Replication

REPOSITORY LINK: https://github.com/acheng257/cs2620

We started by reviewing the design exercise description and understanding what it required and the extra steps needed to implement it according to the requirements.

Below is a list of the requirements, extra credit, and ideas that would be fun to implement. We already accomplished the persistent storage per server as we included database creation for the server in our first implementation.

Core Requirements:

- ☑ ~~**Persistent Storage per Server: (THIS WAS ALREADY DONE BEFORE)**~~
  - ☑ ~~Each server must have its storage instance to ensure message persistence.~~
- ☑ ~~**Replication for Fault Tolerance:**~~
  - ☑ ~~Implement a replication mechanism (e.g., leader-follower or consensus protocol) to store messages across replicas consistently.~~
- ☑ ~~**2-Fault Tolerance:**~~
  - ☑ ~~Ensure the system can tolerate up to two server failures without data loss or disruption.~~
- ☑ ~~**Multi-Machine Deployment:**~~
  - ☑ ~~Demonstrate that the replication works across at least two different machines (or hosts/containers).~~
- ☑ ~~**Demo and Documentation:**~~
  - ☑ ~~Provide a demo (Demo Day III) showing persistence and replication features.~~
  - ☑ ~~Include an engineering notebook detailing design choices, implementation steps, and testing.~~
- ☑ ~~**Test Code and Documentation:**~~
  - ☑ ~~Include comprehensive test code and documentation to support your implementation.~~

Extra Credit:

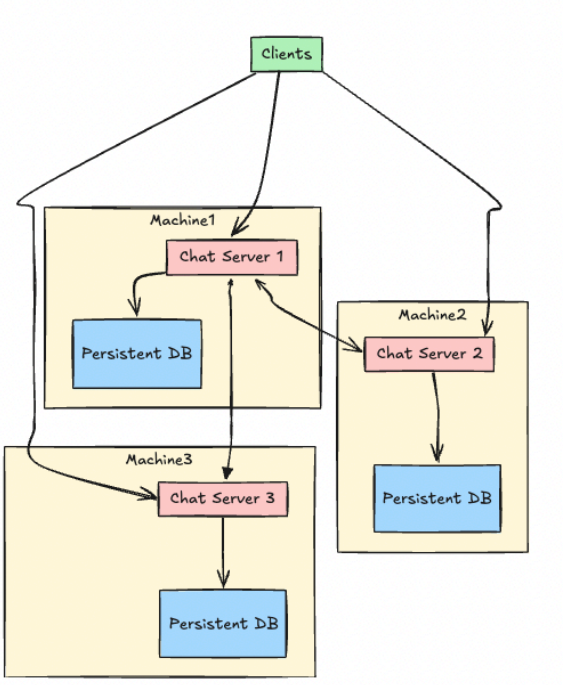
- **Dynamic Replica Addition:**
  - Implement functionality to add new servers to the replica set dynamically.
    - We need to make sure we can replicate the DB and insert it into the replication mechanism.

Extra Ideas:

