Sri Lanka Institute of Information Technology (SLIIT)
Faculty of Computing

Distributed Systems - SE2062

Assignment

Submitted By:

- YATAWARA Y W U M K - IT23289598
- KATUGAMPALA K K V T - IT23425590
- UDARA S M A - IT23372726
- SATHURUSINGHE S A K S - IT23161160

Contents

1. Fault Tolerance
2. Data Replication and Consistency
3. Time Synchronization
4. Consensus and Agreement Algorithms

1. Fault Tolerance

In any distributed system, fault tolerance is critical to ensure the system remains operational, even in the event of server failures. Our distributed logging system is designed to withstand server failures and continue functioning smoothly by utilizing mechanisms like log redundancy, server failover, and recovery. This section outlines the approach we've implemented for fault tolerance.

1.1 Redundancy and Log Replication

Our system employs a log redundancy mechanism, where logs are replicated across multiple servers. This ensures that even if one server fails, the log data is preserved and can be accessed from the backup servers.

To achieve this, the logs are written to the primary server, and once the data is committed, the logs are asynchronously replicated to two other backup servers. This replication mechanism ensures that data is not lost, even if a server goes down before the data can be replicated.

1.2 Failover Mechanism

In the event of a primary server failure, we have implemented an automatic failover mechanism. The system detects server failures and automatically switches to a backup server, promoting it to the role of the primary server. This ensures that the system remains available for new log entries without significant downtime.

Once the primary server is restored, the system seamlessly reverts the role to the original primary, with the backup server reverting to its standby state. This failover process is crucial to maintain the availability and reliability of the system during server outages.

1.3 Server Failure Detection

Our system continuously monitors the health of the servers and can quickly detect when a server fails. This is done through periodic health checks and heartbeat mechanisms between the servers. When a failure is detected, the failover process is triggered automatically.

1.4 Log Recovery Upon Server Restoration

When a server that was previously down comes back online, it must recover the logs that were written during its downtime. This is achieved by having the server pull the missing logs from the other servers that were up and running during the downtime. This ensures that the server is in sync with the rest of the system, and no log entries are lost.

1.5 Impact on Performance and Storage

While redundancy and replication enhance the fault tolerance of the system, they also introduce certain trade-offs. The primary impact is on storage, as multiple copies of logs are maintained across the servers. This requires additional disk space to store these logs, increasing the overall storage requirements of the system.

In terms of performance, the asynchronous replication process introduces a slight delay in the time it takes for the logs to be fully replicated across all servers. However, this delay is generally minimal and does not significantly affect the overall system performance, as the failover mechanism ensures that write operations continue without interruption.

The system's ability to recover logs quickly upon server restoration also adds a layer of fault tolerance but may slightly increase recovery time if the server has been down for an extended period.

1.6 Conclusion

The fault tolerance mechanisms in place—log redundancy, automatic failover, and log recovery—ensure that the distributed logging system remains available and operational, even in the event of server failures. By using these strategies, the system achieves high availability and reliability, critical for maintaining a robust distributed logging system.

2. Data Replication and Consistency

2.1 Introduction

Ensuring availability of data in a distributed system is crucial. In data replication, data is maintained across multiple servers by creating copies of the data to ensure availability and to make sure that the system does not lose the data during a failure. This section covers the data replication and consistency mechanisms and strategies used in our distributed logging system; by analyzing replication strategy, consistency model and deduplication mechanism used in this system.

2.2 Replication Strategy

Our distributed logging system uses a primary-backup replication strategy where the primary server accepts data and saves it in its local database, and after the data is committed locally, the data is replicated to two other backup servers.

Pros of Primary-Backup Replication Strategy:

- Implementation is simple.
- Improves durability by storing data in multiple locations.

Trade-offs of Primary-Backup Replication Strategy:

- More storage is needed across all servers.
- If the primary node fails before replication, it may cause data inconsistency.

