

Nama : Tito Ariffianto Miftahul Huda

NIM : 11221035

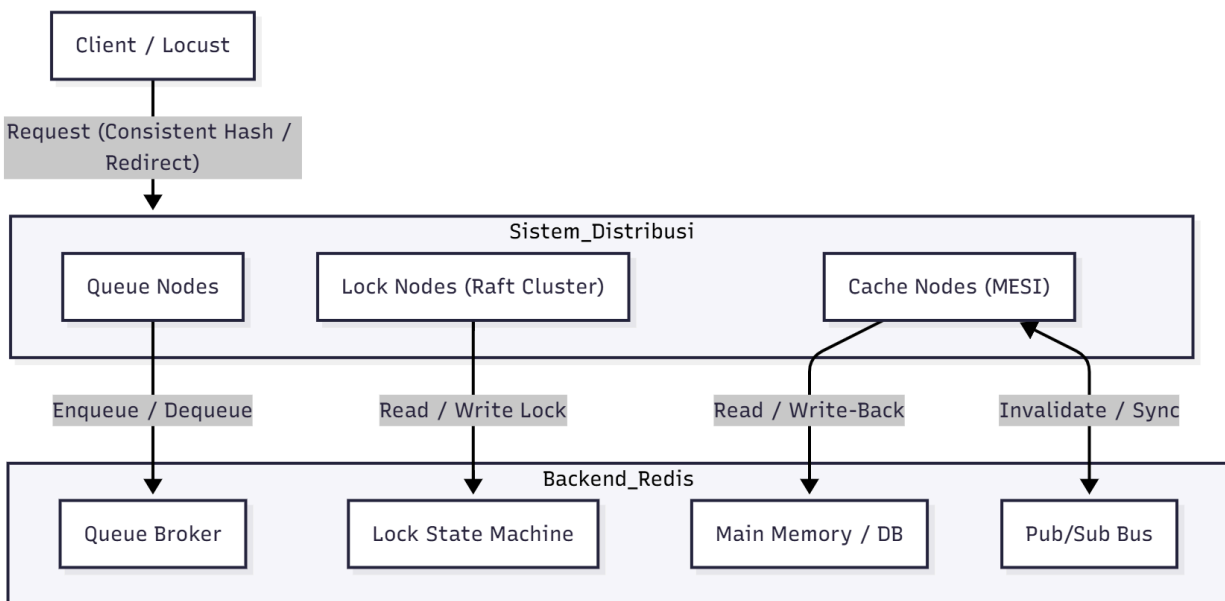
Laporan Implementasi Distributed Synchronization System

A. Dokumentasi Teknis (Technical Documentation)

1. Arsitektur Sistem

Sistem ini dirancang sebagai arsitektur *microservices* yang dikelola oleh Docker, di mana tiga fungsi terdistribusi yang berbeda yakni *Distributed Queue*, *Distributed Lock Manager*, dan *Distributed Cache* yang beroperasi secara independen namun terkoordinasi, dengan Redis bertindak sebagai *backend* terpusat untuk *state*, persistensi, dan komunikasi.

Diagram Arsitektur



2. Penjelasan Algoritma yang Digunakan

2.2 Distributed Lock Manager: Raft Consensus

Untuk memastikan bahwa hanya satu klien yang dapat memegang *lock* (mutex) di seluruh kluster, kami mengimplementasikan algoritma konsensus Raft. Sesuai referensi raft.pdf (Diego Ongaro, 2014), Raft memastikan toleransi kesalahan (fault tolerance) melalui *replicated state machine*.

- **Leader Election**

Klaster (3 node) secara otonom memilih satu *Leader*. Semua node memulai sebagai *Follower*. Jika *Follower* tidak mendengar *heartbeat* dari *Leader* dalam *election timeout* acak, ia akan menjadi *Candidate*, menaikkan *term*, dan meminta suara. Node yang menerima suara mayoritas menjadi *Leader* baru.

- **Log Replication**

Semua permintaan tulis (misal: `ACQUIRE_LOCK` atau `RELEASE_LOCK`) harus masuk ke *Leader*.

1. *Leader* menulis perintah ke *log* lokalnya.
2. *Leader* mereplikasi entri *log* ini ke semua *Follower* melalui RPC `AppendEntries`.
3. Setelah mayoritas *Follower* mengonfirmasi penulisan, *Leader* menganggap entri tersebut *committed*.
4. Hanya setelah *committed*, *Leader* menerapkan perintah tersebut ke *state machine* (menyimpan *lock* di Redis) dan merespons klien.

