FAULT-TOLERANT DISTRIBUTED MESSAGING SYSTEM: DESIGN AND IMPLEMENTATION

INTRODUCTION

In the evolving landscape of distributed computing, fault tolerance, consistency, and high availability have become indispensable properties for messaging systems. Distributed messaging platforms underpin critical applications ranging from real-time communication to data synchronization across large-scale infrastructures. Ensuring that these systems remain operational despite node failures, network disruptions, or data inconsistencies is essential for maintaining reliability and user trust.

This project presents the design and implementation of a simplified distributed messaging system developed in Java. The system emphasizes fundamental distributed systems principles by integrating message replication, a primary-backup architecture, and simulated consensus mechanisms inspired by the Raft algorithm. The primary-backup model ensures fault tolerance by duplicating incoming messages from a primary server to a backup server, providing redundancy in message storage and availability.

To approximate consensus and coordination, aspects of the Raft protocol such as log replication and leader election logic are simulated. Although implemented in a simplified manner, these components illustrate how leader-based consensus and coordinated state replication can maintain consistency across distributed nodes. This approach balances conceptual clarity and practical feasibility, making it suitable for educational purposes and as a foundation for further enhancements.

The relevance of this system lies in its demonstration of key distributed system challenges and solutions, including state replication, failure resilience, and agreement protocols. By focusing on essential mechanisms rather than full protocol complexity, the project provides valuable insight into how distributed messaging systems can be architected to achieve robustness and consistency in real-world deployments.

SYSTEM OVERVIEW

The distributed fault-tolerant messaging system is architected around two principal server nodes operating in a primary-backup configuration to ensure reliability and message durability. These servers run on separate network ports: the primary server listens on port **12345**, while the backup server operates on port **12346**. The primary server acts as the main point of contact for clients, handling all incoming message send and receive requests. In contrast, the backup server plays a passive but critical role, maintaining a replicated log of messages received by the primary to provide fault tolerance.

Clients interface exclusively with the primary server, connecting through a dedicated client interface component designed to manage message transmission and reception in a user-friendly manner. This client interface abstracts the underlying network communication using TCP sockets and provides commands such as SEND to dispatch messages and RECEIVE to fetch stored messages, ensuring a straightforward interaction model.

Central to the server's internal operation is the **RaftNode** abstraction, which simulates key elements of the Raft consensus algorithm to represent the server's current role within the system's distributed state machine. Each RaftNode instance is assigned one of three possible states:

- **Follower:** The default state, passively following the leader's instructions.
- **Candidate:** Periodically initiates an election to become leader by requesting votes.
- **Leader:** Coordinates message replication and manages client requests.

While the current implementation does not support dynamic leader election at runtime, the RaftNode class maintains term and leader identification metadata to simulate leader awareness and election logic.

The system defines a **Message** class, a serializable Java object encapsulating critical metadata for each communication instance. This includes:

- Sender ID: Identification of the message originator.
- Recipient ID: The intended recipient within the distributed system.
- Content: The payload or message body.
- Timestamp: Assigned upon message creation to track delivery order.

Replication between the primary and backup servers is accomplished via TCP socket communication, where serialized Message objects are transmitted across the network. Upon receipt, the backup server appends these messages

to its local log, thereby ensuring message redundancy and providing a baseline for fault tolerance. The replication mechanism enforces a write-to-both-nodes approach, requiring successful storage on the backup before acknowledging the client, thus simulating strong consistency within the constraints of the simplified system.

DESIGN AND ARCHITECTURE

The architecture of the distributed fault-tolerant messaging system is designed around modular components that collectively support reliable message delivery, replication, and simplified consensus simulation. Each component plays a distinct role in fulfilling the system's objectives of fault tolerance, consistency, and message ordering within a primary-backup framework.

KEY SYSTEM COMPONENTS

Message Class

At the core of the system's data model is the **Message** class, a serializable object that encapsulates essential communication metadata and payload. This class includes:

- **Sender ID:** Specifies the originator of the message, helping identify the source node or client.
- **Recipient ID:** Identifies the intended receiver, potentially enabling routing or selective delivery.
- **Content:** Contains the main message payload, which can be any user-generated text.
- **Timestamp:** Automatically assigned at creation using the system clock, this field records when the message was generated to assist in ordering and consistency.

