Distributed Systems COMP SCI 3012 Assignment 4

Gia Bao Hoang - a1814824

Other Group Members:

Marcus Hoang - a1814303

Ran Qi - a1675122

1. Introduction:

1.1. Overview of Assignment 2's Goals and Objectives:

Assignment 2 focuses on the creation and management of a distributed system to handle weather data. It emphasizes the importance of ensuring consistency, fault tolerance, and scalability. A crucial feature in this assignment is the usage of Lamport clocks, which are essential for maintaining partial order among events in the distributed system, thus ensuring synchronization and sequence integrity.

1.2. Short Descriptions of Each Team Member's Implementation:

My Implementation:

- A LoadBalancer acts as the primary entry point, handling requests from both GETClient and ContentServer.
- Requests are distributed to AggregationServers in a round robin fashion by the LoadBalancer, which also monitors the health of the AggregationServers.
- The AggregationServers, governed by the LoadBalancer, employ a shared DataStoreService to store incoming weather data from PUT requests. This service takes care of data integrity by managing stale data and backing up information periodically. It also reloads backup data during startup if available.
- Each AggregationServer is a distinct entity and handles client requests in a queued sequence.
- Every AggregationServer maintains both a local Lamport clock and has access to a shared global Lamport clock. The local clock gets updated for various events such as receiving requests, sending responses, and synchronizing with the global Lamport clock.
- When processing initial client requests, an AggregationServer shares its Lamport clock with the client, facilitating client side partial order synchronization.

Marcus's Implementation:

- A single AggregationServer processes client requests sequentially using a queue. This ensures requests are handled in the order they are received.
- Periodic data integrity checks are performed, and backups are created for data, with the option to load from these backups during system startup.
- Only one AggregationServer exists, which maintains a singular Lamport clock to establish and manage partial order among events.

Ran's Implementation:

- The system has a single point of entry, an AggregationServer, which manages a weather data database. This server initializes by attempting to load data from a backup file.
- Every client request spawns a new thread for processing, ensuring scalability and responsiveness. Operations on the shared database and Lamport clock are synchronized, preserving data integrity.
- Client requests are managed by the ClientHandler, which operates as an independent thread.
- The ContentServer in this implementation has a unique feature. Before sending new weather updates, it queries the server for the current Lamport clock value to synchronize its local clock.

Throughout the three implementations, a consistent theme is the importance of synchronization, especially with Lamport clocks, to maintain order and ensure smooth operation in the distributed system. Error handling and failover strategies, like retrying connections and loading from backups, are essential for resilience in the face of potential issues or server crashes.

2. Evaluation Plan:

To ensure a thorough assessment of the distributed systems, our evaluation will be grounded on a structured approach, emphasizing core functionalities, resilience, scalability, and system synchronization using Lamport clocks.

2.1. Evaluation Criteria:

The criteria for our evaluation are segmented into three principal domains:

- **Failure Handling:** Evaluating the system's adeptness in managing unforeseen errors, handling crashes, and maintaining seamless operations without compromising data integrity or causing significant service interruptions.
- **Scalability:** Assessing the system's capacity to accommodate growth, be it in the form of escalated data volumes, amplified user requests, or the addition of more content servers, all while preserving performance standards and reliability.
- **Usage of Lamport Clocks:** A focused review on the integration, correct application, and operational efficiency of Lamport clocks, ensuring accurate event sequencing in the distributed system environment.

2.2. Methodological Framework:

The evaluation will employ a two pronged methodology:

2.2.1. Experimental Analysis:

Standard & Adverse Conditions:

- Conduct a series of tests to simulate both everyday use and potential edge cases.
- Analyze the system's resilience by simulating occasional or multiple server failures.
- Stress Test with a surge in read client requests to gauge response times and system robustness.
- Introduce server downtime episodes to scrutinize the system's retry and recovery functionalities.

Operational Scenarios:

- Basic Operations: Confirm the efficacy of fundamental operations such as text messaging, fluid client server communications, single PUT actions, simultaneous GET requests, and the periodicity and efficiency of data clearance after 30 seconds.
- Advanced Operations: Investigate the accurate implementation of Lamport clocks, the comprehensiveness of error codes, and the resilience and fault tolerance of content servers.

2.2.2. Code Review & Inspection:

Review Techniques:

- Employ manual inspection strategies to validate the correct application of algorithms, particularly those related to Lamport clocks and error remediation.

- Engage static code analysis utilities to unearth potential anomalies, inconsistencies, or security vulnerabilities.
- Review embedded code documentation and annotations to ascertain clarity, correctness, and comprehensiveness.
- Reconcile documented features and actual code implementations to pinpoint discrepancies or omissions.