2.3 Consistency Model

This distributed logging system uses an eventual consistency model, which means when a log is committed to the primary server, the replication of the log is done asynchronously (backups are not instant, will eventually be consistent)

Pros of Eventual Consistency Model:

- Higher availability
- Avoids blocking operations

Trade-offs of Eventual Consistency Model:

- Some logs may not be replicated in rare failure cases

2.4 Deduplication Mechanism

A deduplication mechanism is proposed for this distributed logging system, by using a unique identifier for each log. When a server receives a backup log, the database is checked in search of any identical UUID or a hash, and if there is an identical log, the log is discarded to prevent duplication.

2.5 Replication Affecting Latency

Write latency is increased due **to the time** consumed during replicating logs to the servers. User-facing latency remains low due to replication being asynchronous. However, Read latency is not affected as reads are served only from the primary node.

2.6 Replication Affecting Storage Efficiency

Replication causes more storage consumption as every log is stored on the primary server and all other backup servers. If duplicate logs are stored that would cause a space wastage if no deduplication or compression is done.

3. Time Synchronization

3.1. Introduction

In a distributed logging system, maintaining accurate and consistent timestamps across multiple nodes is crucial for event tracing, debugging, and system monitoring. However, distributed environments suffer from clock drift, inconsistent network delays, and system load variations, all of which can result in timestamp discrepancies. This document outlines the design and implementation of a time synchronization strategy that ensures timestamp integrity, fault tolerance, and minimal performance overhead in a distributed logging architecture.

3.2. Time Synchronization Protocol

We use the Network Time Protocol (NTP) as the foundational synchronization mechanism for all logging nodes.

3.2.1 Why NTP?

- **Wide support and reliability**: NTP is supported across platforms and has decades of operational stability.
- **Reasonable precision**: Provides millisecond-level synchronization over the internet, which is adequate for logging systems.
- **Low overhead**: Minimal CPU and bandwidth usage, even with frequent polling.

3.2.2 NTP Configuration and Operation

- All nodes use NTP clients (ntpd or chronyd) to sync with public servers like pool.ntp.org.
- Synchronization occurs at regular intervals (e.g., every 5 minutes) to correct clock drift.
- Example configuration in /etc/ntp.conf:

  server 0.pool.ntp.org iburst

  server 1.pool.ntp.org iburst

  server 2.pool.ntp.org iburst
- Additionally, the system uses runtime NTP queries (via ntplib in Python) to fetch current synchronized time for log timestamping.

3.3. Clock Skew Analysis

Clock skew arises when different machines' clocks deviate slightly. Even differences of 10–20 milliseconds can lead to serious issues in distributed systems.

3.3.1 Effects of Clock Skew

- **Incorrect log ordering**: Events may appear out of sequence.
- **Misleading diagnostics**: Makes it harder to correlate events across nodes.
- **Data inconsistency**: Logical operations may fail if based on timestamps.

3.3.2 Skew Mitigation

- sync_time() logs both system time and NTP time to observe skew.
- Timestamp correction is applied using known offsets if clock drift is detected.

3.4. Log Reordering Mechanism

Despite synchronization efforts, logs may still arrive out of order due to network or processing delays. We implemented a log buffering mechanism to ensure correct event ordering.

3.4.1 Reordering Strategy

- Logs are placed into a priority queue (heapq) upon arrival.
- A short buffer delay (1 second) is introduced to allow late-arriving logs to be reordered.

3.4.2 Buffer Flushing Algorithm

```
import heapq
import asyncio

class TimeSyncService:
    def __init__(self):
        self.log_buffer = []
        self.lock = asyncio.Lock()

    async def receive_log(self, log_entry):
        async with self.lock:
            heapq.heappush(self.log_buffer, (log_entry['timestamp'], log_entry))

    async def flush_logs(self, db):
        async with self.lock:
            while self.log_buffer:
                _, log_entry = heapq.heappop(self.log_buffer)
                await store_log(log_entry, db)

    async def delayed_flush(self, db, delay=1):
```

```
        await asyncio.sleep(delay)
        await self.flush_logs(db)
```

3.5. Timestamp Correction Techniques

In scenarios where clocks are temporarily unsynchronized (e.g., before NTP sync), the system applies runtime corrections.