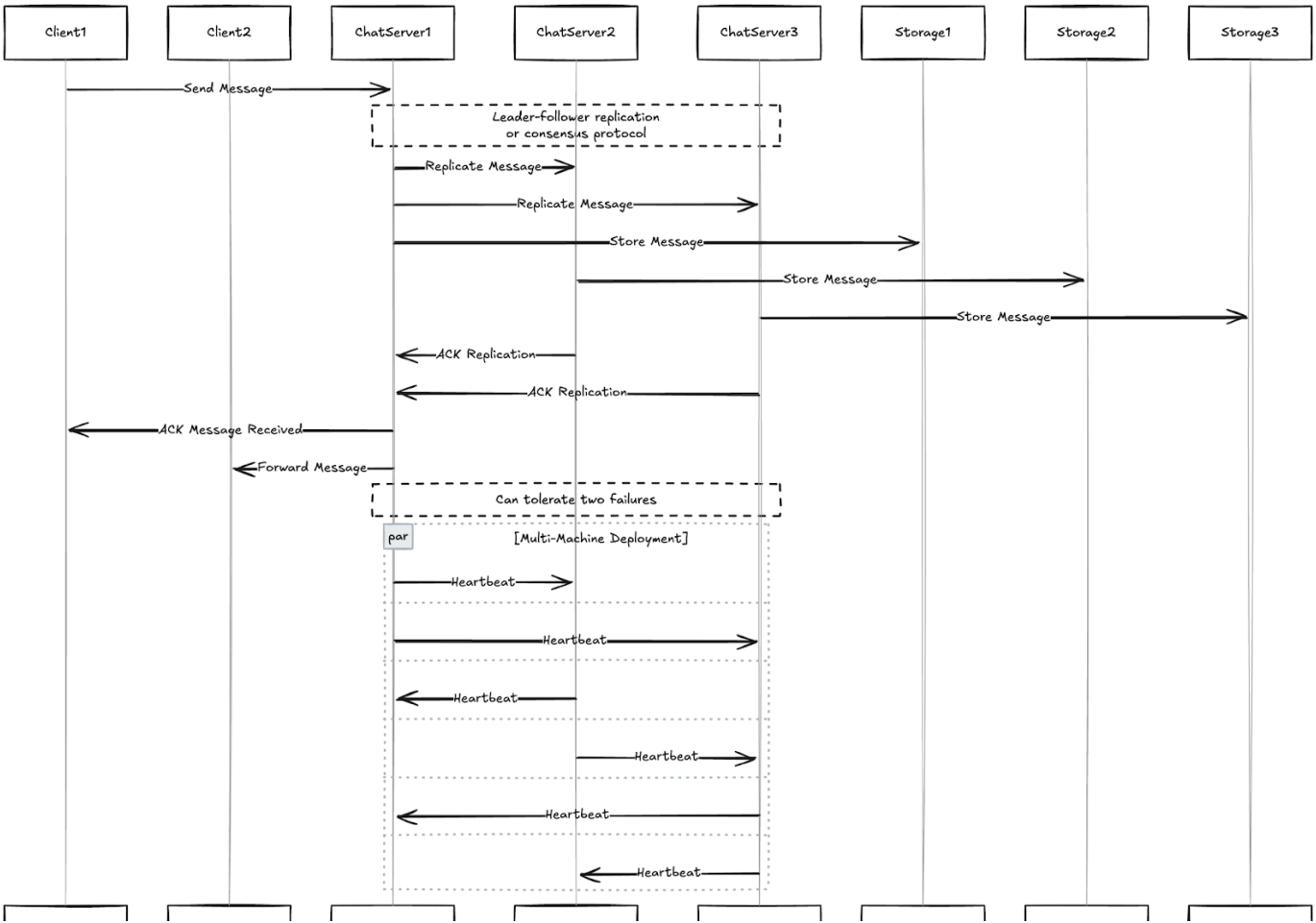
- **Automated Health Monitoring:**
  - Incorporate monitoring tools or scripts to check the number of active replicas.
  - Operate the instantiation of new servers if the replica count falls below the required level.
- **Orchestration Layer:**

- Consider adding an orchestration component to manage and coordinate the replicas, improving fault tolerance in real time.

A rough sketch of this system as a flow chat would look like:



A more detailed communication overview can be seen in the drawing below:



Design Decisions

Consistency Model

We opted for a **strong consistency** model to ensure that every write or update operation is coordinated among a majority of active replicas before a response is sent back to the client. This design guarantees that all servers reflect the latest state, an essential aspect given the small scale of our chat application. Although this may introduce some latency, the trade-off is acceptable to avoid issues that could arise in a larger system where inconsistent states would be much more problematic.

Client Interaction and Leadership

To streamline operations, all client requests are directed to the **leader node**. This node not only processes requests but also orchestrates the replication process by:

- Receiving client write requests (e.g., account creation, message sending, or marking messages as read).
- Forwarding these operations to the follower nodes.
- Waiting for acknowledgments from a majority of the active nodes before confirming the operation. This centralized approach helps maintain order and consistency across the distributed system.

Replication Strategy and Fault Tolerance

The backbone of our system is the ReplicationManager, which implements a basic leader-follower protocol. Key points include:

- Message and Data Replication: Each write operation (e.g., a new message or account creation) is replicated across all active servers. This is achieved through RPCs structured via Protocol Buffers, ensuring that all state changes (including unique message IDs) are consistently applied.
- Dynamic Active Set: Only nodes that are marked as active participate in the replication process. If a server goes down, it is dynamically removed from the active set, ensuring that the leader only waits for acknowledgments from reachable nodes.
- Two-Fault Tolerance: Our design tolerates up to two server failures by relying on a majority-based acknowledgment mechanism. This means that even if two servers fail, the system can still reach consensus and continue to function without data loss.