2.3. Investigative Queries:

To hone our evaluation and ensure it's aligned with our primary criteria, we've articulated specific investigative questions. These questions serve to probe essential areas, ensuring we garner a comprehensive understanding:

1. **Failure Handling:**
   - What are the immediate system reactions to abrupt content server failures?
   - What builtin mechanisms are in place to manage sporadic server unavailability?
   - How does the system manage operation retries, and what is the logic behind retry thresholds?

2. **Scalability:**
   - Under augmented operational loads, how does the system performance graph change?
   - Are there discernible performance bottlenecks when scaling operations?
   - How does the system navigate the challenge of data consistency during peak loads?

3. **Lamport Clocks Integration:**
   - What's the structural design for the integration of Lamport clocks, and how do they function in real time operations?
   - In which scenarios could Lamport clocks potentially malfunction or result in system inconsistencies?
   - How does the system orchestrate clock synchronization across various nodes, especially amidst operational lags or synchronicities?

4. **Code Quality & Structural Integrity:**
   - Does the code adhere to recognized best practices and modern design paradigms?
   - Is the code comprehensively documented, and does this documentation resonate with the actual implementation?
   - Are there evident mismatches between the stipulated features and the code's actual operational capabilities?

Our evaluation methodology, rooted in empirical analysis, code audits, and a set of well defined investigative questions, is designed to offer a comprehensive, in depth assessment of the distributed systems, underlining their strengths, potential enhancement areas, and ensuring they meet the essential criteria laid out.

3. Characteristics and Tradeoffs:

To provide a holistic assessment of the distributed system implementations, our examination is rooted in the structured approach delineated in our Evaluation Plan. The outcomes presented are sourced from both empirical experimental analyses and in-depth code reviews, centered around failure handling, scalability, and Lamport Clocks integration.

An Overview Before the Deep Dive:
Before embarking on a detailed discussion of the evaluations, it's pivotal to encapsulate the essence of each implementation. These summaries provide a snapshot, laying the groundwork for the ensuing in-depth analyses. They highlight the primary design philosophies and the trade-offs inherent to each approach.

My Implementation:
While the decentralized design inherently offers greater resilience against specific server failures, it simultaneously poses challenges. For instance, the introduction of a centralized data service, aimed at promoting data consistency, may indeed streamline operations. However, this centralization could also become a single point of potential scalability bottleneck, thereby reducing the overall system agility in rapidly changing scenarios.

Marcus's Implementation:
Marcus's model seems to emphasize simplicity and elegance, predominantly by limiting its scope to a singular server and sequential processing. However, this simplicity may sometimes be a double-edged sword. While it ensures reduced system components (potentially reducing system inconsistencies), it also means that the entire system's efficiency and operability rely on that single server. This brings about scalability concerns, especially when faced with sudden, large spikes in client requests.

Ran's Implementation:
Ran's approach is emblematic of the modern turn towards concurrency with its emphasis on multithreading. While this indeed allows for handling multiple client requests simultaneously (and thus proving to be more scalable than purely sequential approaches), it also thrusts forward synchronization challenges. Multithreading often necessitates a stringent synchronization mechanism, which, if not handled effectively, can introduce latencies and even potential data inconsistencies.

3.1. Failure Handling:

3.1.1. Findings from Experimental Analysis:

- **My Implementation:** Throughout test simulations, my system demonstrated resilience, notably with its multiple AggregationServers. When a server failure was artificially introduced, the LoadBalancer promptly reacted, diverting client requests. The standalone DataStoreService ensured data remained intact, managing backup and recovery processes effectively.
- **Marcus's Implementation:** Marcus's design, based around a singular AggregationServer, showed commendable error recovery during lighter operational conditions. However, in more adverse scenarios, the recovery time slightly lengthened.
- **Ran's Implementation:** While based on a single server, Ran's system handled sporadic server failures with grace, largely due to its multithreaded nature. Multiple server failure tests revealed occasional inconsistencies in recovery.

3.1.2. Findings from Code Review:

All implementations demonstrated well-documented error handling. While my design presented a layered approach to error handling, it was a tad more intricate yet robust. Marcus's code stood out for its modularity, ensuring enhanced maintainability. Both Marcus and Ran implemented clear retry mechanisms.

3.2. Scalability:

3.2.1. Findings from Experimental Analysis:

- **My Implementation:** Stress tests unveiled the LoadBalancer's capability in evenly distributing client requests among AggregationServers. The standalone DataStoreService was pivotal in retaining brisk responses during heavy traffic.
- **Marcus's Implementation:** Marcus's system operated impeccably under normal conditions. However, when faced with high traffic, a slight increase in response time was noted.
- **Ran's Implementation:** Due to its multithreading, Ran's system ensured superior concurrent request management. However, peak loads did introduce minor latency.

3.2.2. Findings from Code Review:

Code for my implementation suggested provisions for potential future expansion, emphasizing component modularity. Marcus's design, while singular, was exemplary in terms of modularity, making it conducive for maintenance and potential module-based enhancements. Ran's design, though singular, incorporated elements for easier vertical scalability.

