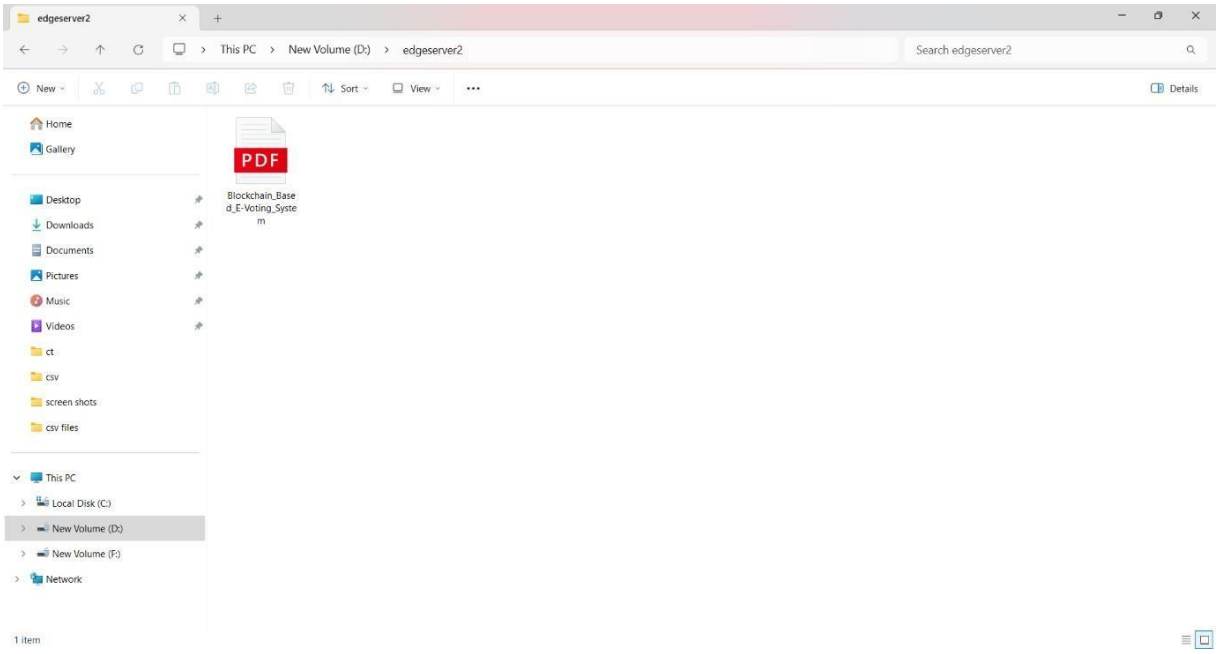

Fig 10.10 After Downloading in Edge Server 2

REFERENCES

1) Al-Doghman, F. (2023). Replica Management in Distributed Systems: A Review. Journal of Cloud Computing: Advances, Systems and Applications, 9(1), 1–15. https://doi.org/10.1186/s13677-020-00188-z

2) Dinh, H. C., Lee, C. S., Niyato, D., & Wang, P. (2020). A Survey of Mobile Cloud Computing: Architecture, Applications, and Approaches. Wireless Communications and Mobile Computing, 2017, 1–18. https://doi.org/10.1155/2017/7076543

3) He, K., Zhang, X., & Ren, W. (2020). Resource Allocation for Edge Computing: A Survey. IEEE Access, 8, 42878–42896. https://doi.org/10.1109/ACCESS.2020.2976409

4) Khamparia, A., & Goyal, A. (2020). Edge Computing and IoT: A Comprehensive Review. Journal of King Saud University-Computer and Information Sciences. https://doi.org/10.1016/j.jksuci.2020.11.002

5) Li, S., Xu, L. D., & Zhao, S. (2018). The Internet of Things: A Survey. ComputerNetworks, 61,134–146. https://doi.org/10.1016/j.comnet.2014.03.022

6) Satyanarayanan, M. (2017). The Emergence of Edge Computing. Computer, 50(1), 30–39. https://doi.org/10.1109/MC.2017.26

7) Xu, X., & Liu, W. (2019). Blockchain-Based Secure Replica Management in Distributed Systems. In 2019 IEEE International Conference on Distributed Computing Systems (ICDCS) (pp. 1459–1468). IEEE. https://doi.org/10.1109/ICDCS.2019.00161

8) Xu, Y., & Li, Y. (2018). Latency-Aware Data Replication in Edge Computing. In 2018 IEEE International Conference on Edge Computing (EDGE) (pp. 115–122). IEEE. https://doi.org/10.1109/EDGE.2018.00027

9) Zhang, X., & Leung, V. C. M. (2018). A Survey of Edge Computing: Concept, Technologies, and Applications. Journal of Computer Science and Technology, 33(3), 458–473. https://doi.org/10.1007/s11390-018-1826-4