3.5.1 Offset Tracking

- Each node logs its NTP offset during sync.
- This offset is cached and used to adjust log timestamps before storage.

3.5.2 Example

- If node A is 15ms behind, then log.timestamp += 0.015s before storing.
- Helps align logs without requiring perfect real-time synchronization.

3.6. Trade-offs and Performance Evaluation

Feature	Advantage	Disadvantage
Frequent NTP Sync	High accuracy	Small network/CPU load
Buffered Reordering	Eliminates minor disorder	Adds ~1s delay
Timestamp Correction	Better accuracy post-skew	May apply wrong offset

Key Considerations

- **Sync interval**: Every 5 minutes strikes a balance between accuracy and load.
- **Flush delay**: 1-second buffer is optimal to catch delayed logs without noticeable latency.
- **Scalability**: Works well with many logging nodes and asynchronous pipelines.

Time synchronization is essential for a fault-tolerant distributed logging system. By integrating NTP-based synchronization, buffered log reordering, and timestamp correction, the system ensures

consistent and trustworthy logs. These mechanisms balance precision and performance, enabling reliable event tracking, debugging, and scalability in real-world distributed environments.

4. Consensus and Agreement Algorithms

In this implementation of a distributed logging system using FastAPI and Python, consensus and strong consistency across multiple log nodes are achieved through a custom-built adaptation of the Raft consensus algorithm. The Raft protocol ensures that replicated logs remain consistent even in the presence of node failures by organizing nodes into three distinct roles: follower, candidate, and leader.

Leader election is triggered when a follower detects the absence of heartbeat signals from the current leader within a randomized election timeout, typically between 1000 and 2000 milliseconds. This randomization helps avoid conflicts caused by simultaneous elections, reducing the chances of vote splits. When this timeout elapses, the follower becomes a candidate, increases its term, and sends out RequestVote RPCs to other nodes to initiate an election.

Once elected, the leader begins broadcasting regular heartbeat signals—empty AppendEntries RPCs—every 150 milliseconds to maintain authority and suppress new election cycles. These heartbeats play a critical role in ensuring the system remains stable and quickly responsive to leader failures.

To further enhance stability and avoid simultaneous candidacy after a heartbeat lapse, the system introduces a randomized delay (jitter) between 100–300 milliseconds before any node initiates an election. This simple mechanism greatly reduces the risk of vote splitting and accelerates consensus.

When it comes to log replication, the leader node handles client requests by appending the submitted log entries to its local log. It then propagates these entries to follower nodes using

AppendEntries RPCs, which include both the new log data and metadata such as the previous log index and term. Followers ensure consistency by checking this metadata before accepting the entries. A log entry is considered committed once it has been acknowledged by a majority of nodes, ensuring consensus is reached.

This approach guarantees that all nodes maintain a consistent and ordered log, even in the face of network

4. Conclusion

In building a distributed logging system, we addressed the core challenges of fault tolerance, data replication, time synchronization, and consensus through a carefully integrated architecture combining industry-standard techniques and custom implementations. Our system is designed to operate reliably under failure conditions while maintaining data consistency, ordering, and availability.

- **Fault Tolerance** was achieved via log redundancy, automated failover, and recovery mechanisms to ensure uninterrupted logging services despite server outages.
- **Data Replication and Consistency** were implemented using a primary-backup strategy with asynchronous replication and deduplication techniques, balancing performance and durability.
- **Time Synchronization** was enforced using the Network Time Protocol (NTP), buffered log reordering, and timestamp correction to guarantee temporal accuracy and ordering in a multi-node environment.
- **Consensus Algorithms**, based on an adapted Raft protocol, ensured that all nodes maintain a consistent state through reliable leader election and log agreement processes.

Together, these components form a robust, scalable, and fault-tolerant distributed logging solution. It can handle real-world demands for reliability, efficiency, and accuracy across distributed systems. This implementation not only fulfills the assignment objectives but also serves as a strong foundation for further enhancements such as security, real-time analytics, and cross-cluster replication in production-grade environments.