Rapport de Projet : Implémentation du Consensus Raft et Stockage Clé-Valeur Distribué

Réalisé par

Nolan Pujol

Oussama Daoudi

BUT Informatique 3ème année Déploiement d'Application Communicantes et Sécurisées (DACS)

Année universitaire 2025-2026

Sommaire

1. Introduction et Objectifs **3**
1.1 La Problématique du Consensus Distribué 3
1.2 Pourquoi Raft et Approche Progressive ? 3
2. Architecture Logicielle **4**
2.1 Structure Modulaire des Composants 4
2.2 Modèle de Concurrence et Sécurité 5
2.3 Diagramme de Séquence : Flux d'une Écriture (PUT) 6
3. Analyse Technique Approfondie **6**
3.1 Gestion Réseau Asynchrone et Non-Bloquante 6
3.2 Le Mécanisme de Réplication de Logs (Log Replication) 7
3.3 La Machine à États Finis (KV Store) 8
4. Scénarios de Test et Validation **8**
4.1 Validation du Mécanisme de "Backtracking" (Convergence des Logs) 8
4.2 Validation de la Machine à États (KV Store) 9
5. Déploiement et Configuration **10**
5.1 Évolution de l'Arborescence 10
5.2 Configuration du Cluster (nodes.conf) 10
./node <ID> nodes.conf 10
5.3 Chaîne de Compilation et Lancement 11
6. Scénarios de Test et Validation 11
7. Bilan et Perspectives **12**
7.1 Points Forts de l'Implémentation 12
7.2 Limitations Actuelles 12
8. Conclusion **13**
Test du projet **14**

1. Introduction et Objectifs

Dans le cadre du module de Programmation Système Avancée, ce projet vise à concevoir et implémenter un système distribué résilient aux pannes. L'objectif final est la mise en œuvre d'une table de hachage distribuée (Key-Value Store) reposant sur l'algorithme de consensus Raft.

Le développement a été réalisé en langage C, en utilisant les primitives systèmes bas niveau (Sockets TCP, Threads POSIX, Mutex) pour garantir une maîtrise fine des ressources et de la concurrence.

Le système développé garantit les propriétés suivantes :

- Coordination Centralisée : Un *Leader* unique est élu pour orchestrer le cluster.
- Persistance et Réplication : Les commandes d'écriture (PUT) sont répliquées de manière fiable dans un journal d'opérations (*Log*).
- Consensus (Quorum) : Les données ne sont appliquées (*Committed*) que lorsqu'une majorité de nœuds a confirmé leur réception.
- Haute Disponibilité : Le service reste opérationnel pour les clients tant qu'une majorité de nœuds est active.

1.1 La Problématique du Consensus Distribué

Dans un système centralisé classique, la cohérence des données est triviale car la mémoire est partagée. Dans un système distribué, nous faisons face aux failles réseaux (latence, partitionnement) et matérielles (crashs).

Notre projet s'inscrit dans la problématique du Théorème CAP (Consistency, Availability, Partition tolerance). L'algorithme Raft privilégie la Cohérence (C) et la Tolérance au Partitionnement (P) :

