1. Network failure

(a) Packet delay

```
Client 1                        Server 1                        Client 2
RPC_lock_acquire();
                                                                RPC_lock_acquire();
                                ...grant the lock to C1
...lock acquired
RPC_append_file('A'); (delayed)

                                ...lock times out
                                ...grant the lock to C2
                                                                 ...lock acquired
                                (receive the delayed file_append)
                                ...refuse file_append from C1
```

Expected Result: Client 1 receives `ERROR: LOCK_EXPIRED` as Client 2 has already acquired the lock, and 'A' is not written to the file.

(b) Packet drop

- Client packet loss

```
Client 1                        Server 1                        Client 2
RPC_lock_acquire(); (lost)                                      RPC_lock_acquire();
                                ...grant the lock to C2
                                                                ...lock acquired
                                                                RPC_append_file('B');
                                ...append 'B'
                                                                ...'B' appended
                                                                RPC_lock_release();
                                ...release the lock for C2

retry: RPC_lock_acquire()
                                ...grant the lock to C1
...lock acquired
RPC_append_file('A');
                                ...append 'A'
...'A' appended
```

Expected Result: Client 1 is able to acquire the lock through the retry mechanism, and the file contains "BA".

- Server packet loss

```
Client 1                      Server 1                    Client 2
RPC_lock_acquire();
                                                          RPC_lock_acquire();
                              ...grant the lock to C1 (lost)
```

```
retry: RPC_lock_acquire()
                              ...inform C1 it has the lock
...lock acquired
RPC_append_file('A');
                              ...append 'A'
...'A' appended
RPC_lock_release();
                              ...release the lock for C1
                              ...grant the lock to C2
                                                          ...lock acquired
                                                          RPC_append_file('B');
                              ...append 'B'
                                                          ...'B' appended
                                                          RPC_lock_release();
                              ...release the lock for C2
```

Expected Result: Client 1 queries the server with retry, finds out that it holds the lock, and sends file_append request; the file contains "AB".

(c) Duplicated packets

- lock_release

```
Client 1                      Server 1                    Client 2
RPC_lock_acquire();
                                                          RPC_lock_acquire();
                              ...grant the lock to C1
...lock acquired
RPC_append_file('A');
                              ...append 'A'
...'A' appended
RPC_lock_release(); (delayed)

retry: RPC_lock_release();
                              ...release the lock for C1
...lock released              ...grant the lock to C2
                                                          ...lock acquired
                              (ignore the duplicated lock_release)
                                                          RPC_append_file('B');
                              ...append 'B'
                                                          ...'B' appended
                                                          ...wait for a while
RPC_lock_acquire();
                                                          RPC_append_file('B');
                              ...append 'B'
                                                          ...'B' appended
                                                          RPC_lock_release();
                              ...release the lock for C2
                              ...grant the lock to C1
...lock acquired
RPC_append_file('A');
```

```
                                ...append 'A'
...'A' appended
```

Expected Result: Client 1 does not mistakenly release the lock for Client 2, and it reacquries the lock after Client 2 releases it; the file contains "ABBA".

(d) Combined network failures

- Multiple packet drops & duplicated file_append

```
Client 1                    Server 1                   Client 2
RPC_lock_acquire();
                                                        RPC_lock_acquire();
                            ...grant the lock to C1
...lock acquired
RPC_append_file('1');
                            ...append '1'
...'1' appended
RPC_append_file('A'); (lost)

retry: RPC_append_file('A');
                            ...append 'A'
                            ...(reponse packet loss)

retry: RPC_append_file('A');
                            ...inform C1 file_append success
...'A' appended
RPC_lock_release();
                            ...release the lock for C1
                            ...grant the lock to C2
                                                       ...lock acquired
                                                       RPC_append_file('B');
                            ...append 'B'
                                                       ...'B' appended
```

Expected Result: Client 1 completes two file_append requests despite packet losses on both client and server sides; Client 2 then acquires the lock; the file contains "1AB," demonstrating idempotency.

