

Nama : Erizki Fadli

NIM : 11221014

Kelas : SisTer B

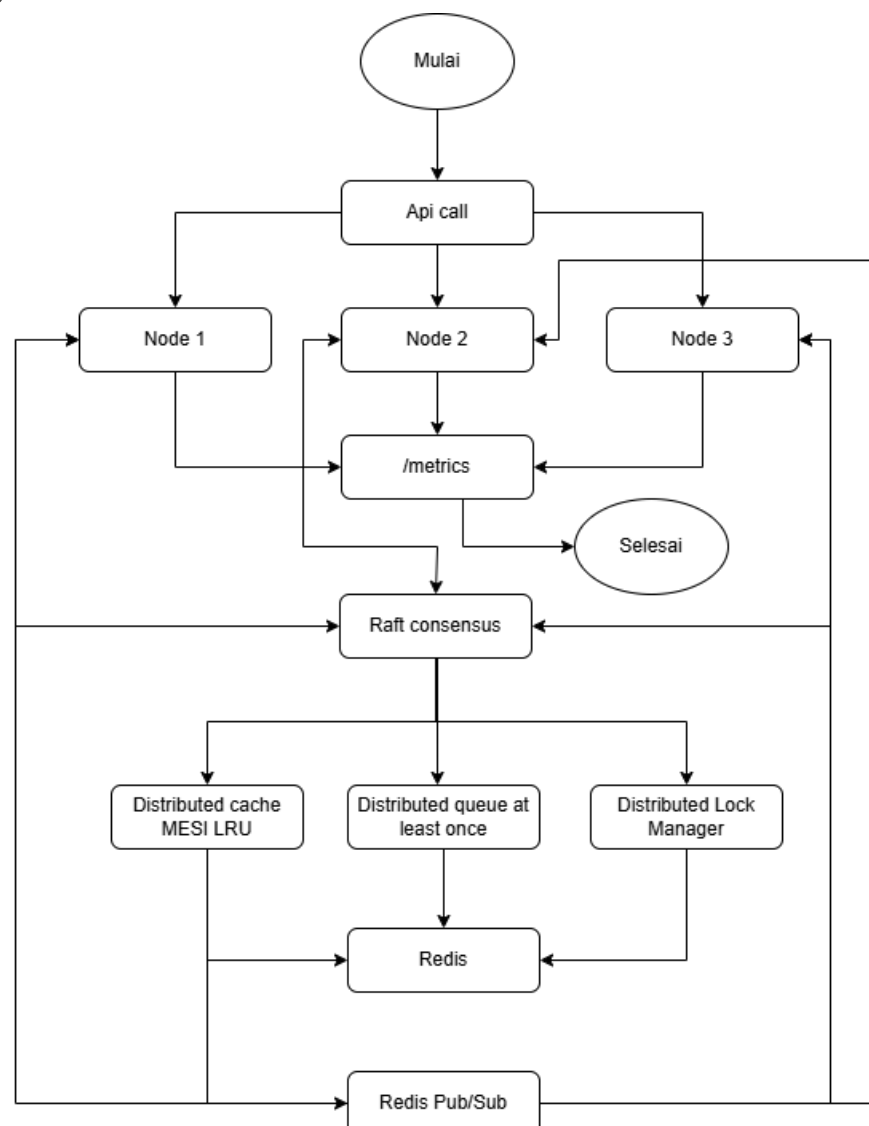
Tugas 2

A. Technical Decomentation

1. Arsitektur sistem lengkap dengan diagram

Sistem terdiri dari beberapa service node dengan tech stack Python/asyncio + aiohttp yang menjalankan tiga fungsi yakni Distributed Lock Manager, Distributed Queue, dan Distributed Cache. Koordinasi ketiganya dilakukan dengan Raft yang berfungsi untuk leader election & log replication. Redis dipakai sebagai durable backing store yakni lock table, queue payload, cache store sekaligus kanal Pub/Sub untuk invalidasi cache duplikat. Semua node mengeksport /metrics berformat Prometheus. Sistem dideploy secara 3 node + 1 Redis melalui Docker Compose.

Diagram:



2. Penjelasan algoritma yang digunakan

Raft: Sistem menggunakan Raft untuk memastikan hanya ada satu *leader* aktif dan semua keputusan *grant/release lock*, *enqueue/ack*, invalidasi *cache* tercatat konsisten. Tiap node menyimpan *log entries* bertingkat *term*; pemimpin dipilih melalui pemungutan suara ketika *election timeout* acak terpicu pada *follower* yang tidak menerima *heartbeat*. Pemimpin yang terpilih menduplikat *AppendEntries* ke mayoritas; sebuah entri dianggap *committed* saat sudah tersalin ke kuorum. Indeks *commit_index* dan *last_applied* menjamin urutan eksekusi deterministik pada *state machine*. Mekanisme *log matching property* (pencocokan (*term, index*)) dan *roll-back* memastikan divergensi log dipangkas aman. Pendekatan ini memberikan keamanan data selama mayoritas node tersedia.

Failure detector (heartbeat + timeout). Di luar Raft, sistem menjalankan pendeteksi kegagalan berbasis *heartbeat* periodik. Node memelihara *last_seen* untuk tiap rekan; bila jeda melebihi ambang waktu, rekan dianggap *suspect* dan rute RPC ke rekan tersebut dihindari. Pemilihan parameter *heartbeat interval* dan *timeout* dibuat konservatif agar tidak memicu *false positive* saat beban tinggi, sekaligus cukup responsif saat benar-benar terjadi kegagalan.