3.3. Lamport Clocks Integration

3.3.1. Findings from Experimental Analysis:

- **My Implementation:** The dual-clock system exhibited impeccable event sequencing. Event synchronization remained consistently accurate, even under heavy operational loads.
- **Marcus's Implementation:** The singular Lamport clock in Marcus's design maintained accurate partial ordering during standard operational scenarios. However, under considerable operational stress, minor synchronization delays were observed.
- **Ran's Implementation:** With its singular clock, Ran's design maintained impressive sequencing accuracy due to its synchronized operations. Adverse conditions introduced occasional synchronization overheads.

3.3.2. Findings from Code Review:

All implementations manifested effective Lamport Clocks integration. My dual-clock code was slightly intricate but thoroughly documented. Marcus's clock synchronization code was straightforward, with added emphasis on module clarity, enhancing its maintainability. Ran ensured clock synchronization clarity in his code, with additional elements for multithreading management.

3.4. Discussion: Pros & Cons for Each Implementation

Drawing from Table 1, a few discernible patterns emerge. Scalability, an ever-present concern in distributed systems, shows varying degrees of robustness across the three implementations. Similarly, while each system has its unique ways of promoting data integrity and handling failures, they all underscore the critical need for a holistic approach to design.

The architectural choices in my implementation are emblematic of a trend towards decentralization, promoting robustness but also introducing complexities. Marcus's inclination for a streamlined model echoes the sentiments of those who advocate for simplicity in design, but it also brings to the fore the risks associated with over-reliance on a single component. Ran's system, meanwhile, is a nod to the rising importance of concurrency in modern computing but also shines a spotlight on synchronization's inherent challenges.

Conclusively, these examinations not only provide specific insights into the characteristics of each implementation but also resonate with broader trends and challenges in distributed system design. As with most engineering endeavors, it's about balancing trade-offs to align with the specific requirements and constraints of a given project.

Criteria/ Implementation	My Implementation	Marcus's Implementation	Ran's Implementation
Pros			
1. Unique feature	**Decentralized Design**	**Simplicity**	**Multithreading**
Description	Enhanced resilience against individual server failures.	Singular server & sequential processing simplify maintenance.	Concurrent processing provides more scalability than sequential methods.
2. System Efficiency	**Load Balancing**	**Data Integrity**	**Synchronized Operations**
Description	Efficient distribution of client requests. Supports scalability.	Fewer components, leading to diminished data inconsistency risks.	Ensures data and clock consistency in a multithreaded environment.
Cons			
1. System Challenge	**Complexity**	**Scalability**	**Synchronization Overheads**
Description	Introducing multiple components can escalate system complexity.	May struggle with large spikes in client requests.	Can introduce latencies during high demand periods.
2. System Risk	**Centralized Data Service**	**Single Point of Failure**	**Single Point of Failure**
Description	Promotes data consistency but can be a potential scalability bottleneck.	Failure in the server can halt the entire system.	Any server malfunction affects the whole system, similar to Marcus's design.

Table 1: Comparative Analysis of Implementation Features and Challenges

4. Conclusion

In Assignment 2, a meticulous exploration of distributed system designs was undertaken, drawing lessons and conclusions from each respective implementation strategy.

4.1. On the Front of Failure Handling:

- **My Implementation:** Opting for a decentralized structure facilitated resilience against individual server failures, but not without raising complexities related to data consistency.
- **Marcus's Design:** The centralization, though straightforward in maintenance and error handling, prominently spotlighted the risks linked to having a singular server.
- **Ran's System:** The use of multithreading accelerated recovery from failures, yet the approach also underscored the essentialness of impeccable synchronization.

4.2. Addressing Scalability:

Each implementation presented a distinct viewpoint on scalability. My model tapped into the potential of load balancing, Marcus's structure posed questions regarding its upward limit, and Ran's approach, while promising in scalability, demonstrated the intricacies of synchronization.

4.3. The Role of Lamport Clocks:

The exercise reconfirmed the significance of Lamport Clocks in a distributed setting. Their utility in ensuring event order and synchronization was evident, with particular emphasis in Ran's design.

4.4. Group Learnings:

- **Marcus's Structure:** Its simplicity serves as a reminder of the importance of preparing for scalability in distributed frameworks.
- **Ran's Strategy:** Offers a deep dive into the interplay of synchronization and concurrent operations.
- **Personal Observations:** The assignment solidified the understanding that design choices, like opting for scalability and resilience, require thoughtful consideration, especially when factoring in robust failure handling.

To wrap up, the insights derived from Assignment 2 offer a comprehensive view into the intricacies of distributed system designs. Emphasizing the intersections of failure management, scalability techniques, and synchronization via Lamport Clocks, it forms a primer for subsequent engagements in distributed systems research.