**SIMATS ENGINEERING**

***CAPSTONE PROJECT REPORT***

**PROJECT TITLE**

**Fault-Tolerant Distributed File Systems**

***DSA0199-Object Oriented Programming with C++ in Using Encapsulation***

*Submitted by*

R.Sanjai

(192211425)

R.Ranjith

(192211667)

*Under the supervision of*

*S.Jayanthi*

*Department of Information Security*

*Department of Computer Science and Engineering*

**DECLARATION**

We, R.Sanjai and R. Ranjith, are the students of Bachelor of Engineering in the Department of Computer Science and Engineering, Saveetha Institute of Medical and Technical Sciences, Saveetha University, Chennai. We hereby declare that the work presented in this Capstone Report for the Object Oriented Programming with C++ in Using Encapsulation (DSA0199) entitled “” is the outcome of our own work and is correct to the best of our knowledge and understanding.

Ranjith.R (192211667)

Sanjai.R (192211425)

Date: 22/09/2024

Place: Chennai

**ABSTRACT:**

Fault-tolerant distributed file systems are essential for ensuring data reliability and availability across distributed computing environments. These systems use various techniques to handle hardware and software failures, making them resilient to data loss or corruption. This abstract introduces the concept of a fault-tolerant distributed file system and highlights its importance in modern distributed computing.

**INTRODUCTION:**

In distributed computing environments, ensuring data availability and reliability is a significant challenge, especially in the face of system failures. Fault-tolerant distributed file systems (FTDFS) are designed to provide robust data storage solutions that can recover from failures, ensuring continuous service and minimal data loss. The design of these systems requires techniques that handle replication, redundancy, and failover mechanisms to avoid single points of failure.

The choice of fault-tolerant techniques impacts several critical factors, including:

* **Data Availability**: Ensures that the system remains operational even when certain components fail.
* **Reliability**: Guarantees that data is preserved and accessible in the event of failures.
* **Performance**: Efficient mechanisms allow the system to continue functioning without significant performance degradation during failures.

This introduction provides an overview of FTDFS, setting the stage for a detailed exploration of various fault-tolerance techniques and their impact on performance, reliability, and scalability.

**Fault-Tolerant Distributed File System: Module-wise Description**

**Module 1: Fundamentals of Distributed File Systems**

* **Introduction to DFS**: Concepts of a distributed file system, its architecture, and use cases.
* **Challenges in DFS**: Scalability, consistency, replication, and fault tolerance.
* **Overview of Fault Tolerance**: Understanding the need for fault tolerance in distributed systems.

**Module 2: Data Replication**

* **Replication Strategies**: How data is replicated across multiple nodes for fault tolerance.
  + **Synchronous vs. Asynchronous Replication**: Trade-offs between consistency and performance.
  + **Master-Slave Replication**: A master node handles writes, while slave nodes replicate the data.
  + **Multi-Master Replication**: Multiple nodes can handle write requests, improving reliability.
* **Advantages**: Improved availability, data redundancy.
* **Challenges**: Maintaining consistency between replicas, dealing with replication lag.

**Module 3: Consistency Models**

* **Eventual Consistency**: The system guarantees that, given enough time, all replicas will converge to the same state.
* **Strong Consistency**: Ensures that all replicas are synchronized at any given time, at the cost of performance.
* **Quorum-Based Systems**: Decisions (like reads/writes) are made based on majority voting among replicas.

**Module 4: Failure Detection and Recovery**

* **Failure Models**: Types of failures (crash, Byzantine failures, etc.) that can occur in distributed systems.
* **Heartbeat Mechanisms**: Monitoring nodes' health by sending periodic signals to detect failure.
* **Failover Mechanisms**: Switching to a backup node in the event of a primary node failure.
* **Data Recovery**: Techniques for ensuring that data is not lost after a node or system failure.
  + **Checkpoints and Logs**: Capturing system state to recover after failures.
  + **Replication Repair**: Rebuilding lost replicas through data recovery from healthy nodes.

**Module 5: Distributed File System Architectures**

* **Master-Slave Architecture**: A centralized approach where a master coordinates file access and replication.
* **Peer-to-Peer Architecture**: Nodes are equal, and data is distributed without a central controller.
* **Cluster-Based Architecture**: Data is distributed across a cluster of nodes, often with a dedicated metadata service.

**Module 6: Load Balancing and Data Placement**

* **Dynamic Data Placement**: Placing data across nodes to ensure balanced load and prevent bottlenecks.
* **Distributed Hash Tables (DHTs)**: A common mechanism for determining data placement in distributed systems.
* **Data Sharding**: Dividing data into smaller pieces (shards) distributed across multiple nodes for improved fault tolerance.

**Module 7: Real-World Examples**

* **Google File System (GFS)**: An overview of GFS's approach to fault tolerance using replication and failover techniques.
* **Hadoop Distributed File System (HDFS)**: Focus on block replication and its role in fault tolerance.
* **Ceph**: A peer-to-peer fault-tolerant file system with no single point of failure.