Leader Responsibilities and Election Mechanism

The leader is the central point for handling client requests and ensuring state consistency. Its responsibilities include:

- Processing Requests: Accepting and processing all write operations.
- Forwarding and Synchronization: Propagating operations to follower nodes and ensuring that each replica updates its local persistent store.
- Heartbeat Monitoring: Sending regular heartbeat messages to all replicas to monitor their health. If heartbeats from a majority of nodes are not received, an election is triggered.

- **Lead**er Election: In the event of a leader failure, an election mechanism (inspired by consensus algorithms like Raft, Paxos, or even the Bully Algorithm) is initiated to choose a new leader. This failover process ensures minimal service disruption.

Persistence and Storage

Each server maintains its own instance of persistent storage (i.e., a local database). This design decision eliminates any single point of failure and ensures that even if one server fails, the data remains intact on the surviving nodes. Our earlier implementation of persistent storage per server is a critical component of this approach, making it possible to stop and restart any server without losing historical data.

Additional Considerations

- **Dynamic Replica Addition:** While not a core requirement, we have designed the system with extensibility in mind. The architecture supports the dynamic addition of new servers into the replica set, which would include replicating the existing database state to the new node.
- **Automated Health Monitoring and Orchestration:** Future improvements may include implementing scripts or tools that continuously monitor the health of replicas. Such an orchestration layer would automatically instantiate new servers when the active replica count falls below the required threshold, further enhancing fault tolerance.

Communication Flow

A simplified overview of the communication process is as follows:

1. **Client Request:** A client sends a write request to any server.
2. **Forwarding:** If the contacted server is not the leader, it forwards the request to the current leader.
3. **Replication:** The leader processes the request, updates its local store, and replicates the change to follower nodes.
4. **Acknowledgment:** Once a majority of active nodes acknowledge the update, the leader confirms the operation to the client.
5. **Heartbeat:** Continuous heartbeat messages ensure that all replicas remain in sync, and any failure triggers a re-election process.

Implementation Details

Replication Strategy

We started by implementing a **ReplicationManager** class which uses a basic leader-follower protocol with majority acknowledgement from the active servers to manage replication between leader and follower servers (inspired by the Raft consensus algorithm). We started with methods to handle replicating messages and accounts across all servers, since we want to enforce

strong consistency and therefore each server needs to update the local persistent storage with any updates. Each server checks if they are marked as a leader or a follower, and forwards the replication handling to the leader server if they are not the leader themselves. The leader then waits for majority acknowledgement to determine whether replication is successful.

We then edited the application to handle replication for write operations. For instance, when creating an account, we wanted all servers to write the account information to their respective local databases. One change we made was to add new RPCs such as MarkRead so the servers handle these write operations rather than the UI, which originally was by default writing to the same chat.db file rather than each servers' local storages.

Leader Responsibility

The leader node is responsible for processing client requests and replicating the requests to all active follower nodes. This ensures that all operations are recorded in a centralized location for consistency purposes, before being propagated to other nodes. The code below shows an example of followers forwarding operations to the leader (this was specifically from the CreateAccount RPC method defined in our server).

```
if self.replication_manager.role != ServerRole.LEADER:
    self.logger.debug("Not leader, forwarding CreateAccount request to leader")
```

Active Node Majority

Replication only involves nodes that are currently active. The leader computes the majority based on the count of active nodes (including itself) and requires acknowledgments from a majority before committing any operation. This approach adapts to dynamic cluster membership, ensuring that the replication process remains robust even as nodes fail or recover and providing us with fault tolerance. The code below shows an example where the leader waits for a majority of the active nodes to send acknowledgement, otherwise it steps down.

```
# Decide if we still keep leadership based on majority of active servers
needed_acks = (alive_count // 2) + 1
if acks < needed_acks:
    logging.warning(
        f"Leader sees only {acks}/{alive_count} active acks, needed={needed_acks}. Stepping down."
    )
```

Replication

The system leverages gRPC to send replication messages, which are structured using Protocol Buffers. Each replication message contains relevant data—such as message IDs, sender, recipient, content, or account details—allowing followers to replicate the state change accurately. Methods like `replicate_message`, `replicate_account`, and `replicate_operation` handle these processes, ensuring that each operation is confirmed by enough replicas.