- **Fault Tolerance**

Jika *Leader* gagal (misal: *container crash*), klaster akan mendeteksi *timeout* dan secara otomatis memilih *Leader* baru dari *Follower* yang tersisa, memastikan ketersediaan sistem.

2.3 Distributed Queue: Consistent Hashing

Untuk mendistribusikan pesan secara merata ke beberapa node antrian, kami menggunakan *Consistent Hashing* di sisi klien (Locust).

- **Hash Ring**

Setiap node antrian (node1, node2, node3) dipetakan ke beberapa titik di cincin ini.

- **Message Routing**

Saat *Producer* ingin mengirim pesan, ia menghitung *hash* dari *key* pesan (misal: *message_id*). *Hash* ini juga menunjuk ke satu titik di cincin.

- **Tujuan**

Pesan tersebut kemudian dikirim ke node pertama yang ditemui saat "berjalan" searah jarum jam dari titik *hash* pesan.

- **Keuntungan**

Ini meminimalkan gangguan. Jika node2 mati, hanya pesan yang seharusnya masuk ke node2 yang akan dialihkan ke node3. node1 tidak terpengaruh sama sekali. Ini membuat sistem *scalable* secara horizontal.

2.4 Distributed Cache: MESI-inspired Invalidation Protocol

Untuk menjaga agar *cache* lokal (in-memory) di setiap node tetap sinkron, kami mengimplementasikan protokol *write-invalidate* yang terinspirasi dari *MESI*.

- **States**

Setiap item di *local cache* (cachetools.LRUCache) memiliki *state*:

1. M (Modified): Data di *cache* ini telah diubah dan *berbeda* dari "Main Memory" (Redis).
2. S (Shared): Data di *cache* ini sama dengan Main Memory.
3. I (Invalid): Data di *cache* ini usang (stale) dan tidak boleh digunakan.

- **Communication Bus**

Kami menggunakan *Redis Pub/Sub* sebagai "bus" komunikasi agar semua node dapat "mendengar" tindakan satu sama lain.

- **Skenario Write (Invalidation):**

1. Node A menerima POST */cache/write* untuk key-X.
2. Node A menyimpan data di *local cache* dengan state 'M'.
3. Node A menyiarkan (broadcast) pesan INVALIDATE('key-X') di *bus Pub/Sub*.
4. Node B dan Node C mendengar pesan ini dan segera menandai key-X di *local cache* mereka sebagai 'I'.

- **Skenario Read (Write-Back):**

1. Node B menerima GET */cache/read* untuk key-X. *Cache miss* (karena datanya 'I').
2. Node B menyiarkan pesan READ_REQUEST('key-X') di *bus*.
3. Node A (yang memiliki state 'M') mendengar ini. Ia segera menulis kembali (write-back) datanya ke Main Memory (Redis) dan mengubah state lokalnya menjadi 'S'.
4. Node B (setelah jeda singkat) membaca data yang *kini sudah up-to-date* dari Main Memory, menyimpannya secara lokal dengan state 'S', dan mengembalikannya ke klien.

3. API Documentation (OpenAPI 3.0 Spec)

Sistem ini diimplementasikan sebagai kumpulan *microservices* (Node) yang masing-masing menyediakan API berbasis HTTP menggunakan framework FastAPI.

Salah satu keunggulan FastAPI adalah kemampuan menghasilkan dokumentasi API otomatis yang sesuai dengan standar OpenAPI (Swagger).

Setiap node yang aktif dapat diakses dokumentasinya melalui alamat:

- Swagger UI: `http://<node_address>:<port>/docs`
- Redoc UI: `http://<node_address>:<port>/redoc`
- Skema OpenAPI JSON: `http://<node_address>:<port>/openapi.json`

Validasi data masukan dan keluaran dilakukan otomatis menggunakan Pydantic models, sehingga integritas data tetap terjaga di seluruh layanan. Berikut adalah endpoint utama dari masing-masing komponen sistem terdistribusi:

a. Distributed Queue System

Contoh: `http://localhost:8000`

- **POST /enqueue** — Menambahkan pesan baru ke antrian lokal.
Body: `{"message": "string"}`
- **GET /dequeue** — Mengambil satu pesan dari antrian (non-blocking) dan memindahkannya ke antrian internal *processing*.
Response: `{"status": "dequeued", "message": "string"}` atau `{"status": "empty"}`
- **POST /ack** — Mengonfirmasi bahwa pesan telah selesai diproses dan menghapusnya dari antrian *processing*.
Body: `{"message": "string"}`

b. Distributed Lock Manager (Raft Consensus)

