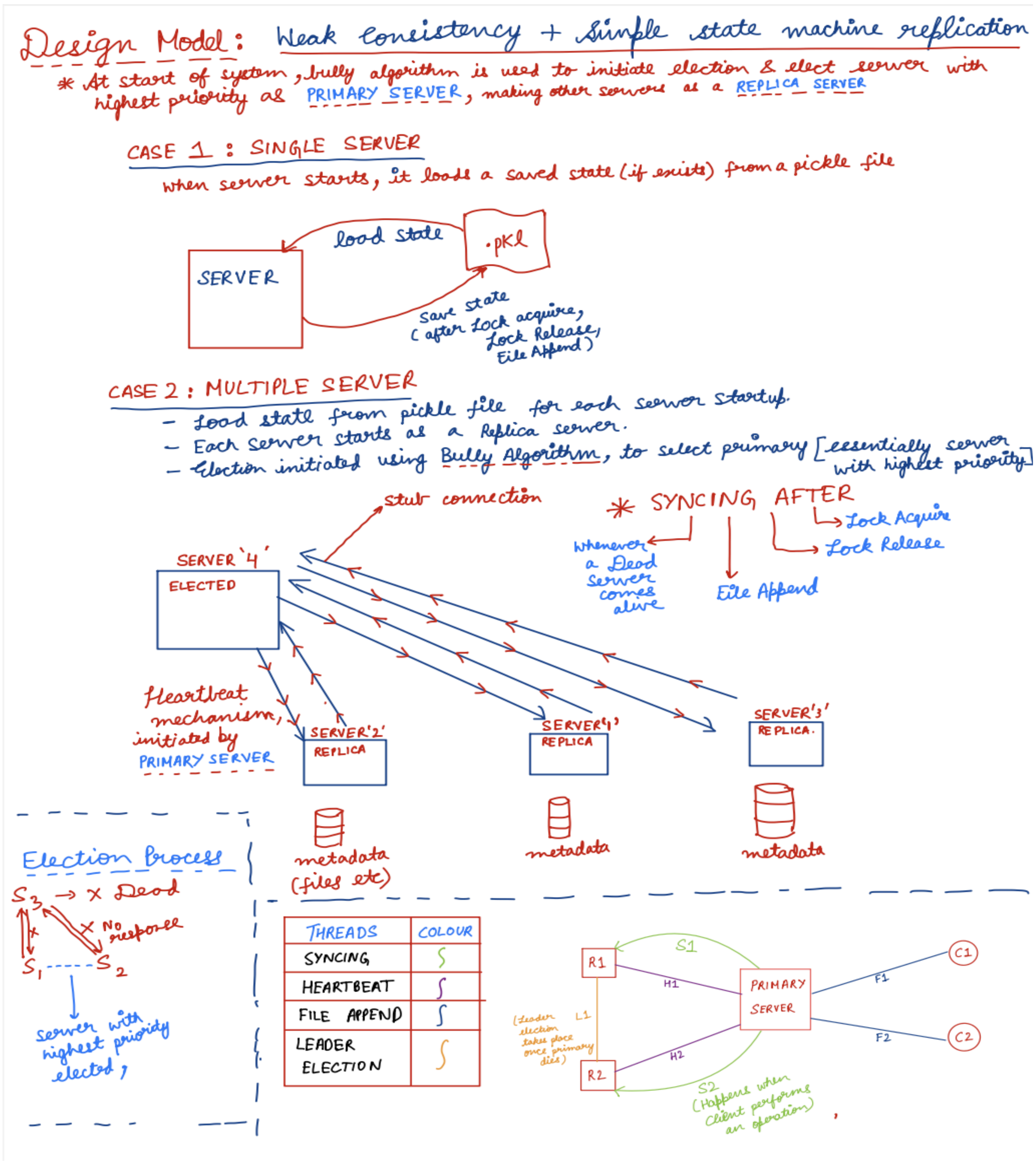
Distributed Systems Design Document

University of Edinburgh
Abhay Maurya (s2029730), Arnesh Saha (s2223898) , Arjandeep Singh Bawa (s2232089)

System Overview:

The Distributed Lock Manager (DLM) allows synchronized access to shared resources in a distributed system by employing a state-machine replication model with weak consistency. The system is designed to function in cases with potential server node crashes and unreliable networks.