Heartbeat and Elections

The leader periodically sends heartbeat messages to all replicas to communicate activity and to detect failures. If a majority of active nodes become unresponsive, an election is triggered to select a new leader, ensuring that the system remains highly available.

Other Issues

When handling marking messages as read, we encountered an issue where servers writing to local storage may use different message IDs for the same message when performing the replication. We modified the database manager to take in a message ID when writing new messages to the database so that we could ensure consistency of message IDs across persistent stores, which simplifies the process of deleting messages as well.

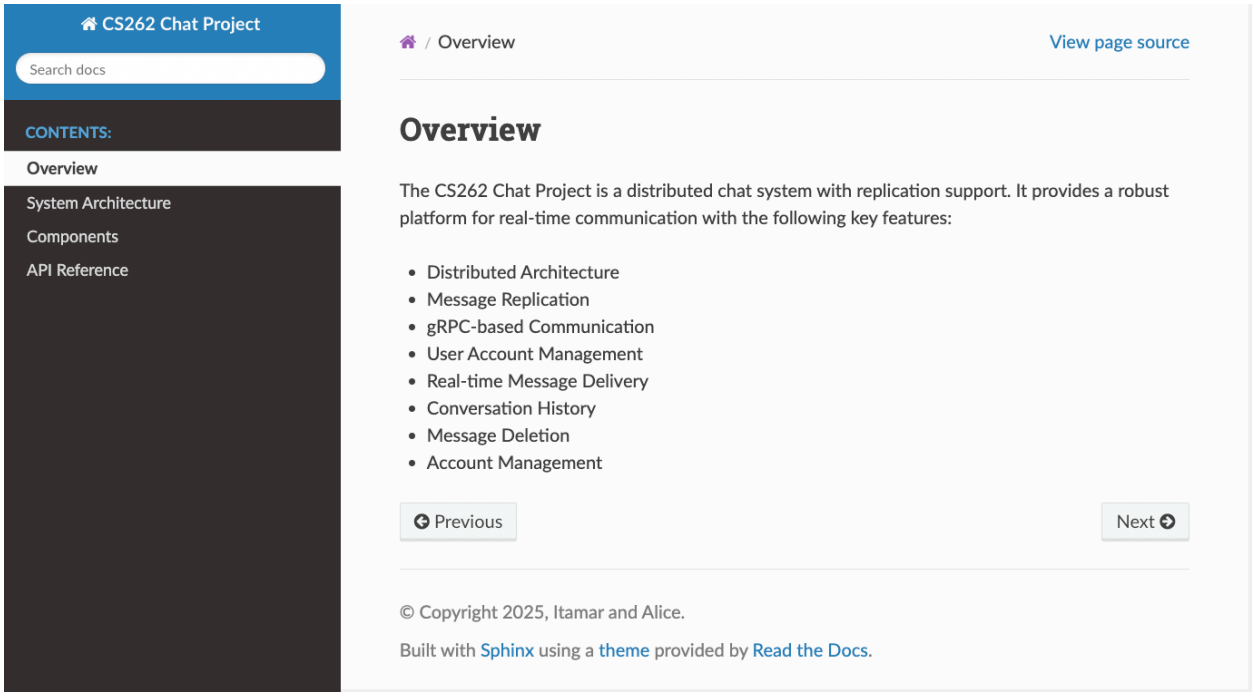
Since our implementation also takes in a specific list of server hosts/ports in the cluster, when we bring a server down we decided to implement functionality to remove it from a list of active servers. That way, we can avoid errors regarding the leader trying to send a heartbeat to the now inactive server. This also allows us to easily handle re-elections, as we only want the majority vote out of all the active servers.

Failover Mechanism

All servers exchange heartbeat messages to monitor the leader's health. Server shutdowns are detected when heartbeats are not received from a node after a certain period of time. When a server is shut down, the remaining active servers will undergo a re-election if the removed server was the leader. If a client was connected to the server that was shut down, it will automatically be reconnected to the new leader, allowing for minimal time loss while maintaining data integrity even in the face of multiple machine failures.

Docs

We also created some documentation for the project that can be accessed inside the repository, containing all the information about the code.



There, you can browse through our code's documentation. This doc was built using Sphinx and is based on the docstrings split across the code. In the README.md, you can find instructions on how to run and recreate the code.

Tests

We implemented tests for the RPC methods used by the client and server, as well as for database methods and for our ReplicationManager class which handles the replication operations. We included both unit and integration tests.