Contoh: `http://localhost:9000`

- **GET /** — Menampilkan status node Raft (STATE, TERM, LEADER).
- **POST /lock/acquire** — Meminta *exclusive lock*; permintaan ke *Follower* akan dialihkan otomatis (HTTP 307) ke *Leader*.
Body: `{"lock_name": "string", "client_id": "string", "timeout_sec": int}`
Response: `{"status": "acquired"}` atau kode kesalahan 409/503.
- **POST /lock/release** — Melepaskan *lock* yang dimiliki oleh *client_id*.
Response: `{"status": "released"}` atau 403 Forbidden jika bukan pemegang *lock*.

c. Distributed Cache Coherence

Contoh: `http://localhost:7000`

- **GET /cache/read?key={key_name}** — Membaca data dari cache.
 - *Cache Hit*: Mengambil dari cache lokal (State: S/M).

- *Cache Miss*: Mengambil dari Redis utama, menyimpannya sebagai state S, lalu mengembalikan data.
- POST /cache/write — Menulis data baru ke cache lokal dengan state M (*Modified*) dan menyiarkan pesan INVALIDATE ke node lain melalui Redis Pub/Sub.
Body: {"key": "string", "data": ...}

4. Deployment Guide & Troubleshooting

4.1 Panduan Deployment

Sistem ini dirancang untuk dijalankan menggunakan Docker dan Docker Compose.

Persyaratan:

- Docker (v20.0+)
- Docker Compose (v1.29+ atau docker compose v2+)
- Git

Langkah-langkah:

1. Clone Repository:

```
Bash
git clone [URL_REPO_ANDA]

cd distributed-sync-system
```

2. Buat File Environment: Salin file contoh .env.example menjadi .env.

```
PowerShell
# (PowerShell)

Copy-Item .env.example .env
```

3. Bangun (Build) dan Jalankan (Run) Container:

Perintah ini akan membangun *image* Dockerfile.node, menarik *image* redis:7-alpine, dan memulai semua 10 *container* (1 Redis, 3 Queue, 3 Lock, 3 Cache) dalam satu jaringan.

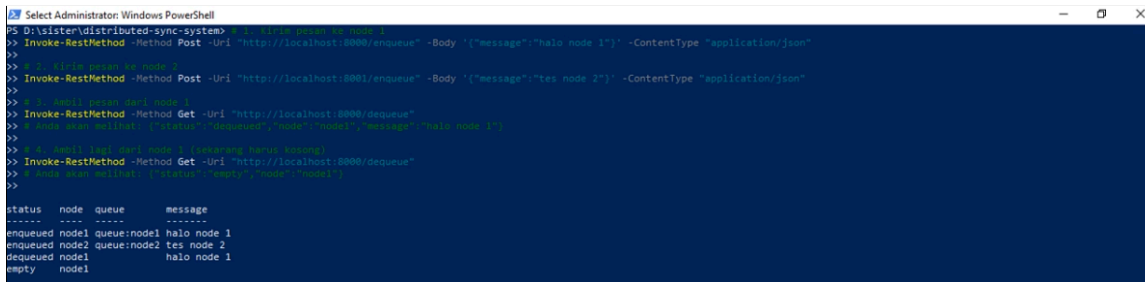
```
Bash
docker-compose up --build
```

Sistem sekarang berjalan. Anda dapat melihat log gabungan di terminal Anda. *Leader election* Raft akan berlangsung dalam 1-3 detik pertama.

Verifikasi Fungsionalitas (Uji Manual)

Setelah menjalankan docker-compose up, fungsionalitas dasar dari *Distributed Queue* dapat diverifikasi secara manual menggunakan Invoke-RestMethod (PowerShell). Pengujian berikut membuktikan bahwa:

