LAPORAN DOKUMENTASI & ANALISIS KINERJA

“Implementasi Distributed Synchronization System”

Nama : Gusti Muhammad Risandha

NIM : 11221028

1. Pendahuluan

1.1. Latar belakang

Dengan menggunakan sistem terdistribusi, beberapa node dapat bekerja sama untuk mencapai tujuan bersama, seperti menjaga konsistensi data, meningkatkan kinerja, dan memastikan toleransi kegagalan. Sinkronisasi antar-node adalah masalah utama dalam sistem terdistribusi untuk mencegah konflik atau inkonsistensi saat beberapa proses menggunakan sumber daya yang sama secara bersamaan. Oleh karena itu, mekanisme penyinkronan sistem yang didistribusikan harus andal, efektif, dan tahan terhadap berbagai kondisi jaringan.

1.2. Tujuan

Tugas ini bertujuan untuk:

A. Mengimplementasikan sistem sinkronisasi terdistribusi yang menggunakan algoritma Raft Consensus untuk menjaga konsistensi antar-node.

B. Menerapkan Distributed Lock Manager, Distributed Queue, dan Distributed Cache Coherence untuk mendukung skenario sistem nyata.

C. Menguji performa sistem dalam hal throughput, latency, dan scalability pada berbagai kondisi jaringan.

D. Menyediakan dokumentasi teknis dan panduan deployment agar sistem mudah direplikasi dan diuji.

1.3. Ruang lingkup

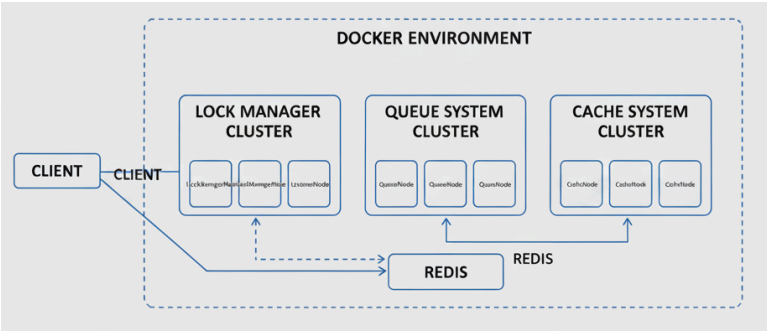
Ruang lingkup tugas ini meliputi:

A. Implementasi mekanisme distributed lock dengan dukungan shared dan exclusive locks.
B. Penanganan deadlock detection dan network partition antar-node.
C. Pengembangan Distributed Queue System dan Cache Coherence Protocol (MESI) secara modular.
D. Penggunaan Redis sebagai penyimpanan state terdistribusi dan Docker untuk containerization.
E. Pengujian performa sistem menggunakan tool seperti Locust untuk analisis beban dan metrik kinerja.

2. Arsitektur Sistem

2.1. Gambaran umum arsitektur

Sistem ini dirancang sebagai Distributed Synchronization System yang terdiri dari beberapa node independen yang saling berkomunikasi untuk menjaga konsistensi status sistem secara global. Setiap node memiliki peran sebagai bagian dari Raft Cluster, di mana salah satu node berfungsi sebagai leader dan node lainnya sebagai followers. Semua operasi sinkronisasi seperti penguncian, antrian, dan pembaruan cache dijalankan melalui mekanisme log replication pada Raft untuk memastikan bahwa seluruh node memiliki status yang sama.



Komponen utama sistem meliputi:

A. BaseNode, menyediakan fungsi inti untuk komunikasi antar-node dan replikasi log berdasarkan algoritma Raft.
B. LockManagerNode, mengelola distributed locks (shared dan exclusive) serta mendeteksi deadlock.
C. QueueNode, menangani antrian pesan dengan consistent hashing untuk mendistribusikan beban antar node.
D. CacheNode, mengimplementasikan cache coherence protocol (MESI) agar data antar-cache tetap sinkron.