Design Model: Weak consistency + Simple state machine replication

* At start of system, bully algorithm is used to initiate election & elect server with highest priority as PRIMARY SERVER, making other servers as a REPLICA SERVER

CASE 1 : SINGLE SERVER

when server starts, it loads a saved state (if exists) from a pickle file

SERVER
load state
.pkl
save state (after Lock acquire, Lock Release, File Append)

CASE 2 : MULTIPLE SERVER

- Load state from pickle file for each server startup.
- Each server starts as a Replica server.
- Election initiated using Bully Algorithm, to select primary [essentially server with highest priority]

* SYNCING AFTER
  - Whenever a Dead Server comes alive
  - File Append
  - Lock Acquire
  - Lock Release

Election Process

THREADS	COLOUR
SYNCING	∫
HEARTBEAT	∫
FILE APPEND	∫
LEADER ELECTION	∫

R1
R2
PRIMARY SERVER
C1
C2
S1
H1
H2
F1
F2
L1
(Leader election takes place once primary dies)
S2 (Happens when Client performs an operation)

System Model & Key Features:

This section gives an overview of the system and the key design choices that we have made.

1. **Weak Consistency Model:** The system prioritizes performance and low latency by allowing replicas to momentarily hold stale data. Synchronization is enforced during critical operations such as lock acquisition and release and file append, ensuring eventual consistency without significant overhead.Additionally, synchronization with replicas runs on a separate thread, ensuring that operations are not delayed while waiting for replicas to acknowledge updates.
2. **State-Machine Replication:** A primary node processes all client requests and synchronizes the state periodically with its replicas. This approach ensures that replicas maintain a consistent state through serialized updates from the primary, enabling seamless failover.
3. **Timeout-Based Lock Management:** Locks are assigned a fixed expiration time to prevent deadlocks caused by client or server crashes. Clients are required to renew locks before expiration to continue using them, ensuring resource availability to others.
4. **Bully Algorithm for Leader Election during node crashes:** We have chosen this algorithm for its simplicity and suitability for our use case. It provides a deterministic and fair leader election process. Unlike Raft or more complex algorithms, it avoids the overhead of distributed consensus, and compared to ring-based algorithms, it offers faster convergence by directly involving higher-priority nodes. Its time complexity is **O(n^2)**, which can be high in large systems, but is manageable for our limited replica set. This makes it an efficient and practical choice for minimizing downtime during failovers.
5. **Heartbeat Mechanism:** Heartbeat signals are continuously exchanged between the primary and replicas to monitor liveness. Failures detected via heartbeat absence trigger leader elections, enabling quick recovery and continuity.
6. **Retry Mechanism and Idempotency:** The system employs retry mechanisms for clients to handle packet loss or server unavailability gracefully. Servers track processed requests to ensure idempotency, preventing duplicate effects from retries.

Limitations and Tradeoffs / Assumptions:

- **Outdated primary node during recovery:**
  - **Limitation:** The current sync mechanism is primary-initiated, which can lead to a critical bottleneck during recovery. If a replica crashes and then comes back online in isolation (while other nodes are offline), it can be elected as the new primary through the Bully algorithm. This new primary, with outdated data, can overwrite more up-to-date replicas as they reconnect, leading to propagation of stale state across the system.
  - **Assumption**: We assume that, in most cases, at least one node will remain online to mitigate the impact of this issue. Additionally, the system assumes no malicious behavior from replicas (i.e., Byzantine failures are not considered).

- **File synchronization bottleneck:**
  - **Limitation:** The system synchronizes all files with replicas during state updates, regardless of whether they have changed. As the number of files grows, this approach introduces significant latency and resource overhead, particularly during high-frequency updates or when a replica re-joins the cluster.

- **Analysis:** This design prioritizes simplicity but creates inefficiencies for systems with a large number of files. A more efficient approach could involve delta-based synchronization, where only modified files or state changes are propagated to replicas, reducing the synchronization overhead.
- **Assumption**: The system assumes that the number of files and the frequency of updates remain within a manageable range to minimize the impact of this bottleneck. It also assumes sufficient network bandwidth and storage for handling full file synchronizations.