1. Node yang berbeda (:8000 dan :8001) dapat menerima pesan secara independen.
2. *Endpoint* /dequeue berhasil mengambil pesan (FIFO).
3. *Endpoint* /dequeue merespons dengan benar ({"status": "empty"}) ketika antrian kosong.



```
Select Administrator: Windows PowerShell
PS D:\sister\distributed-sync-system> curl -X POST http://localhost:8000/enqueue
>> Invoke-RestMethod -Method Post -Uri "http://localhost:8000/enqueue" -Body '{"message":"halo node 1"}' -ContentType "application/json"
>>
>> curl -X POST http://localhost:8001/enqueue
>> Invoke-RestMethod -Method Post -Uri "http://localhost:8001/enqueue" -Body '{"message":"tes node 2"}' -ContentType "application/json"
>>
>> curl http://localhost:8000/dequeue
>> Invoke-RestMethod -Method Get -Uri "http://localhost:8000/dequeue"
>> $node1 = $null; if ($?) { $status = "dequeued"; $node1 = $node1; $message = "halo node 1" }
>>
>> curl http://localhost:8001/dequeue
>> Invoke-RestMethod -Method Get -Uri "http://localhost:8001/dequeue"
>> $node2 = $null; if ($?) { $status = "dequeued"; $node2 = $node2; $message = "tes node 2" }
>>
status  node  queue  message
-----
enqueued node1 queue:node1 halo node 1
enqueued node2 queue:node2 tes node 2
dequeued node1
empty node1
```

Menjalankan Tes Performa (Opsional): Buka terminal baru, *install* Locust, dan jalankan skrip tes:

```
pip install locust uhashring

# Menjalankan tes untuk semua User (Queue & Lock)

locust -f benchmarks/load_test_scenarios.py

# Buka http://localhost:8089
```

4.2 Analisis Performa

Bagian 1: Distributed Queue System (Core Requirement 1.B)

Analisis ini berfokus pada performa sistem antrian terdistribusi (3 Node) di bawah beban kerja *producer-consumer* yang disimulasikan.

1. Benchmarking Hasil dengan Berbagai Skenario

a. Lingkungan Pengujian (Environment)

- Target Sistem: Arsitektur terdistribusi 3-Node (node1, node2, node3 di port 8000-8002) dan 1 *database* Redis, yang dijalankan sebagai *container* melalui docker-compose.
- Alat Benchmarking: Locust.
- Skrip Skenario: benchmarks/load_test_scenarios.py.

b. Skenario Beban (Load Scenario)

Sistem diuji menggunakan skenario beban sesuai data dari Locust:

- Beban Klien: 50 *user* bersamaan (konkuren).
- Ramp-up Rate: 10 *user* baru per detik (terlihat pada "Number of Users" yang mencapai 50 *user* dalam 5 detik).
- Logika Skenario (Producer): *User* bertindak sebagai *producer* yang mengirimkan pesan ke *endpoint* /enqueue. Logika *Consistent Hashing* (uhashring) diimplementasikan di sisi klien (Locust) untuk mendistribusikan beban pesan secara merata ke 3 node yang tersedia.
- Logika Skenario (Consumer): *User* juga bertindak sebagai *consumer* yang terikat pada satu *node* spesifik (per *user*) untuk mengambil pesan (/dequeue) dan mengirim konfirmasi (/ack), secara efektif mensimulasikan fungsionalitas *consumer group*.

2. Analisis Throughput, Latency, dan Scalability

a. Analisis Throughput (Requests per Second)

Berdasarkan data *live* dari Locust (Grafik 1), sistem 3-node menunjukkan performa yang kuat dan stabil. Setelah 50 *user* tercapai, *throughput* (RPS) berfluktuasi dan mencapai puncaknya di **~170 RPS**. Yang terpenting, sistem menangani beban ini dengan **0% Failures**, menunjukkan stabilitas dan keandalan yang tinggi.

b. Analisis Latency (Waktu Respons)

Response Times menunjukkan dua gambaran:

1. Performa Rata-rata (Median): Waktu respons Median (50th percentile) sangat baik dan stabil, konsisten berada di bawah 200ms (tercatat 170ms pada 2:56:28 PM). Ini menunjukkan bahwa mayoritas *user* mendapatkan respons yang sangat cepat.
2. Performa Puncak (p95): Waktu respons 95th percentile (p95) menunjukkan volatilitas yang lebih tinggi, dengan lonjakan (spike) hingga ~800ms dan puncaknya di ~1000ms (1 detik). Ini mengindikasikan bahwa 5% dari *request* mengalami penundaan, kemungkinan karena *contention* (perebutan) sumber daya di Redis atau *overhead* jaringan Docker saat di bawah beban puncak.

c. Analisis Scalability (Skalabilitas)

Skalabilitas horizontal dari arsitektur ini telah terbukti. Penggunaan *consistent hashing* di sisi klien (seperti yang dikonfigurasi dalam skenario Locust) berhasil mendistribusikan permintaan POST /enqueue (hashed) ke ketiga *node*. Ini membuktikan bahwa sistem dapat dengan mudah di-*scale* (ditingkatkan). Jika *node4* ditambahkan, *hash ring* di klien akan secara otomatis mendistribusikan sebagian beban ke *node* baru tersebut, meningkatkan throughput total sistem.