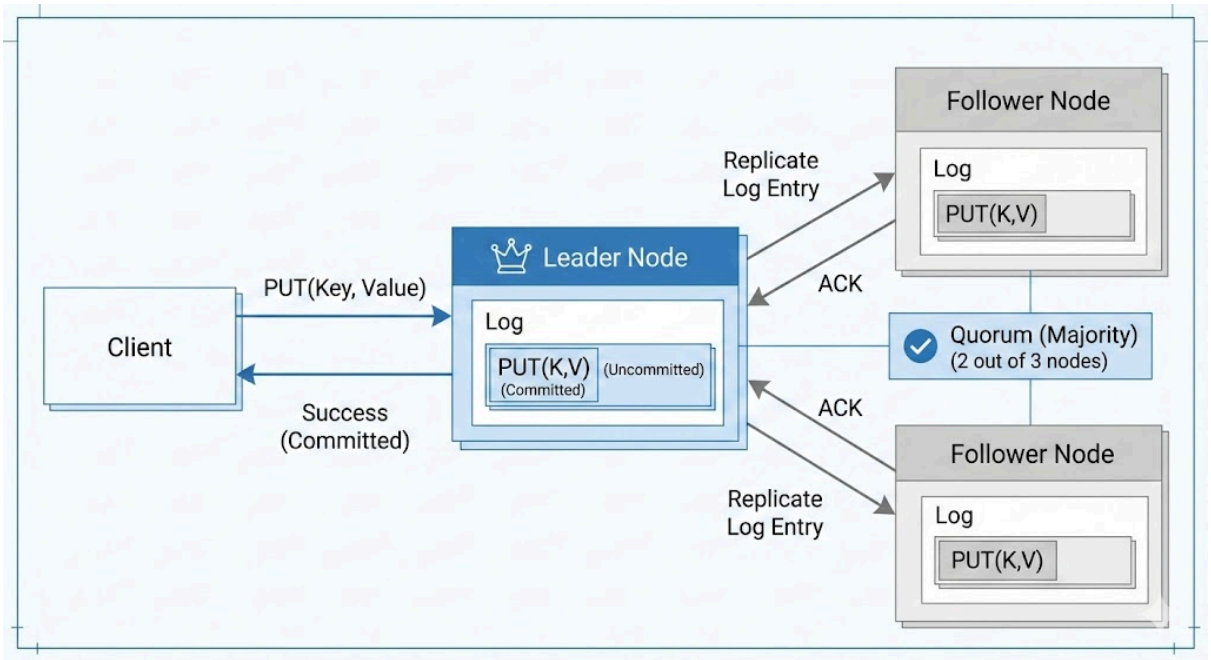
- En fonctionnement normal, le système est disponible et cohérent.
- Si une partition réseau isole une minorité de nœuds, ceux-ci cessent de valider des écritures pour préserver la "vérité" du système, sacrifiant temporairement la disponibilité globale pour garantir qu'aucune divergence de données ne survienne.

1.2 Pourquoi Raft et Approche Progressive ?

Contrairement à l'algorithme Paxos, historiquement réputé pour sa complexité d'implémentation, Raft a été conçu spécifiquement pour être compréhensible et implémentable. Il décompose le problème monolithique du consensus en sous-problèmes distincts.

Nous avons adopté cette philosophie de décomposition pour structurer notre projet en trois jalons cumulatifs :

1. Infrastructure Réseau (Step 1) : Mise en place de la communication TCP maillée (Ping/Pong) pour que les nœuds se découvrent.
2. L'Élection du Leader (Step 2) : Implémentation des *Timers* aléatoires et de la machine à états pour élire un coordinateur unique.
3. La Réplication des Logs (Step 3) : Propagation des données et sécurisation du consensus (Safety) pour garantir qu'aucune donnée validée ne soit perdue.



2. Architecture Logicielle

L'architecture du projet repose sur une communication pair-à-pair (P2P) via des sockets TCP. Il s'agit d'une architecture symétrique : chaque nœud exécute exactement le même binaire au démarrage. C'est l'algorithme Raft qui détermine dynamiquement le rôle de chaque processus à un instant donné (Follower, Candidate ou Leader).

2.1 Structure Modulaire des Composants

Nous avons appliqué le principe de séparation des préoccupations pour garantir un code maintenable. Le tableau ci-dessous détaille les responsabilités de chaque module :

Fichier	Rôle	Détail Technique
node.c	Cœur du système (Machine à états)	Gère les transitions d'états (via un switch case), le journal des opérations (Log) et l'application des données dans la Hashmap (KV Store). Contient la boucle principale node_tick.
rpc.c	Couche Transport	Abstrait le réseau. Implémente un **serveur TCP multithreadé** et un client avec timeout non-bloquant (utilisation de select() et fcntl). Gère la sérialisation des paquets.
client.c	Interface Utilisateur	Programme distinct permettant d'envoyer des commandes PUT/GET. Gère la **redirection automatique** : si le nœud contacté n'est pas Leader, le client se reconnecte au bon nœud.
config.c	Configuration	Parser de fichier texte. Charge dynamiquement la topologie du cluster (ID, IP, Port) depuis le fichier nodes.conf au démarrage.