Messages are serialized for network transmission, enabling efficient delivery and replication across server nodes.

RaftNode Abstraction

The **RaftNode** class models the internal state of each server node, adhering loosely to the Raft consensus roles to simulate leader election and log management. Each RaftNode maintains:

- **Node State:** One of FOLLOWER, CANDIDATE, or LEADER, influencing behavior such as handling client requests or responding to election events.
- **Current Term:** Tracks the logical term or epoch of the node to manage election cycles and leader legitimacy.
- **Leader ID:** Records the identity of the current leader node, allowing clients and replicas to recognize the primary.
- **Logs:** A list data structure holding the sequence of replicated messages to maintain a consistent state between primary and backup.

Despite the Raft-inspired structure, the implementation limits dynamic behavior; leader election is simulated at startup without live transitions, and logs are stored in-memory without persistent backing.

ClientHandler

The **ClientHandler** component manages the lifecycle of client connections and interprets commands issued by clients. Its responsibilities include:

- **SEND:** Accept incoming messages from clients, append them to the primary's log, and forward them to the backup for replication.
- **RECEIVE:** Retrieve messages from the primary's local log to deliver to the client in order.
- **REPLICATE:** Accept replication requests from the primary on the backup server to update logs accordingly.

ClientHandler acts as the crucial interface bridging external clients and the internal server logic, forwarding and synchronizing data while enforcing the primary-backup distinction in command handling.

Server Initialization

The **Server** class is responsible for starting a node instance in either primary or backup mode based on runtime arguments. Key design elements include:

- **Primary Mode:** Listens for client connections on port 12345, processes client commands, manages message logs, and initiates replication.
- **Backup Mode:** Listens on port 12346, passively accepts replication requests but rejects client commands, maintaining a replicated log without serving clients directly.

This delineation ensures clear separation of roles, where the backup server acts as a fault-tolerant replica without interfering in client communication, providing a foundation for redundancy and availability.

RaftElection Simulation

The **RaftElection** module simulates the leader election process typical of Raft consensus but in a simplified manner. While it includes mechanisms such as:

- Randomized vote requests from candidate nodes to followers.
- Election results printing to track leader selection outcomes.

It currently does not enact dynamic state changes in live operation, meaning that after initial setup, leadership remains static without real-time failover or re-election capabilities. This simulation primarily serves an educational purpose to demonstrate consensus concepts rather than provide operational leader management.

PRIMARY-BACKUP REPLICATION MODEL

The defining architectural pattern of the system is the primary-backup replication model, wherein:

- The **primary server** is the authoritative node that interacts directly with clients. It manages the acceptance, ordering, and local logging of messages.
- On processing a client SEND command, the primary appends the message to its log and immediately initiates replication by transmitting the serialized message to the backup server.
- The **backup server** accepts replication messages through the REPLICATE command, updating its local log to mirror the primary's state, but does not serve any client-originating requests.
- A successful replication acknowledgement from the backup is required before the primary returns an OK response to the client, simulating strong consistency via synchronous write acknowledgment.

This design guarantees that the backup maintains a near-identical state to the primary at all times, providing fault tolerance by preserving message logs even if the primary fails. However, the backup's role remains passive, lacking mechanisms to detect primary failures or automatically promote itself, which are areas marked for future enhancement.

INTERACTION BETWEEN COMPONENTS

Together, these components form an integrated architecture where client requests are funneled exclusively through the primary server's **ClientHandler**. Upon receiving a message, the primary's **RaftNode** appends it to the internal log and coordinates replication by marshaling the message to the backup via TCP sockets. The backup, operating under a mirrored RaftNode instance, appends replicated messages to its local log but does not engage with the client directly.

The **RaftElection** logic is invoked primarily at startup to simulate the initial leader designation, influencing the roles and permissions each node assumes. This tight coupling between client communication, log management, and replication under a well-defined primary-backup scheme fulfills the system's goal of demonstrating basic fault tolerance and state consistency within a distributed messaging environment.