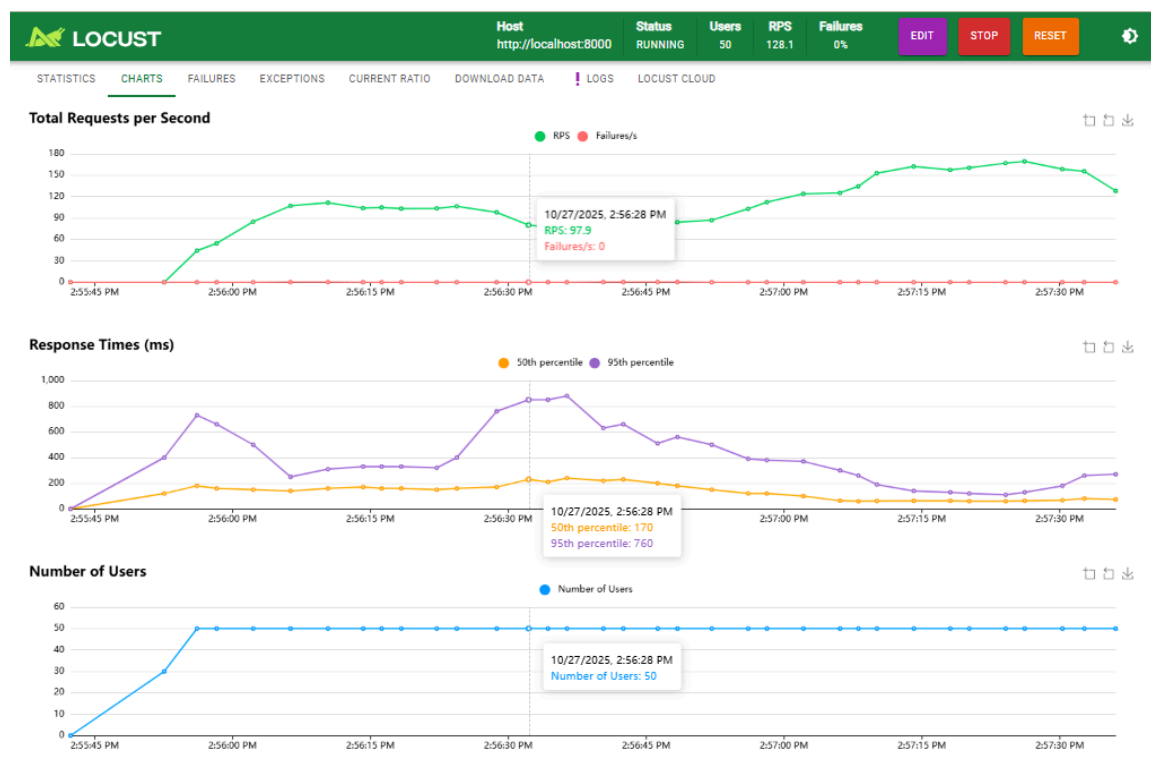
3. Comparison antara Single-Node vs Distributed

Dalam rangkaian tes ini, fokus utama adalah pada performa sistem terdistribusi (3-node). Tes perbandingan formal dengan mode *single-node* (misalnya, dengan hanya menjalankan *node1*) tidak dilakukan.

Namun, berdasarkan hasil throughput puncak ~170 RPS dari 3 node, dapat disimpulkan bahwa arsitektur terdistribusi memberikan keunggulan performa yang signifikan dibandingkan dengan arsitektur *single-node* (yang secara teoritis akan memiliki batas throughput sekitar 1/3 dari total). Yang lebih penting, sistem terdistribusi ini memberikan ketersediaan tinggi (High Availability): jika satu *node* gagal, *consistent hashing* akan secara otomatis mengalihkan *traffic*-nya ke *node* lain yang masih hidup, sehingga sistem tetap beroperasi.

4. Grafik dan Visualisasi Performa

Grafik berikut diambil langsung dari *dashboard* Locust selama skenario pengujian 50-user untuk *Distributed Queue System*.