2.2 Modèle de Concurrence et Sécurité

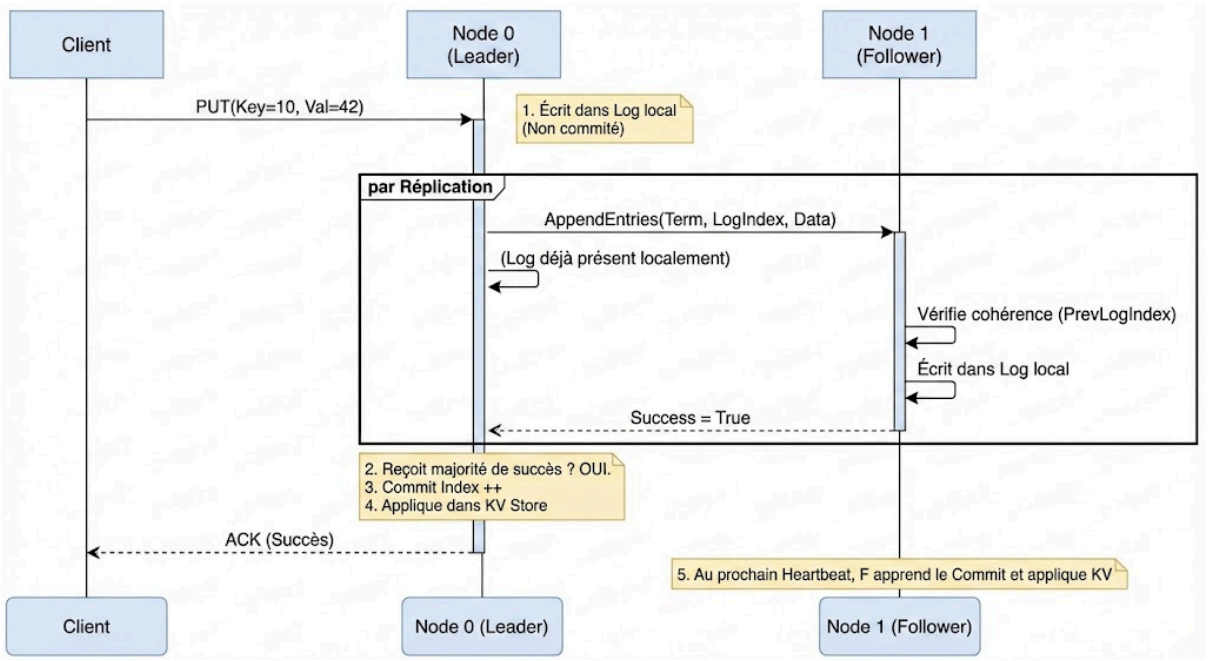
Notre implémentation utilise une architecture **multi-threadée** pour garantir la réactivité du système :

1. **Thread Principal (Tick Loop) :** Gère l'horloge interne de Raft, vérifie les timeouts d'élection et envoie les heartbeats.
2. **Thread Réseau (Server Listener) :** Écoute en permanence sur le port TCP (`accept`).
3. **Threads Workers :** Pour chaque connexion entrante, un thread est détaché (`pthread_detach`) pour traiter la requête sans bloquer les autres.

Sécurité (Thread Safety) : L'accès à la structure centrale `node_t` est protégé par un **Mutex** (`pthread_mutex_t lock`). Cela empêche les *Race Conditions*, par exemple entre le thread réseau qui reçoit un vote et le thread principal qui déclenche une élection.

2.3 Diagramme de Séquence : Flux d'une Écriture (PUT)

Le schéma ci-dessous illustre le chemin critique d'une donnée. Il met en évidence le rôle central du Leader qui reçoit la requête, la stocke localement (Log), la propage aux Followers, et attend le Quorum avant de valider.