MEMBER 1: FAULT TOLERANCE

Fault tolerance is a fundamental objective of this distributed messaging system, aiming to maintain message availability and system reliability even when individual nodes encounter failures. To achieve this, the system employs a redundancy strategy through backup replication. Every message received by the primary server is immediately forwarded and stored on the backup server, ensuring that message data is duplicated across two distinct nodes. This replication mechanism minimizes the risk of data loss in case the primary node fails unexpectedly.

Communication between the primary and backup servers is facilitated via TCP sockets, where messages are serialized Java objects encapsulated within the Message class. The serialization preserves complete message metadata — including sender, recipient, content, and timestamp — allowing the backup to store an exact copy of each message. Upon replicating a message, the system only acknowledges success to the client once the backup confirms it has safely stored the data, thereby enforcing durability guarantees.

Despite this robust replication scheme, the current implementation exhibits several limitations regarding fault tolerance. Notably, the system lacks failure detection capabilities; there are no heartbeat messages or monitoring mechanisms to detect if the primary server has crashed or become unreachable. Additionally, the backup server plays a strictly passive role, solely receiving replicated messages and maintaining an internal log without

the ability to serve client requests or initiate failover. Consequently, if the primary node fails, the system cannot automatically switch to the backup, resulting in potential service downtime until manual intervention occurs.

To address these shortcomings, two primary enhancements are proposed:

- **Heartbeat Messages for Failure Detection:** Implementing periodic heartbeat signals from the primary to the backup can enable timely failure detection. If heartbeats are missed consecutively, the backup server could assume the primary node has failed, triggering failover procedures.
- **Backup Promotion to Primary:** Empowering the backup server to autonomously transition to the primary role upon detecting a primary failure would facilitate automatic recovery. This requires implementing leader election protocols, consistent state management, and client redirection mechanisms to restore uninterrupted service.

Incorporating these enhancements would significantly improve system resilience, enabling it to handle node failures transparently and maintain continuous message availability without manual recovery steps.

MEMBER 2: DATA REPLICATION AND CONSISTENCY

Maintaining synchronization and consistency between the primary and backup servers is vital for the integrity and reliability of the distributed messaging system. The primary goal is to ensure that both servers hold an identical sequence of messages so that the system can provide fault tolerance without compromising data correctness or availability.

PRIMARY-BACKUP REPLICATION APPROACH

The system employs a straightforward primary-backup replication model where all client interactions occur exclusively through the primary server. Upon receiving a message from a client, the primary appends it to its local log and then immediately initiates replication by sending the serialized message to the backup server over a TCP socket.

The backup node accepts replication requests and appends the messages to its own local log, mirroring the primary's state. Only after the backup acknowledges the successful storage of the message does the primary send a confirmation response back to the client. This synchronous acknowledgement

process simulates strong consistency by ensuring that a message is considered committed only when it is durably stored on both nodes.

SIMULATED STRONG CONSISTENCY

Although the implementation does not realize a full consensus protocol, the replication approach approximates strong consistency semantics. The client perceives a message as "sent" only after both primary and backup maintain identical logs, thereby minimizing the risk of message loss due to node failure.

This model assumes that the backup log state is always consistent and up-to-date, enabling durability guarantees within the system's limitations. However, this strategy relies on synchronous communication and blocking acknowledgments, which may impact performance under high load or network latency.

TRADE-OFFS

The major benefit of this replication scheme is its simplicity and ease of implementation. Since the backup does not participate in client interactions or leader election at runtime, the system avoids complex coordination and concurrency challenges inherent in fully distributed consensus algorithms.

However, this simplicity comes at the expense of stronger consistency guarantees. The system lacks quorum-based decision-making or multi-node voting mechanisms, which means it cannot handle partial failures or network partitions robustly. Moreover, without log indexing or deduplication, replayed replication messages could cause inconsistencies or wasted storage.

PROPOSED ENHANCEMENTS

- **Quorum-Based Replication:** Introducing a quorum protocol would enable the system to tolerate additional faults by requiring agreement from a majority of replicas before committing messages, thus strengthening consistency and availability guarantees.
- **Deduplication and Log Indexing:** Implementing mechanisms to detect and discard duplicate replication requests would prevent replay errors and optimize log storage efficiency, especially in scenarios involving retransmissions or network delays.