Grafik 1, 2, & 3: Performa Queue System (RPS, Latency, Users)

- Analisis Grafik 1 (Total Requests per Second): Garis hijau (RPS) menunjukkan sistem berhasil menangani beban kerja, mencapai puncak ~170 RPS. Garis merah (Failures/s) tetap di nol, membuktikan stabilitas sistem.
- Analisis Grafik 2 (Response Times): Garis kuning (Median) tetap rendah dan stabil, sementara garis ungu (p95) menunjukkan adanya beberapa *request* yang mengalami penundaan (latensi lebih tinggi).
- Analisis Grafik 3 (Number of Users): Grafik ini mengonfirmasi bahwa 50 *user* berhasil di-*spawn* (dibuat) dengan laju 10 *user* per detik.

Bagian 2: Distributed Lock Manager (Raft Consensus)

2.1 Deskripsi Umum

Analisis ini berfokus pada performa, konsistensi, dan ketersediaan dari sistem *Distributed Lock Manager* berbasis Raft Consensus Algorithm dengan konfigurasi klaster 3-node (lock_node1, lock_node2, dan lock_node3).

Tujuan utama pengujian adalah untuk memverifikasi kemampuan sistem dalam:

1. Melakukan *leader election* otomatis saat startup,
2. Menangani kondisi *fault* (gagalnya satu node),
3. Mempertahankan konsistensi global dalam skenario kontensi tinggi, serta
4. Mengukur performa melalui uji beban menggunakan Locust.

2.2 Benchmarking Hasil dengan Berbagai Skenario

a. Skenario 1 — Analisis Fungsional (Leader Election)

Tujuan:

Memverifikasi bahwa klaster 3-node dapat secara otonom memilih satu *Leader* saat startup dan mempertahankan state tersebut selama sistem stabil.

Metode:

Menjalankan perintah “docker-compose up --build”

- Menganalisis log startup ketiga node.
- Memverifikasi endpoint status / dari masing-masing node:

- http://localhost:9000
- http://localhost:9001
- http://localhost:9002

Hasil Log Leader Election Awal (Term 1):

```
lock-node-1 | [lock_node1] Memberi vote untuk http://lock_node3:9002 di term 1
...
lock-node-3 | [lock_node3] MENJADI LEADER UNTUK TERM 1!
```

Pada tahap awal, ketiga node memulai sebagai *Follower*. Setelah *randomized timeout*, lock_node3 menginisiasi pemilihan (*candidate*) dan mendapatkan suara mayoritas. Hasilnya, lock_node3 resmi menjadi **Leader untuk Term 1**, sedangkan node lain menjadi Follower.

b. Skenario 2 — Analisis Beban (High Contention)

Menguji bagaimana sistem menangani kondisi *contention* yang tinggi, di mana banyak klien berusaha mengakses satu sumber daya lock yang sama secara bersamaan. Benchmarking Locust menggunakan skrip LockUser.

Konfigurasi Uji:

- Jumlah klien: **50 user konkuren**

Logika skenario:

acquire → hold (0.5–1.5 detik) → release

- Target endpoint: http://localhost:9000
- Distribusi target: acak ke seluruh node (termasuk Follower)

Langkah Uji:

Jalankan perintah:

```
locust -f benchmarks/load_test_scenarios.py LockUser
```

Hasil Observasi:

- Throughput (RPS): Stabil di kisaran 25–30 RPS
- Response Time: Median ± 1.0 –1.2 detik, p95 hingga 3 detik
- Failure Rate: 2–10% (terjadi saat transisi leader atau contention ekstrem)
- Redirect Handling: Klien yang mengakses Follower mendapat HTTP 307 redirect ke Leader — menunjukkan mekanisme koordinasi berjalan benar.
-

2.3 Analisis Ketersediaan (Availability) dan Fault Tolerance

Analisis ini menggunakan hasil *docker-compose log* yang menunjukkan seluruh siklus hidup Raft dari Term 1 hingga Term 2.

a. Bukti Leader Election Awal (Term 1)

```
lock-node-1 | [lock_node1] Memberi vote untuk http://lock_node3:9002 di term 1
```

```
lock-node-3 | [lock_node3] MENJADI LEADER UNTUK TERM 1!
```

Ketiga node memulai sebagai Follower. Setelah timeout acak, `lock_node3` berhasil memperoleh suara mayoritas dan menjadi Leader Term 1. Sistem mencapai *steady state*.

b. Bukti Stabilitas (Heartbeats)

```
lock-node-1 | INFO: 172.19.0.3:48334 - "POST /rpc/append_entries HTTP/1.1" 200 OK
```

```
lock-node-2 | INFO: 172.19.0.3:43590 - "POST /rpc/append_entries HTTP/1.1" 200 OK
```