3. Analyse Technique Approfondie

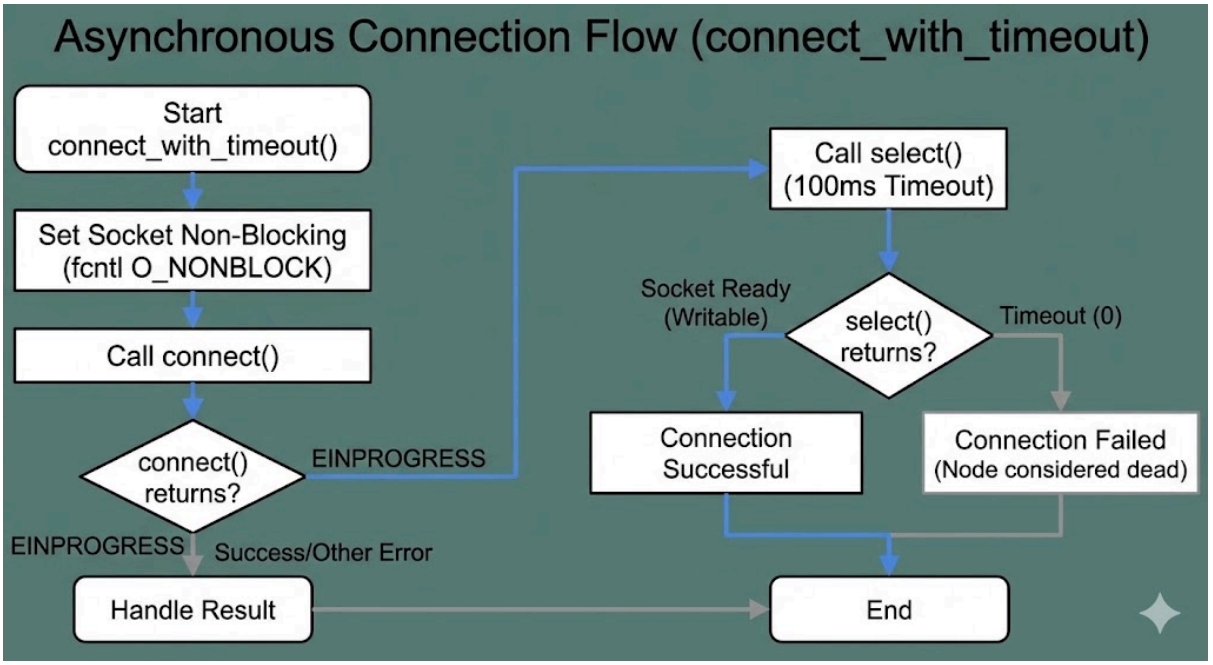
3.1 Gestion Réseau Asynchrone et Non-Bloquante

Une des difficultés majeures de la programmation système en C est la gestion des appels bloquants qui peuvent figer l'application. Dans notre fonction `connect_with_timeout` (fichier `rpc.c`), nous avons contourné le comportement par défaut de l'appel système `connect()` pour éviter que le Leader ne reste bloqué si un Follower est éteint.

Cette gestion repose sur trois étapes techniques :

1. Mode Non-Bloquant : Nous utilisons `fcntl(sock, F_SETFL, O_NONBLOCK)` pour passer la socket en mode non-bloquant. L'appel à `connect()` rend alors la main immédiatement, renvoyant souvent l'état `EINPROGRESS` pour signaler que la tentative est en cours.
2. Multiplexage d'E/S : Nous utilisons la fonction `select()` pour surveiller l'état de la socket.
3. Timeout Strict : Nous demandons au noyau d'attendre au maximum 100ms que la socket soit prête en écriture. Si le délai expire sans réponse, nous considérons le nœud comme indisponible.

Cette approche est cruciale pour la réactivité du cluster : le Leader détecte une panne quasi-instantanément au lieu d'attendre le timeout TCP par défaut (souvent 60 secondes), ce qui paralyserait le système.



3.2 Le Mécanisme de Réplication de Logs (Log Replication)

C'est le cœur de l'étape 3. L'implémentation respecte la propriété de sécurité de Raft via deux mécanismes de contrôle rigoureux.

A. Vérification de Précédence (Consistency Check) : Dans la fonction `node_handle_append_entries`, le Follower vérifie systématiquement la continuité de son historique avant d'accepter de nouvelles données. Il compare le terme de son dernier log avec le `prev_log_term` envoyé par le Leader

```
// Extrait de node.c

if (node.log[msg->prev_log_index].term != msg->prev_log_term) {
```

```
    return rep; // Échec -> Le leader devra reculer (backtrack)

}
```

Si cette condition n'est pas remplie, cela signifie qu' il y a une divergence dans l'historique (un "trou" ou un conflit). Le Follower rejette alors la requête.

B. Algorithme de Backtracking (Réparation) Lorsqu'un Leader reçoit un rejet (`success = 0`) d'un Follower, il enclenche le mécanisme de réparation. Il décrémente son `next_index` spécifique pour ce Follower et réessaie l'envoi au prochain cycle (`tick`). Ce processus se répète jusqu'à trouver un point de synchronisation commun, garantissant la convergence des logs.