2. Client fails/stucks

(a) Stucks before editing the file

```
Client 1                    Server 1                   Client 2
RPC_lock_acquire();
                                                        RPC_lock_acquire();
                            ...grant the lock to C1
...lock acquired

(...garbage collection happens)
                            ...lock times out (C1)
                            ...grant the lock to C2
```

```
                                                    ...lock acquired
                                                    RPC_append_file('B');
(...GC ends)                  ...append 'B'
RPC_append_file('A');                               ...'B' appended
                              ...refuse file_append from C1
...ERROR: LOCK_EXPIRED                              ...wair for a while
                                                    RPC_append_file('B');
                              ...append 'B'
RPC_lock_acquire();                                 ...'B' appended
                                                    RPC_lock_release();
                              ...release the lock for C2
                              ...grant the lock to C1
...lock acquired
RPC_append_file('A');
                              ...append 'A'
...'A' appended
RPC_append_file('A');
                              ...append 'A'
...'A' appended
```

Expected Result: Client 2 acquires the lock after Client 1's lock ownership expires; Client 1 receives `ERROR: LOCK_EXPIRED` after its long pause and needs to reacquire the lock; the file contains "BBAA" (with Client 2's appended data appearing first).

(b) Stucks after editing the file

```
Client 1                      Server 1                  Client 2
RPC_lock_acquire();
                                                        RPC_lock_acquire();
                              ...grant the lock to C1
...lock acquired
RPC_append_file('A');
                              ...append 'A'
...'A' appended
(...garbage collection happens)
                              ...lock times out (C1)
                              ...(option1) remove 'A'
                              ...grant the lock to C2
                                                        ...lock acquired
                                                        RPC_append_file('B');
(...GC ends)                  ...append 'B'
RPC_append_file('A');                                   ...'B' appended
                              ...refuse file_append from C1
...ERROR: LOCK_EXPIRED                                  ...wait for a while
                                                        RPC_append_file('B');
                              ...append 'B'
RPC_lock_acquire();                                     ...'B' appended
                                                        RPC_lock_release();
                              ...release the lock for C2
                              ...grant the lock to C1
...lock acquired
```

```
RPC_append_file('A');
                              ...append 'A'
...'A' appended
RPC_append_file('A');
                              ...append 'A'
...'A' appended
```

Expected Result: Client 1's unfinished requests are aborted: 'A' is removed from the file; Client 2 gets the lock after timeout and appends 'B' twice; Client 1 comes back and redo its critical section; the file contains "BBAA".

NOTE: The workflow above only shows one option (option1) to guarantee the atomicity of clients' critical sections, you can also implement other solutions (e.g., transaction-based) as long as you can guarantee the two 'A's of client 1 are appended together.

3. Single server fails

(a) Lock is free

```
Client 1                          Server 1
RPC_lock_acquire();
                                  ...grant the lock to C1
...lock acquired
RPC_append_file('A');
                                  ...append 'A'
...'A' appended
RPC_append_file('A');
                                  ...append 'A'
...'A' appended
RPC_lock_release();
                                  ...release the lock for C1
...lock released
                                  (x) Node crashes
RPC_lock_acquire();

                                  (v) Recovers
retry: RPC_lock_acquire();
                                  ...grant the lock to C1
RPC_append_file('1');
                                  ...append '1'
...'1' appended
RPC_lock_release();
                                  ...release the lock for C1
...lock released
```

Expected Result: Server is able to remember the previous commited data "AA", and it continues to accept clients' requests after recovery, thus Client 1 can acquire the lock again and append '1' to the file; the file contains "AA1".