These enhancements would significantly improve the robustness and scalability of the replication mechanism, moving the system closer to production-grade distributed fault-tolerant messaging infrastructures.

MEMBER 3: TIME SYNCHRONIZATION

Accurate time synchronization plays a critical role in distributed messaging systems, primarily to ensure reliable timestamping and proper ordering of messages across nodes. When messages originate from multiple servers, consistent and comparable timestamps allow the system to maintain a coherent global order, which is essential for consistency, debugging, and event tracing. Without synchronized clocks, messages may appear out of order, leading to ambiguity in delivery semantics and potential application-level errors.

CURRENT IMPLEMENTATION

In the implemented system, message timestamps are assigned using Java's standard `new Date()` at the time each message is created by the client or server. This timestamp is embedded within the `Message` class and accompanies the message during replication between the primary and backup servers. However, the current design does not incorporate any mechanism to synchronize system clocks between the two servers.

Consequently, the timestamps reflect each node's local system time, which can lead to discrepancies if the clocks are not aligned. For example, if the primary's clock is ahead of the backup's clock by several seconds, replicated messages might be recorded on the backup with earlier timestamps or vice versa. Such inconsistency can complicate log analysis and ordering logic, potentially causing confusion when determining the sequence of events or the causal relationships between messages.

LIMITATIONS

- **Lack of Clock Synchronization:** Without synchronization protocols, each server's clock drifts independently, making cross-node comparison of timestamps unreliable.
- **Incorrect Message Ordering:** Timestamps may not accurately represent the real-time order of message generation and replication, leading to out-of-order logs.

- **No Logical Time Tracking:** The system currently does not track causality or event dependencies, relying solely on wall-clock time, which cannot capture concurrent events properly.

PROPOSED ENHANCEMENTS

To improve time consistency and message ordering, several advanced synchronization strategies can be adopted:

- **NTP-Based Clock Synchronization:** Implementing Network Time Protocol (NTP) synchronization on all nodes ensures system clocks are periodically adjusted to reflect standardized Coordinated Universal Time (UTC), minimizing drift and skew across servers.
- **Logical Clocks (Lamport Clocks):** Using Lamport timestamps assigns integer counters that increment upon events and message transfers, providing a partial ordering of events that respects causality without relying on physical clocks.
- **Hybrid Logical Clocks:** A combination of physical time and logical counters can offer better fault-tolerant ordering by embedding causality information along with real-world timestamps, allowing systems to maintain monotonically increasing timestamps even during clock skew or adjustments.

Incorporating these enhancements will strengthen the system's ability to maintain consistent event ordering, simplify debugging and recovery, and lay the groundwork for more advanced consensus and synchronization mechanisms required in robust distributed systems.

MEMBER 4: CONSENSUS AND AGREEMENT ALGORITHMS

Achieving consensus is a cornerstone of distributed systems that need to maintain consistent state across multiple nodes despite failures, delays, or network partitions. In this messaging system, the primary objective concerning consensus is to simulate key elements of leader-based agreement protocols to ensure consistency and coordination between the primary and backup servers. The system adopts principles inspired by the Raft consensus algorithm, which emphasizes an understandable and robust approach to leader election and log replication.

OBJECTIVES

The consensus mechanism aims to guarantee that only one server—the leader—controls the order and commitment of messages, thereby preventing conflicting logs and ensuring strong consistency. The leader election process establishes which node acts as the primary authority, while agreement algorithms coordinate replication of client requests to backups. This coordination is crucial to avoid inconsistencies and to provide a foundation for fault tolerance.

RAFTNODE CLASS IMPLEMENTATION

The **RaftNode** abstraction encapsulates the state and behavior of a distributed server node according to Raft-like roles. Each node maintains three core pieces of metadata:

- **Current Term:** A monotonically increasing integer representing the current election term or epoch. Terms help distinguish outdated leaders and coordinate elections.
- **Leader ID:** The identity of the node currently recognized as the leader. This allows both clients and backup nodes to identify the primary node responsible for processing requests.
- **Node State:** The operational role of the node within Raft's protocol states—FOLLOWER, CANDIDATE, or LEADER. These states dictate the node's behavior, such as whether it votes, initiates elections, or serves client requests.