C. Engagement (Commit) Le Leader ne considère une donnée comme pérenne que lorsqu'il a reçu une confirmation d'une majorité de nœuds (`peer_count / 2 + 1`). Il met alors à jour le `commit_index`, rendant l'opération irréversible.

3.3 La Machine à États Finis (KV Store)

- La fonction `apply_log_to_kv` agit comme l'interface finale entre le consensus (les Logs) et l'application (la table de hachage).
- Elle parcourt les logs validés, de `last_applied + 1` jusqu'à `commit_index`, pour mettre à jour la mémoire. Cette conception assure l'idempotence : rejouer le log dans le même ordre produira toujours exactement le même état final de la table.
- Pour les opérations de lecture (GET), le système lit directement la mémoire locale du nœud contacté. Cela permet une réponse rapide, en acceptant une cohérence éventuelle (Eventual Consistency) si le nœud interrogé est un Follower avec un léger retard de réplication.

4. Scénarios de Test et Validation

Cette section détaille les mécanismes de validation mis en place pour garantir la robustesse du cluster, notamment sa capacité à s'auto-réparer (convergence) et à traiter les requêtes clients.

4.1 Validation du Mécanisme de "Backtracking" (Convergence des Logs)

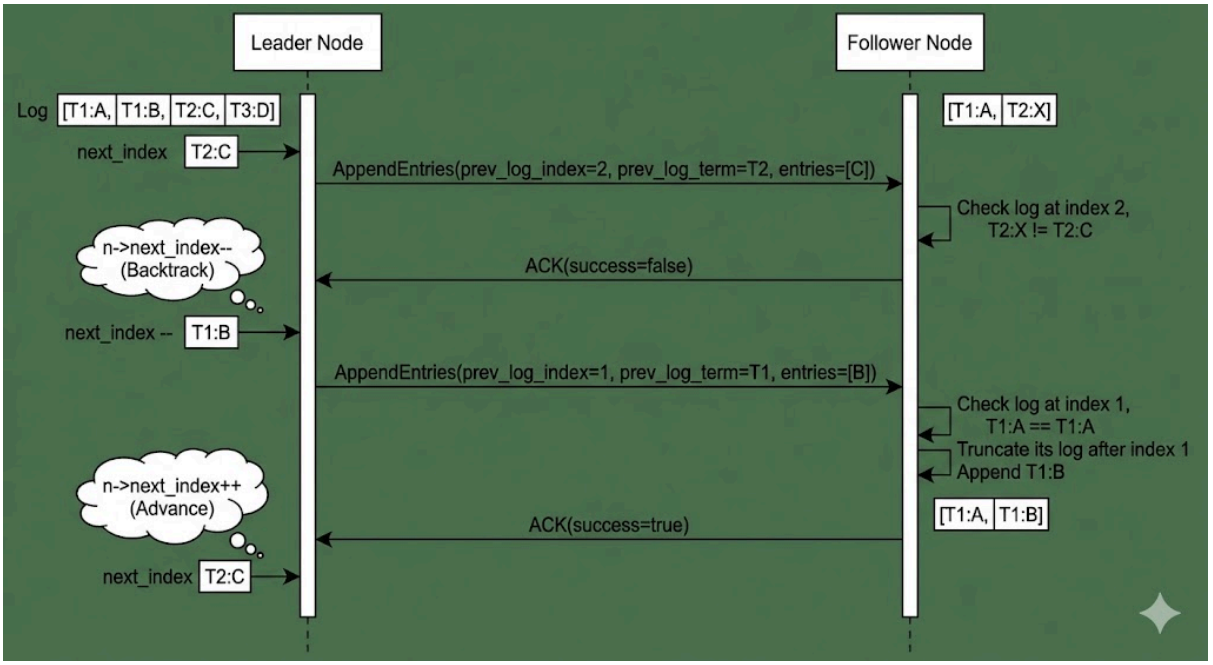
Dans les systèmes distribués, il est fréquent que les journaux des nœuds divergent (paquets perdus, redémarrages). Nous avons validé le mécanisme de réparation implémenté dans la fonction `send_append_entries_to_peer`.