(b) Lock is held

```
Client 1                  Server 1                  Client 2
RPC_lock_acquire();
                                                    RPC_lock_acquire();
                          ...grant the lock to C1
...lock acquired
RPC_append_file('A');
                          ...append 'A'
...'A' appended
RPC_lock_release();
                          ...release the lock for C1
                          ...grant the lock to C2
                                                    ...lock acquired
                                                    RPC_append_file('B');
                          ...append 'B'
                          (x) Node crashes          ...'B' appended
                                                    RPC_append_file('B');

                          (v) Recovers              retry: RPC_append_file('B');
                          ...append 'B'
                                                    ...'B' appended
RPC_lock_acquire();
                                                    RPC_lock_release();
                          ...release the lock for C2
                          ...grant the lock to C1
...lock acquired
RPC_append_file('A');
                          ...append 'A'
...'A' appended
```

Expected Result: Server is able to remember the previous commited data "AB" and the current lock holder Client 2, so that Client 2 can continue to send file_append requests after the server recovers from the failure; Client 1 can reaqruie the lock after Client 2 releases the lock; the file contains "ABBA".

4. Complex Cases: Failures with replication

(a) Replica node failures (fast recovery)

- Description: The cluster contains 3 server nodes, Server 1 acts as the primary node, with Server 2 and Server 3 serving as replica nodes. Client 1 and Client 2 each want to append a character—'A' and 'B', respectively—to three different files. After the first `file_append` request is received, Server 2 experiences a transient failure, going offline temporarily but recovering soon...
- Possible Steps in Details:
  - Client 1 sends RPC_lock_acquire() to Server 1(Primary)
  - Server 1 syncs the lock acquire request with Server 2 & 3 (replicas), and Server 2 & 3 reply ACK
  - Server nodes commit the lock state change, then grant the lock to Client 1
  - Client 1 sends RPC_file_append('A', "file_1"); Client 2 also wants to use the lock and sends RPC_lock_acquire()
  - Server 1 syncs the file append request with Server 2 & 3
  - Server 2 & 3 reply ACK

    - Server 1 commits the file change, then informs Client 1
    - Client 1 sends another RPC_file_append('A', "file_2")
    - Server 1 syncs the file append request with Server 2 & 3
    - **Server 2 fails; Server 3 replies ACK, while Server 2 does not reply**
    - (For strong consistency model, Server 1 retries to contact Server 2; For weak and bounded consistency models, Server 1 may proceed without waiting for Server 2... During the waiting time, clients may also retry their requests)
    - **Server 2 recovers quickly (within retry timeout thresholds)**
    - Server 1 successfully communicates with Server2, syncs and commits C1's file_append request
    - After the last append requests succeeds, Client 1 sends RPC_file_append('A', "file_3") and the server processes the request
    - Client 1 sends RPC_lock_release()
    - Server 1 syncs the lock release of C1 and the lock acquire request of C2 with Server 2 & 3, both servers reply ACK back
    - Server 1 commits the lock state changes, informs C1 the lock is released, and informs C2 the lock is granted
    - Client 2 sends RPC_file_append('B') three times for "file_1", "file_2", and "file_3", one at each time; Server 1 communicates with replicas (both replicas are available now) in order to commit the file changes
    - Server1 commits file changes and informs Client 2 for each operation
    - Client 2 sends RPC_lock_release()
    - ...

Expected Results:

- The lock server ensures that all three file_append operations from each client (Client 1 appending 'A' and Client 2 appending 'B') are atomic, meaning each client's operations are complete without interference from the other client's operations.
- Eventually, all the 3 files reflect the combined results of both clients' operations, and the lock server guarantees that there's no interleaving of critical sections: either all files will contain "BA" or "AB", which depends on whether the server times out Client 1's lock ownership during the failure of Server 2.
- Server 1 (Primary) is able to continue making progress on file_append requests once Server 2 has recovered; For strong consistency, the system ensures consistent state across nodes.
- During Server 2's failure, Server 1 can choose to proceed with file updates and lock state changes, but once Server 2 recovers from the failure, it needs to synchronizes with Server 1 to update missed file changes and lock state changes and ensure that the consistency guarantees are maintained.
- Both Clients are able to acquire the lock sequentially, allowing each to complete their append operations.