**Module 8: Security and Fault Tolerance**

* **Data Encryption**: Ensuring that even if a node is compromised, the data remains secure.
* **Access Control and Auditing**: Maintaining fault tolerance while ensuring that only authorized users can access the system.

**About The Software Programming:**

Dev-C++ is a free and open-source **Integrated Development Environment (IDE)** specifically designed for programming in the C and C++ languages. It's particularly popular among beginners due to its user-friendly interface and ease of use. Here's a closer look at Dev-C++:

**Key Features:**

* **Free and Open-source:** Anyone can download and use Dev-C++ for free, and its source code is available for modification and contribution.
* **Lightweight and Portable:** Being a native Windows application, Dev-C++ has a relatively small footprint and doesn't require complex installations.
* **Supports GCC Compilers:** Dev-C++ utilizes the MinGW port of the GNU Compiler Collection (GCC) for compiling C and C++ code. This ensures compatibility with a wide range of libraries and tools.
* **Graphical User Interface (GUI):** Dev-C++ offers a user-friendly GUI with features like syntax highlighting, code completion, and project management. These features improve code readability, reduce errors, and streamline development workflows.
* **Debugging:** Dev-C++ integrates a debugger that allows you to step through your code line by line, identify errors, and analyze program behavior.
* **Customizable:** Dev-C++ allows for customization of the editor interface, syntax highlighting themes, and keyboard shortcuts to suit your preferences.
* **Project Management:** The IDE provides features for managing multiple projects, organizing files, and building complex applications.
* **Extensible:** Dev-C++ supports plugins and extensions, allowing you to add functionalities like additional language support or specialized development tools.

**Who is it for?**

* **Beginners:** Due to its user-friendly interface, clear code organization, and focus on C and C++, Dev-C++ is a popular choice for students and individuals learning these programming languages for the first time.
* **Hobbyists and Personal Projects:** The free and open-source nature of Dev-C++ makes it suitable for hobbyists and personal projects where budget might be a constraint.

**Things to Consider:**

* **Limited Functionality Compared to Professional IDEs:** While Dev-C++ provides the essentials for C and C++ development, it might lack some advanced features and functionalities found in professional IDEs like code refactoring, advanced debugging tools, and extensive code analysis capabilities.
* **Potentially Outdated:** While still functional, Dev-C++'s development seems to have slowed down in recent years. This means it might not integrate seamlessly with the latest C++ standards or libraries.

Overall, Dev-C++ is a great starting point for those venturing into the world of C and C++ programming. Its user-friendly interface, essential features, and free availability make it an attractive option for beginners and hobbyists. However, if you're working on larger projects or require advanced functionalities, you might consider exploring professional IDEs that offer a broader range of development tools.

**CODE:**

#include <stdio.h>

#include <string.h>

#include <stdlib.h>

#define MAX\_NODES 3

#define MAX\_FILE\_SIZE 100

typedef struct {

char fileName[50];

char fileData[MAX\_FILE\_SIZE];

int nodeID;

} File;

typedef struct {

int nodeID;

File files[MAX\_FILE\_SIZE];

int fileCount;

} Node;

Node nodes[MAX\_NODES];

// Function to initialize the nodes

void initializeNodes() {

for (int i = 0; i < MAX\_NODES; i++) {

nodes[i].nodeID = i + 1;

nodes[i].fileCount = 0;

}

}

// Function to replicate file across all nodes

void replicateFile(char \*fileName, char \*data) {

for (int i = 0; i < MAX\_NODES; i++) {

strcpy(nodes[i].files[nodes[i].fileCount].fileName, fileName);

strcpy(nodes[i].files[nodes[i].fileCount].fileData, data);

nodes[i].files[nodes[i].fileCount].nodeID = nodes[i].nodeID;

nodes[i].fileCount++;

}

printf("File '%s' replicated successfully across all nodes.\n", fileName);

}

// Function to read the file from a specific node

void readFileFromNode(char \*fileName, int nodeID) {

for (int i = 0; i < nodes[nodeID - 1].fileCount; i++) {

if (strcmp(nodes[nodeID - 1].files[i].fileName, fileName) == 0) {

printf("Reading file '%s' from Node %d: %s\n", fileName, nodeID, nodes[nodeID - 1].files[i].fileData);

return;

}

}

printf("File '%s' not found on Node %d.\n", fileName, nodeID);