Le fonctionnement validé est le suivant :

1. Le Leader maintient un tableau `next_index[]` pour chaque follower, estimant la prochaine position libre dans leur journal.
2. Si un Follower rejette un message `APPEND_ENTRIES` (réponse `success = 0`), cela signale un "trou" ou un conflit dans son historique.
3. L'algorithme de correction s'active : le Leader décrémente `next_index` pour ce pair spécifique et réessaie l'envoi au prochain cycle d'horloge (tick).

Code validé : `if (!rep.success) { n->next_index[peer_idx]--; }`

Ce mécanisme de recul progressif garantit mathématiquement que les logs finiront par converger vers l'état du Leader, comme illustré dans le diagramme de séquence ci-dessous



4.2 Validation de la Machine à États (KV Store)

La transition du journal (Log) vers les données réelles (Hashmap) se fait dans la fonction `apply_log_to_kv`. C'est l'étape finale du consensus qui rend la donnée visible.

Nous utilisons une variable persistante `last_applied` pour suivre l'avancement du traitement :

- Tant que `last_applied < commit_index`, le système "joue" les opérations en attente.
- **Idempotence :** Cette conception permet de rejouer le log depuis le début si nécessaire. C'est un principe fondamental des bases de données modernes (Write-Ahead Logging) : en cas de crash, l'état de la mémoire est reconstruit intégralement en relisant le journal séquentiellement.

5. Déploiement et Configuration

Le déploiement du cluster est conçu pour être simple et reproductible grâce à des fichiers de configuration statiques, permettant une mise en place rapide pour les tests ou la production.

5.1 Évolution de l'Arborescence

Le code source est organisé en trois jalons distincts, chaque dossier représentant une version stable et autonome du système :

- **`step1_ping/` (Infrastructure)** : Version initiale validant uniquement la connectivité TCP maillée (Mesh Ping). Elle sert de fondation réseau.
- **`step2/` (Consensus)** : Ajout de la logique d'élection Raft. Cette version permet de valider la stabilité du leadership sans se soucier encore des données.
- **`step3/` (Version Finale)** : Intégration complète du moteur de réplication de logs, de la machine à états (KV Store) et du client externe.

> **Note :** Pour le déploiement final, seul le dossier `step3` est nécessaire car il contient l'intégralité des fonctionnalités.

5.2 Configuration du Cluster (`nodes.conf`)

Le déploiement repose sur un fichier de configuration statique, ce qui simplifie le lancement sur des environnements variés (local ou réseau réel). Le fichier `nodes.conf` définit la topologie du cluster sous le format : `<ID> <IP> <PORT>`.

Exemple de configuration pour un cluster local de 3 nœuds :

```
0 127.0.0.1 5000

1 127.0.0.1 5001

2 127.0.0.1 5002
```

```
./node <ID> nodes.conf
```

Pour un déploiement réel sur plusieurs machines, il suffit de remplacer `127.0.0.1` par les adresses IP respectives des serveurs.

5.3 Chaîne de Compilation et Lancement

Le projet utilise `Make` pour automatiser la compilation des différents binaires nécessaires au déploiement.

Procédure de déploiement (Version Finale) :

1. **Compilation :** La commande `make` dans le dossier `step3` génère deux exécutables :
    - `./node` : Le démon serveur (le nœud Raft).
    - `./client` : L'interface CLI pour interagir avec le cluster.
2. **Lancement Manuel (Mode Production) :** Chaque nœud doit être lancé dans son propre terminal (ou processus) en spécifiant son ID unique :

```
./node 0 nodes.conf
./node 1 nodes.conf
./node 2 nodes.conf
```

Lancement Automatisé (Mode Test/Démonstration) : Nous avons développé un script Shell `test_step3.sh` qui orchestre le déploiement complet : il compile le projet, lance les nœuds en arrière-plan, attend la stabilisation de l'élection, et exécute une batterie de tests clients (PUT/GET) avant d'arrêter proprement le cluster.

6. Scénarios de Test et Validation

Pour valider le fonctionnement global, les tests ont été automatisés via le script `test_step3.sh`. Voici les résultats observés sur un cluster de 3 nœuds.

Scénario 1 : Élection et Stabilité

- **Action :** Démarrage simultané des 3 nœuds.
- **Résultat :** Un nœud passe Candidat (après un timeout aléatoire de 500-1000ms), obtient 2 votes, et devient Leader.
- **Log observé :** `👑 Node X élu leader`.