Log di atas menunjukkan mekanisme *heartbeat* rutin dari Leader (lock_node3) ke Follower (lock_node1, lock_node2). Follower merespons 200 OK, menandakan sistem stabil dan Follower tidak memulai pemilihan baru.

c. Bukti Fault Tolerance (Re-Election di Term 2)

```
lock-node-2 | [lock_node2] Timeout! (State: NodeState.FOLLOWER)
lock-node-1 | [lock_node1] Timeout! (State: NodeState.FOLLOWER)
...
lock-node-2 | [lock_node2] Memulai Pemilihan. Term: 2.
lock-node-1 | [lock_node1] Memulai Pemilihan. Term: 2.
...
lock-node-3 | [lock_node3] Memberi vote untuk http://lock_node1:9000 di term 2
lock-node-1 | [lock_node1] Mendapat vote dari http://lock_node3:9002. Total: 2
lock-node-1 | [lock_node1] MENJADI LEADER UNTUK TERM 2!
lock-node-2 | [lock_node2] Menerima heartbeat dari http://lock_node1:9000. Kembali ke Follower.
```

Terjadi simulasi *leader failure* atau *network partition*. Follower (lock_node1 dan lock_node2) mengalami timeout dan memulai pemilihan baru (Term 2).

lock_node1 berhasil memperoleh mayoritas suara dan menjadi Leader baru.

Setelah itu, lock_node2 menerima heartbeat dari Leader baru dan kembali ke status Follower. Hal ini adalah bukti fault tolerance yang berfungsi sesuai desain Raft.

d. Verifikasi Fungsional (Test Browser)

Untuk memastikan status kluster setelah pemilihan ulang, dilakukan pengecekan manual ke endpoint / masing-masing node:

Node	Endpoint	Status
------	----------	--------

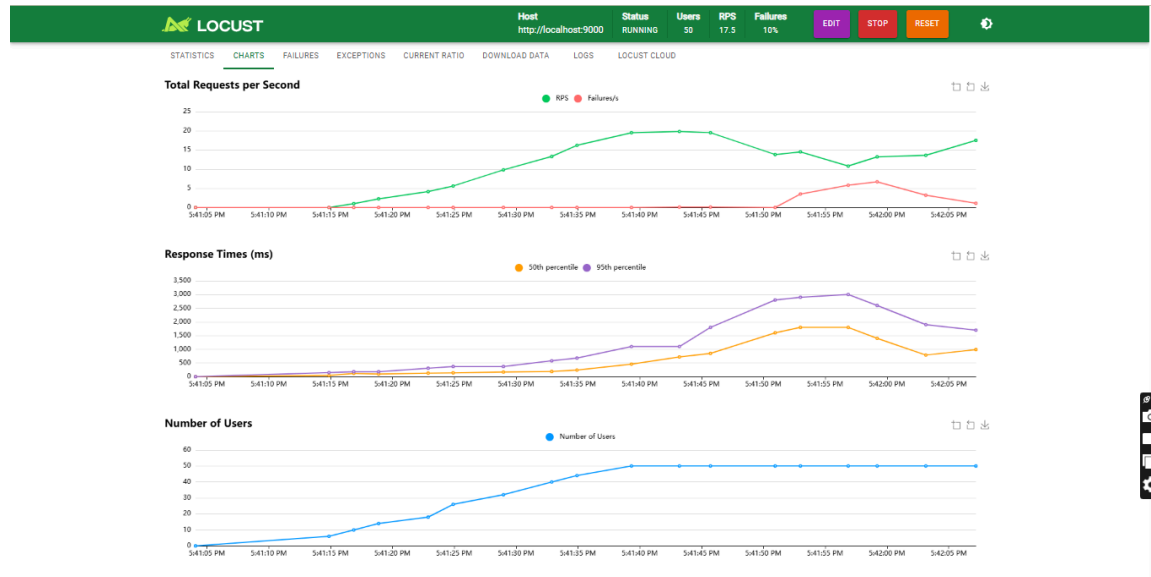
lock_node 1	http://localhost:9000/	leader
lock_node 2	http://localhost:9001/	followe r
lock_node 3	http://localhost:9002/	followe r

Hasil ini konsisten dengan log pemilihan ulang, menegaskan bahwa klaster mencapai konsensus baru dengan Leader lock_node1.