}

int main() {

initializeNodes();

char fileName[50], fileData[MAX\_FILE\_SIZE];

int choice, nodeID;

while (1) {

printf("\n1. Replicate File\n2. Read File from Node\n3. Exit\nEnter your choice: ");

scanf("%d", &choice);

switch (choice) {

case 1:

printf("Enter file name: ");

scanf("%s", fileName);

printf("Enter file data: ");

scanf("%s", fileData);

replicateFile(fileName, fileData);

break;

case 2:

printf("Enter file name to read: ");

scanf("%s", fileName);

printf("Enter node ID to read from (1 to %d): ", MAX\_NODES);

scanf("%d", &nodeID);

if (nodeID < 1 || nodeID > MAX\_NODES) {

printf("Invalid node ID. Please try again.\n");

} else {

readFileFromNode(fileName, nodeID);

}

break;

case 3:

printf("Exiting program.\n");

exit(0);

default:

printf("Invalid choice. Please try again.\n");

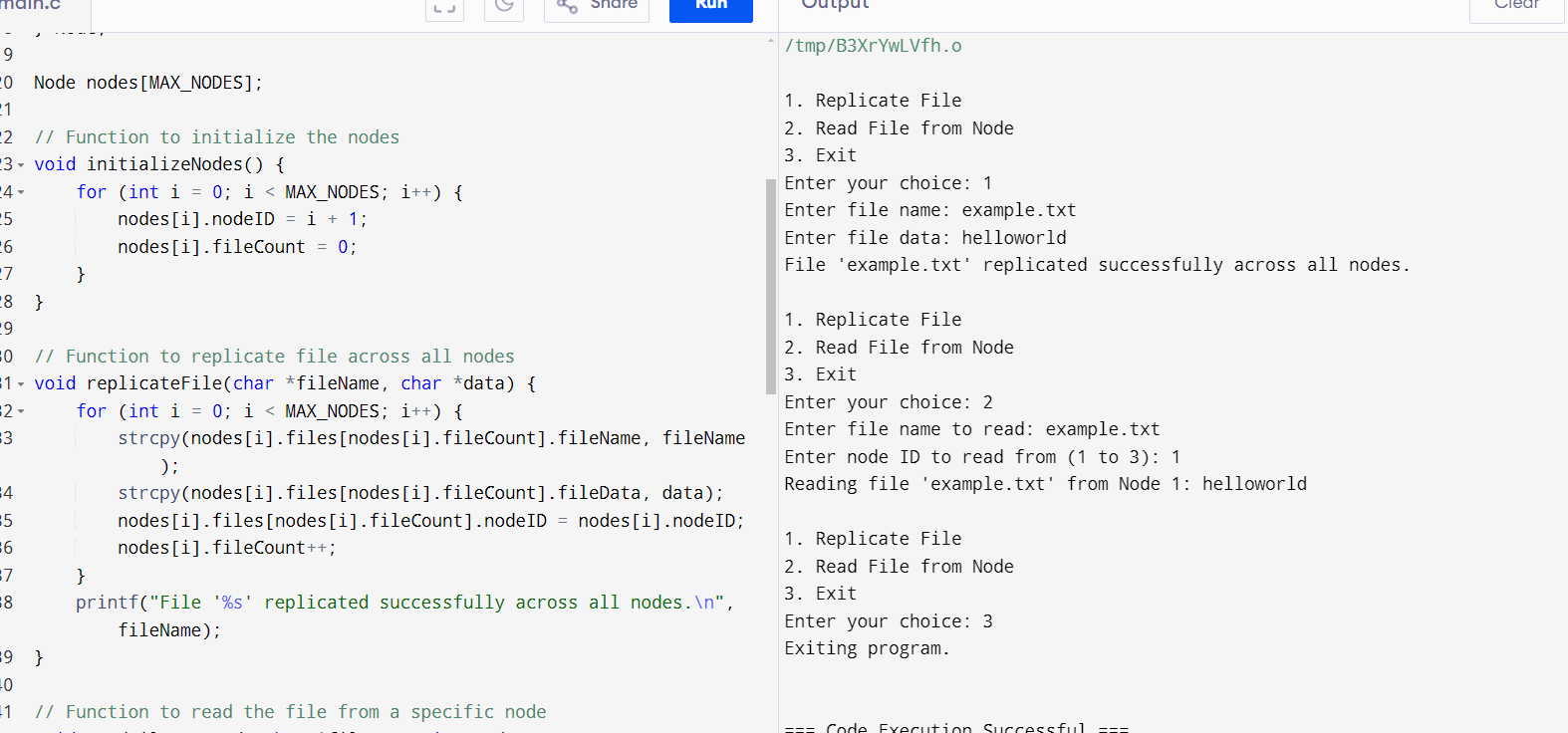
}

}

return 0;

}

**OUTPUT**



**CONCLUSION:**

In conclusion, fault-tolerant distributed file systems are essential for managing data in distributed computing environments. By employing replication, redundancy, and advanced failure detection techniques, these systems ensure data availability and reliability even in the face of failures. Understanding the trade-offs between performance, consistency, and fault tolerance enables developers to build systems that maximize efficiency while minimizing the risk of data loss.