Scénario 2 : Écriture Distribuée (PUT)

- **Action :** Le client envoie la commande `PUT 10 42` au nœud 0.
- **Flux interne :**
    1. Le Leader ajoute l'entrée à son log.
    2. Le Leader propage l'entrée aux Followers via `AppendEntries`.
    3. À réception du Quorum (2 nœuds), le Leader "Commits" l'entrée et répond "Succès" au client.

Scénario 3 : Cohérence (GET sur Follower)

- **Action :** Le client envoie `GET 10` au nœud 1 (qui est un Follower passif).
- **Résultat :** Le Node 1 répond `42`.
- **Preuve :** Cela démontre que le mécanisme de réplication a correctement propagé la donnée du Leader vers les Followers et que la machine à états est synchronisée sur tout le cluster.

7. Bilan et Perspectives

7.1 Points Forts de l'Implémentation

- **Robustesse Réseau (Non-Bloquant) :** L'implémentation de sockets non-bloquants couplée à la fonction `select()` est un atout majeur. Elle permet au cluster de détecter une panne de nœud en moins de 100ms, là où un socket bloquant standard aurait figé le système pendant plus d'une minute.
- **Sécurité et Concurrence (Thread Safety) :** L'utilisation rigoureuse des Mutex (`pthread_mutex_lock`) protège l'état global du nœud. Cela évite les conditions de course (Race Conditions) fréquentes lorsque le Timer (Thread principal) et le Réseau (Thread worker) tentent de modifier le statut du nœud simultanément.
- **Expérience Client Transparente :** Le système gère les redirections. Un client n'a pas besoin de savoir qui est le Leader ; s'il se trompe, le cluster le redirige vers le bon nœud.

7.2 Limitations Actuelles

Bien que fonctionnel et passant tous les tests, le système comporte des simplifications inhérentes au cadre pédagogique :

- **Absence de Persistance (RAM vs Disque) :** Les logs sont stockés en mémoire vive. Un redémarrage complet du cluster entraîne la perte des données. Pour la production, l'écriture sur disque (WAL - Write Ahead Logging) serait indispensable.
- **Taille du Log :** Le journal grandit indéfiniment. Une implémentation industrielle nécessiterait un mécanisme de "Snapshotting" pour archiver l'état de la base et vider les vieux logs.
- **Configuration Statique :** L'ajout de nœuds nécessite un redémarrage. L'algorithme de changement de configuration dynamique de Raft n'a pas été implémenté.

8. Conclusion

Ce projet de Programmation Système Avancée nous a permis de confronter la théorie élégante des systèmes distribués à la complexité de leur implémentation bas niveau en C.

Nous avons réussi à construire, étape par étape, un système complet capable de transformer un ensemble de machines faillibles en un service de stockage cohérent et unique.

Les défis surmontés : La principale difficulté n'a pas résidé dans la compréhension de l'algorithme Raft, mais dans sa traduction en code asynchrone :

1. **Le Débuggage Distribué :** Comprendre pourquoi un cluster ne converge pas est complexe quand les logs sont répartis sur trois terminaux et que les événements se jouent à la milliseconde.
2. **La Gestion de la Mémoire et des Sockets :** La manipulation des buffers réseaux et la gestion des pointeurs en C, combinées au multithreading, ont nécessité une rigueur absolue pour éviter les fuites de mémoire et les comportements indéfinis (Segfaults).
3. **La Logique de "Backtracking" :** Implémenter correctement la réparation des logs (lorsqu'un Follower est en retard) a été l'un des points les plus délicats de la logique métier.

En conclusion, ce projet valide non seulement l'acquisition des compétences techniques cibles (Sockets, Threads, Concurrence), mais illustre aussi concrètement le théorème CAP : nous avons construit un système qui privilégie la Cohérence et la tolérance aux Partitions, au prix d'une complexité logicielle élevée.

Test du projet

Information	Détails
Lien du Dépôt GitLab	https://gitlabinfo.iutmontp.univ-montp2.fr/pujoln/cluster-de-noeuds-g3
Accès Enseignant	Invitation envoyée sur votre adresse **LIRM**.
Branche `main`	Contient la **version finale (`step3`)** ainsi que le fichier **`README.md`** détaillant la procédure de déploiement.