The RaftNode also manages an in-memory log of messages appended during operation, which it coordinates in replication with the backup. However, persistent storage of logs is not currently implemented.

RAFTELECTION SIMULATION

The **RaftElection** component simulates the election process by which a candidate node requests votes from other nodes in order to become leader. This simulation includes:

- A randomized voting procedure mimicking Raft's election timer randomness, aimed at minimizing election conflicts.
- Printing election results to console logs to visualize leader selection within the system.

Despite this, the election simulation does not enact any dynamic or real-time state changes. Once the initial role is assigned, the leadership remains static during runtime, meaning no actual re-election or failover is triggered by node failures in the current system.

IMPLEMENTED FEATURES

- **Simulated Leader Election:** Candidate nodes issue vote requests and receive votes from peers in a controlled manner, demonstrating the logic of leader selection.
- **Leader Awareness:** Both client components and server nodes recognize the current leader using the leaderId field in RaftNode, facilitating consistent client-server interactions with the primary.
- **Role-Based Behavior:** Nodes conditionally accept client requests or replication commands based on their current state, enforcing primary-backup roles aligned with Raft principles.

UNIMPLEMENTED AREAS AND LIMITATIONS

While the system embraces foundational Raft concepts, several critical consensus features remain unimplemented:

- **Dynamic Leader Promotion and Demotion:** The system lacks mechanisms for runtime leader re-election or failover, meaning the backup cannot become leader automatically if the primary fails.
- **Persistent State Management:** All logs, terms, and election votes are stored in memory only, risking data loss upon node restarts and limiting durability guarantees.
- **Full Consensus Log Agreement:** The implementation does not include quorum-based commits or replication acknowledgments from a majority to confirm message commitment, as in full Raft.
- **Heartbeat Messages:** The regular leader-to-follower heartbeat that maintains leadership and detects failures is absent.

FUTURE WORK

Extending this system to fully implement Raft consensus would involve multiple enhancements to robustly manage distributed agreement:

- **Election Timeouts and Heartbeats:** Introducing timing mechanisms for elections and periodic heartbeat messages would enable dynamic leadership changes and failure detection.

- **Persistent Logs and State:** Storing logs and election state on stable storage prevents data loss and enables recovery after crashes.
- **Quorum-Based Commit Protocols:** Logging and committing messages only after acknowledgment by a majority of nodes would guarantee stronger consistency and fault tolerance.
- **Handling Network Partitions and Split-Brain Scenarios:** Implementing proper leader election and log reconciliation to avoid conflicting leaders during partitions and support node rejoining.
- **Leader Promotion:** Enabling backups to autonomously transition to leaders upon primary failure, facilitating seamless failover and continuous availability.

These improvements would enhance the system from a simulation to a fully operational consensus-based distributed messaging platform capable of handling complex faults and providing robust availability guarantees.

CONCLUSION

This distributed fault-tolerant messaging system successfully demonstrates core principles of fault tolerance and distributed consensus through a simplified primary-backup architecture with simulated Raft consensus elements. By replicating messages synchronously from a primary to a backup server, the system ensures data durability and availability despite potential node failures, illustrating fundamental fault tolerance mechanisms. The inclusion of a Raft-inspired election simulation and leader identification introduces basic consensus concepts, serving as an educational stepping stone for understanding leader-based coordination in distributed environments.

While the current implementation effectively establishes a foundational messaging framework, it also highlights critical areas for improvement to achieve a robust production system. Notably, the lack of persistent log storage, absence of heartbeat and failure detection protocols, and the inability for dynamic failover limit the system's operational resilience and fault recovery capabilities. Implementing these enhancements, alongside a full Raft protocol integration with persistent logs, real-time leader election, and quorum-based replication, will greatly enhance consistency, availability, and fault tolerance.

Overall, this project lays a solid groundwork for future development by combining simplified replication with consensus simulation, offering a practical and extensible platform for exploring distributed fault-tolerant

messaging system designs and advancing toward production-ready implementations.