(b) Replica node failures (slow recovery)

- Description: The setup is the same as the previous scenario: Client 1 wants to append 'A' and Client 2 wants to append 'B' to three different files. But this time, during Client 1's file_append requests, Server 2 (a replica node) experiences a crash and goes offline for an extended period. **The recovery time of Server 2 exceeds all timeout thresholds, so the servers may take actions and proceed without waiting for its recovery. However, when Server 2 comes back online, it must synchronize with the primary node to ensure that the consistency guarantees of the chosen model (whether strong or weak consistency) are upheld.**

Expected Results:

- The lock server ensures that all three file_append operations from each client (Client 1 appending 'A' and Client 2 appending 'B') are atomic, meaning each client's operations are complete without interference from the other client's operations.
- The lock server guarantees that there's no interleaving of critical sections: either all files will contain "BA" (if Client 1's operations finish after Client 2) or "AB", which depends on whether the server times out Client 1's lock ownership during the failure of Server 2.
- For all consistency models, the servers can make progress, grant the lock to each client sequentially, and completes their append operations; During the crash of Server 2, Server 1 and Server 3 detect this failure, they can choose to continue processing client requests since the primary node is still functional.
- For Strong Consistency, once Server 2 has recovered, it will synchronize with the primary node to ensure strong consistency guarantees.
- For Weaker Consistency, upon Server 2's recovery, it properly synchronizes with Server 1 to update any missed file changes and lock state changes.
- If any clients do not hear responses from the primary node and go to contact replicas for help (Server 3 is always available in this scenario), Server 3 should tell the client that Server 1 is still running as the primary node, thus clients' requests are still sent to Server 1.

(c) Primary node failures (slow reovery outside critical sections)

- Description:
  - The cluster contains 3 server nodes: Server 1 starts as the primary node, with Server 2 and Server 3 as replica nodes.
  - Client 1, Client 2, and Client 3 each want to append a character—'A', 'B', or 'C', respectively —five times to the same file (e.g., Client 1 will send file_append('A', 'file_1') five times)
  - Client 1 completes all 5 `file_append` successfully, and Server 1 successfully commits Client 1's `file_append` and `lock_release` requests. But before processing Client 2 & 3's `lock_acquire` requests, the primary node Server 1 crashes
  - Replicas/Clients figure out that the primary node is not available
  - A new primary node is selected, and clients start to send their requests to the new primary node
  - Client 2 and 3 send the `lock_acquire` requests to the new primary node
  - The new primary node process the rest of Client 2 and 3's requests
  - When Server 1 recovers from the crash, it figures out the new primary node and updates all file changes and the lock state based on the consistency guarantees

Expected Results:

- The lock server guarantees the atomicity of all file_append operations from each client, ensuring that the characters are appended consecutively without interleaving (e.g., "AAAAABBBBBCCCCC" or "AAAAACCCCCBBBBB").
- Replica nodes and clients can detect the failure, and a new primary node will be elected (based on the length of their log entries), maintaining the liveness property of the system.
- Clients will be notified about the decision of the new primary and send the unprocessed lock_acquire requests to the new primary node.
- Clients communicate with the new primary nodes, acquire the lock sequentially, and complete all file_append operations.
- For Strong Consistency: The new primary node can immediately take over with up-to-date information and continue processing client requests without issues.
- For Weak/Bounded Consistency: if new new primary node does not have the most up-to-date states, it needs to first ensure correctness before proceeding.

- Once Server 1 recovers from the failure, it will synchronize with the new primary node to ensure that the consistency guarantees of the chosen model (strong or weak consistency) are maintained.