Distributed Lock Manager (shared/exclusive + deadlock detection). Penerapan *lock* membolehkan beberapa *shared* bersamaan, sedangkan *exclusive* tunggal memblokir yang lain. Permintaan yang bertentangan masuk antrian teratur; keputusan *grant* dan *release* dibuat lewat *log* Raft agar konsisten lintas node. Untuk mencegah *deadlock*, sistem membangun *waits-for graph* terdistribusi: setiap klien yang menunggu *lock X* menambahkan sisi dari klien pemegang *lock X* menuju dirinya. Deteksi siklus dilakukan dengan DFS/Tarjan pada graf tersebut; jika ditemukan, dipilih *victim* (misalnya permintaan termuda) untuk dibatalkan, dan antrian dievaluasi ulang.

Distributed Queue (at-least-once dengan visibility timeout). Penerbitan pesan menyimpan *payload* pada *hash* (*queue:msg:{id}*) dan *id* pesan ke *list* topik (*queue:{topic}*). Consume memindahkan *id* ke *ZSET inflight* (*queue:{topic}:{group}:inflight*) dengan skor *expire_at = now + timeout* lalu mengembalikan *{msg_id, value}*. *ACK* menghapus *id* dari *inflight* dan *payload* dari *hash*. Sebuah *scavenger loop* memindai *inflight* yang kadaluarsa untuk *requeue* ke *list*, sehingga bila consumer mati, pesan tidak hilang. Inilah yang memberikan jaminan at-least-once. sisi konsumen dibuat idempoten untuk menangani duplikasi. Untuk penyebaran beban yang stabil, *routing key* dipetakan ke partisi menggunakan consistent hashing sehingga penambahan/pengurangan node hanya memindahkan sebagian kecil *key*.

Cache coherence (MESI-like invalidation) dan LRU. Penulisan PUT bersifat *write-through* ke penyimpanan redis lalu menerbitkan pesan invalidasi pada Pub/Sub (`cache:inval`) berisi key dan *version*. Node lain yang menerima invalidasi menandai entri lokal menjadi *Invalid* (I) atau membuangnya; GET berikutnya yang *miss* akan *fetch* dari penyimpanan dasar dan menyimpannya ke cache lokal sebagai *Shared* (S). LRU menjaga ukuran cache lokal: setiap akses memindahkan entri ke *most-recently used*; saat melewati ambang kapasitas, entri *least-recently used* dikeluarkan. Kombinasi invalidasi + *fetch-on-read* menjaga data tetap baru dengan latensi baca yang rendah.

Pengumpulan metrik (instrumentation). Semua jalur kritis mengeksport *metric* Prometheus: peran Raft, jumlah *AppendEntries/RequestVote*, *lock acquire/release*, *queue publish/consume/redeliver/inflight*, serta *cache put/get/invalidate* dan ukuran cache. Pengukuran ini tidak hanya memudahkan *benchmark* (throughput/latency), tetapi juga menjadi *feedback loop* untuk mengukur *timeout*, ukuran batch, dan kapasitas cache agar sistem tetap stabil di bawah variasi beban.

3. API documentation

- Import file Postman Collection dan Environment dari folder postman/
- Set nilai environment:
baseUrlNode1 = http://localhost:8101
baseUrlNode2 = http://localhost:8102
baseUrlNode3 = http://localhost:8103
timeoutMs = 1000

Jalankan request sesuai folder:

- Raft: GET /raft/status untuk melihat node LEADER/FOLLOWER.
- Lock Manager:
POST /lock/acquire (mode exclusive atau shared)
POST /lock/release
- Queue:
POST /queue/publish
POST /queue/consume (ambil msg_id)
POST /queue/ack (acknowledge msg_id)
- Cache (MESI):
POST /cache/put
GET /cache/get?key=...

4. Deployment guide

- Requirement: python 3.11+, docker, locust, dan postman
- Clone project
git clone
<https://github.com/Erizki0712/distributed-sync.git>
cd distributed-sync

- Buat virtual environment
python -m venv .venv
Windows
.\.venv\Scripts\activate
Linux/Mac
source .venv/bin/activate
- Install requirement
pip install -r requirements.txt
- Build dan jalankan docker
cd docker
docker compose up --build (3 node)
docker compose -f docker-compose.single.yml -p dss-single
up --build -d (Single node)

B. Performance Analysis Report

1. Benchmarking hasil dengan berbagai skenario

Benchmark dilakukan menggunakan locust dengan cara melakukan pengujian beban pada versi 1 dan 3 node. Workload locust berisi publish/consume/ack untuk queue, put/get untuk cache, dan acquire/release untuk lock dengan skenario 200 pengguna dalam 60 detik.

2. Analisis throughput, latency, dan scalability

Endpoint	Latency 3 node (ms)	Latency 1 node (ms)	Selisih latency (%)	Req/s 3 node	Req/s 1 node
/cache/get	34	27	20.6	305.8	324.8
/cache/put	45	43	4.4	306	325.1
/lock/acquire	46	35	23.9	293.5	351.4
/lock/release	43	35	18.6	293.3	351.3
/queue/ack	34	33	2.9	253.9	281.8
/queue/consume	45	33	26.7	254	261.1
/queue/publish	40	39	2.5	254.3	261.5

3. Visualisasi performa

Dari tabel terlihat bahwa single node unggul pada throughput dan latency dikarenakan tidak ada quorum/replication hop sedangkan 3 node lebih lambat karena harus forward event ke leader untuk setiap proses.