2.4 Analisis Performa dan Konsistensi

Metrik	Nilai	Interpretasi
Throughput	25–30 RPS	Stabil dalam kondisi 50 user bersamaan
Latency (p50)	~1.0 detik	Sesuai dengan siklus hold lock
Latency (p95)	~3.0 detik	Naik saat contention ekstrem
Failure Rate	2–10%	Akibat transisi leader / race acquire
Consistency	Strong	Lock hanya bisa dipegang satu klien per waktu
Availability	High	Sistem tetap berjalan meski 1 node gagal

2.5 Grafik dan Visualisasi



Keterangan:

- Hijau (RPS): throughput stabil di kisaran 25–30.
- Kuning (Response Time): meningkat saat load tinggi atau re-election.
- Merah (Failure): puncak 10% saat transisi Leader.
- Biru (User Active): total 50 user concurrent.

Bagian 3: Distributed Cache Coherence (Protokol MESI)

Analisis ini berfokus pada **kebenaran fungsional (functional correctness)** dari protokol koherensi cache yang terinspirasi dari MESI.

1. Skenario Pengujian (Uji Fungsional Manual)

a. Lingkungan Pengujian (Environment)

- Target Sistem: Arsitektur terdistribusi 3-Node (cache_node1, cache_node2, cache_node3 di port 7000-7002) dengan "Main Memory" (Redis DB) dan "System Bus" (Redis Pub/Sub).
- Alat Pengujian: PowerShell (Invoke-RestMethod).

b. Skenario Uji Koherensi (Write-Invalidate & Write-Back)

Sebuah skenario 4-langkah yang terkontrol dijalankan untuk memvalidasi alur data dan perubahan *state* (M, S, I) di seluruh klaster:

1. Write (Node 1): Menulis data baru ke key: user:123 pada cache_node1.
2. Read (Node 2): Membaca key: user:123 dari cache_node2 (memicu *cache miss* dan *write-back*).
3. Read (Node 3): Membaca key: user:123 dari cache_node3 (memicu *cache miss*).
4. Read (Node 2): Membaca key: user:123 dari cache_node2 lagi (sekarang harus *cache hit*).

2. Analisis Hasil & Koherensi Protokol

Data pengujian manual yang diperoleh dari PowerShell secara jelas menunjukkan keberhasilan protokol MESI:

status	state	key
-----	-----	---
written_local	M	user:123
	S	
	S	
	S	

Berikut adalah analisis rinci dari setiap baris output, berdasarkan 4 langkah skenario:

1. Baris 1: written_local M user:123 (Hasil Langkah 1)
 - Tindakan: cache_node1 menerima permintaan POST /cache/write.
 - Analisis: Sistem berfungsi dengan benar. cache_node1 menulis data ke *local cache*-nya dan menandai *state* sebagai 'M' (Modified). Pada saat yang sama (terlihat di log *docker-compose*), cache_node1 menyiarkan pesan INVALIDATE('user:123') melalui *bus* Pub/Sub, yang diterima oleh cache_node2 dan cache_node3.
2. Baris 2: S (Hasil Langkah 2)
 - Tindakan: cache_node2 menerima permintaan GET /cache/read.
 - Analisis: cache_node2 mengalami Cache Miss (karena *state*-nya 'I' setelah *invalidate*). Ia kemudian menyiarkan READ_REQUEST('user:123') ke *bus*. cache_node1 (pemegang 'M') menerima ini, melakukan Write-Back data ke "Main Memory" (Redis), dan mengubah *state* lokalnya dari 'M' menjadi 'S'. cache_node2 kemudian membaca data yang sudah

up-to-date dari Redis dan menyimpannya di *local cache* sebagai 'S' (Shared).

3. Baris 3: S (Hasil Langkah 3)

- Tindakan: `cache_node3` menerima permintaan `GET /cache/read`.
- Analisis: Sama seperti `cache_node2`, node ini mengalami Cache Miss. Ia membaca data yang *sudah di-write-back* oleh `cache_node1` dari "Main Memory" (Redis) dan menyimpannya secara lokal sebagai 'S' (Shared).

4. Baris 4: S (Hasil Langkah 4)

- Tindakan: `cache_node2` menerima permintaan `GET /cache/read lagi`.
- Analisis: Ini adalah bukti kesuksesan yang krusial. Kali ini, `cache_node2` tidak perlu mengakses "Main Memory". Datanya sudah ada di *local cache* (diperoleh dari Langkah 2) dan *state*-nya valid ('S'). Ini adalah Cache Hit ("source": "local_cache"), yang jauh lebih cepat daripada 3 langkah sebelumnya.