(d) Primary node failures (slow reovery during critical sections) + stress test for atomicity

- Description:
  - The cluster contains 3 server nodes: Server 1 starts as the primary node, with Server 2 and Server 3 as replica nodes
  - Client 1, Client 2, and Client 3 each want to append a character—'A', 'B', or 'C', respectively—20 times to the same file (this time, we have longer critical sections)
  - Client 1 completes all 20 `file_append` successfully and releases the lock. But in the middle of Client 2's `file_append` requests, Server 1 crashes (OMG)
  - Replicas/Clients figure out that the primary node is not available
  - A new primary node is decided
  - The new primary node figures out the states of the system and fixes any issues
  - Client 2 and 3 send the rest of their requests to the new primary node
  - When Server 1 recovers from the crash, it figures out the new primary node and updates all file changes and the lock state based on the consistency guarantees

Expected Results:

- The lock server guarantees the atomicity of all file_append operations from each client, ensuring that all 20 characters of each client are appended consecutively without interleaving.
- Replica nodes and clients can detect the failure, and a new primary node will be elected based on the length of each server's log entries, maintaining the liveness property of the system.
- Clients can recognize the decision of the new primary node and send the rest of their requests to the new primary node.
- The new primary node is capable of handling uncommitted file_append requests and aborted critical sections.
- For Strong Consistency: The new primary node can immediately take over with up-to-date information and continue processing client requests without issues.
- For Weak/Bounded Consistency: if new new primary node does not have the most up-to-date states, it needs to first ensure correctness before proceeding.
- Once Server 1 recovers from the failure, it will synchronize with the new primary node to ensure that the consistency guarantees of the chosen model are maintained.
- All 3 clients are able to acquire the lock sequentially, allowing each to complete their respective file_append operations.

(e) Primary + Replica node failures

- Description:
  - The cluster contains 3 server nodes: Server 1 starts as the primary node, with Server 2 and Server 3 as replica nodes
  - Client 1, Client 2, and Client 3 each want to append a character—'A', 'B', and 'C', respectively—ten times across five different files (appending twice per file)
  - First Failure: During Client 1's 10 file_append operations, Server 2 crashes. Server 1 and Server 3 detect this failure, since the primary node is still operational, the cluster can choose to continue processing client requests
  - Server 2 recovers, updating the lock and file states from the primary node (Server 1)
  - Second Failure: Later, both Server 1 and Server 2 crash
  - The system becomes unavailable: With both peers down, Server 3 detects the majority of the cluster is offline and halts processing client requests until its peers recover

    - Recovery: Once either Server 1 or Server 2 recovers, a majority is reestablished, and the cluster resumes functionality. A new primary node is elected based on the length of each server's log entries.
    - Clients then direct requests to the new primary node, while the replica servers synchronize necessary file updates and lock states to maintain system consistency guarantees

Expected Results:

- Atomicity: the lock server guarantees atomicity for each client's file_append operations, ensuring that eventually the sequences in all 5 files follow the same order of "AA", "BB", and "CC".
- During the first failure of Server 2, the servers can choose to continue processing client requests, because the primary node + Server 3 are still alive.
- When both Server 1 and Server 2 fail, Server 3 detects the failure of its peers, and the cluster becomes unavailable.
- Once either Server 1 or Server 2 recovers, the node with the most log entries becomes the new primary, restoring the system's functionality.
- The new primary node is capable of managing uncommitted file_append requests and resolving any interrupted critical sections.
- For Strong Consistency: when servers recover, they immediately synchronize with the primary to ensure full state consistency.
- For Strong Consistency: when a new primary node is selected, it has up-to-date information, allowing it to take over immediately and continue processing client requests.
- For Weak/Bounded Consistency: the system may continue processing client requests even if some nodes have outdated logs. When failed nodes recover, they synchronize to eventually achieve consistency.
- For Weak/Bounded Consistency: a newly elected primary without the latest state needs to first ensure correctness before proceeding with client requests.
- All 3 clients are able to acquire the lock sequentially, allowing each to complete their respective file_append operations.