

1. Network failure

(a) Packet delay

Client 1	Server 1	Client 2
RPC_lock_acquire();		
	...grant the lock to C1	RPC_lock_acquire();
...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	
	...append 'B'	...lock acquired
		RPC_append_file('B');
	...release the lock for C2	...'B' appended
		RPC_lock_release();
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()
...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();		
	...grant the lock to C1	RPC_lock_acquire();
...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 reacquires 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();		
	...grant the lock to C1	RPC_lock_acquire();
...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	
	...append 'B'	...lock acquired
		RPC_append_file('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();		
	...grant the lock to C1	RPC_lock_acquire();
...lock acquired		
(...garbage collection happens)	...lock times out (C1)	
	...grant the lock to C2	

		...lock acquired RPC_append_file('B');
(...GC ends) RPC_append_file('A');	...append 'B'	...'B' appended
...ERROR: LOCK_EXPIRED	...refuse file_append from C1	...wait for a while RPC_append_file('B');
RPC_lock_acquire();	...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 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) RPC_append_file('A');	...append 'B'	...'B' appended
...ERROR: LOCK_EXPIRED	...refuse file_append from C1	...wait for a while RPC_append_file('B');
RPC_lock_acquire();	...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	
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();	
...	lock acquired
RPC_append_file('A');	...
...	'A' appended
RPC_append_file('A');	...
...	'A' appended
RPC_lock_release();	...
...	lock released
RPC_lock_acquire();	
retry: RPC_lock_acquire();	
RPC_append_file('1');	
...	'1' appended
RPC_lock_release();	
...	lock released

Expected Result: Server is able to remember the previous committed 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();		
	...grant the lock to C1	RPC_lock_acquire();
...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
	(x) Node crashes	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 committed 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 reacquire 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: 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...
- Detailed Steps:
 - 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 crashes; 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 clients and servers' retry timeout thresholds, so no further actions are taken)**
- Server 1 successfully communicates with Server2, syncs and commits C1's `file_append` request, reply back to C1
- Client 1 sends `RPC_file_append('A', "file_3")` and the server process 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
- 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.
- For Strong Consistency, Server 1 (Primary) is able to continue making progress on `file_append` requests once Server 2 has recovered, ensuring consistent state across nodes; For Weak/Bounded Consistency, Server 1 proceeds with file updates and lock state changes while Server 2 is offline, and upon Server 2's recovery, it synchronizes with Server 1 to update missed file changes and lock state changes.
- 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, after receiving the first `file_append` request, 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, regardless of the consistency model. 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, Server 1 (Primary) is able to continue making progress.
- For Strong Consistency, once Server 2 has recovered, it will synchronize with the primary node to ensure strong consistency guarantees.
- For Weaker Consistency, Server 1 can directly proceed without waiting for Server 2, and 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.
- Both Clients are able to acquire the lock sequentially, allowing each to complete their append operations.

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

- Description:
 - Server 1 acts as the primary node, with Server 2 and Server 3 serving 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 the new primary node notifies clients it starts to receive requests
 - Client 2 and 3 send the `lock_acquire` requests to the new primary node
 - The new primary node is processing 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 elect a new primary node, 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: The new primary node must first synchronize with available replicas or clients or wait for a timeout to ensure the correctness of the state 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 recovery during critical sections) + stress test for atomicity

- Description:
 - Server 1 acts as the primary node, with Server 2 and Server 3 serving 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, fixes any issues, and notifies clients it starts to receive requests
 - 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 elect a new primary node, maintaining the liveness property of the system.
- Clients will be notified about the decision of the new primary.
- 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: The new primary node must first synchronize with available replicas or clients or wait for `lock_release/timeout` to ensure the correctness of the state 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.
- All 3 clients are able to acquire the lock sequentially, allowing each to complete their respective `file_append` operations.

(e) Multiple server node failures

- Description:
 - 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, remove Server 2 from the active replicas, and continue processing client requests.
 - Server 2 recovers, rejoining the cluster and updating the lock and file states from the primary node (Server 1).
 - Second Failure: Later, both Server 1 and Server 2 crash simultaneously.
 - New Primary Election: Server 3 detects that both peers have gone down and promotes itself to primary, taking over client requests.
 - Clients redirect their requests to Server 3, the new primary node.
 - When Server 1 & 2 recover, they recognize Server 3 as the new primary and synchronize all file changes and lock states, respecting the system's 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, Server 1 (the primary) detects the failure, removes it from the active replicas, and continues processing requests.
- When both Server 1 and Server 2 fail, Server 3 detects the failures of its peers and promotes itself to primary, maintaining the liveness property of the system.
- Clients will be notified when Server 3 becomes the new primary node.
- The new primary (Server 3) is capable of managing uncommitted file_append requests and resolving any interrupted critical sections.
- For Strong Consistency: When servers recover, they synchronize with the primary to ensure full state consistency.
- For Strong Consistency: when Server 3 becomes the new primary node, it has up-to-date information, allowing it to take over immediately and continue processing client requests.
- For Weak/Bounded Consistency: during failures, the system may continue to process client requests. When failed nodes recover, they synchronize to eventually meet consistency guarantees.
- For Weak/Bounded Consistency: when Server 3 becomes the new primary node, it should first synchronize with any available replicas or clients or wait for lock_release/timeout to ensure the correct state before proceeding.
- All 3 clients are able to acquire the lock sequentially, allowing each to complete their respective file_append operations.