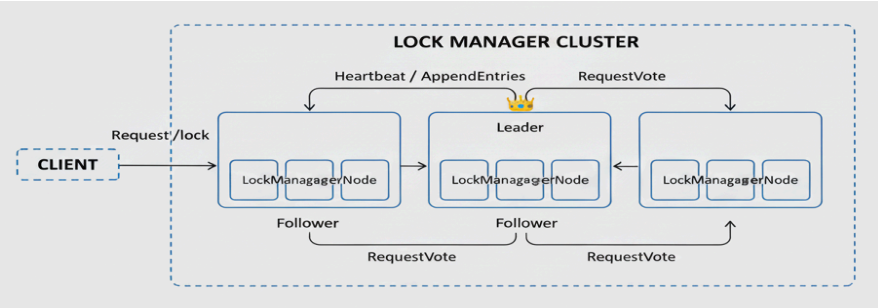
E. Redis, berfungsi sebagai penyimpanan state terdistribusi dan media sinkronisasi antar komponen.

2.2. Mekanisme komunikasi antar node

Komunikasi antar node dilakukan secara asinkron menggunakan pustaka aiohttp atau asyncio, dengan format pesan berbasis JSON. Setiap node menjalankan server HTTP yang menerima permintaan dari node lain untuk operasi seperti AppendEntries (replikasi log), RequestVote (pemilihan leader), dan LockRequest (permintaan kunci). Pesan-pesan ini dikirim melalui saluran komunikasi internal yang dijamin konsistensinya oleh mekanisme timeout dan retry pada protokol Raft.

Setiap node juga mengimplementasikan heartbeat mechanism untuk memantau kesehatan antar-node dan failure detector untuk mendeteksi node yang tidak responsif. Dalam kondisi network partition, sistem tetap konsisten melalui pemilihan ulang leader berdasarkan mayoritas node yang aktif.

Untuk menggambarkan bagaimana proses koordinasi terjadi dalam cluster tertentu, misalnya pada Lock Manager Cluster, diagram berikut menunjukkan interaksi antar-node saat proses pemilihan leader dan replikasi log.



2.3. Deployment architecture

Bagian ini menjelaskan bagaimana seluruh komponen dideploy di lingkungan Docker. Setiap cluster (Lock Manager, Queue, Cache) dijalankan sebagai service terpisah di dalam container yang saling terhubung melalui internal Docker network, sementara Redis bertindak sebagai shared coordination layer. Skema ini memastikan sistem tetap modular, mudah di-scale, dan mampu bertahan saat salah satu node gagal.

3. Implementasi Teknis

3.1. Distributed lock manager

Distributed Lock Manager (DLM) dirancang untuk memastikan konsistensi akses sumber daya dalam sistem terdistribusi. Sistem menggunakan algoritma Raft Consensus untuk menjaga replikasi log antar node, memastikan bahwa setiap perubahan status lock diterapkan secara seragam di semua node.

Mekanisme kunci yang dilakukan meliputi:

A. Pemberian Kunci (Lock Acquisition):
    1) Klien mengirimkan permintaan ACQUIRE_SHARED atau ACQUIRE_EXCLUSIVE ke leader.
    2) Leader menambahkan perintah ke replicated log dan mendistribusikannya ke semua follower.
    3) Setelah mayoritas node mengonfirmasi, kunci dianggap committed dan diterapkan ke state machine.
B. Pelepasan Kunci (Lock Release):
    1) Klien mengirimkan aksi RELEASE ke leader.
    2) Setelah disetujui dan direplikasi, entri dihapus dari pemegang kunci, dan waiters (permintaan yang menunggu) diberi sinyal untuk melanjutkan.
C. Deteksi Deadlock:
    1) Sistem membangun waits-for graph setiap kali klien menunggu kunci.
    2) Jika ditemukan siklus dalam graf, sistem menolak permintaan terbaru untuk mencegah deadlock.

Pengujian dilakukan menggunakan tiga node (node1, node2, node3) yang berkomunikasi melalui socket lokal dan menyimpan state di Redis. Hasil pengujian menunjukkan bahwa mekanisme lock tetap konsisten meskipun terjadi network partition, serta dapat mendukung beberapa proses dengan tipe lock yang berbeda tanpa konflik.

3.2. Distributed queue system

Bagian ini mengimplementasikan sistem antrian terdistribusi menggunakan Python asyncio, aiohttp, dan Redis-ready persistence dengan mekanisme Consistent Hashing. Arsitektur terdiri atas beberapa QueueNode independen yang saling berkomunikasi melalui HTTP endpoint, dan sebuah QueueRouter yang berfungsi untuk mendistribusikan pesan ke node yang sesuai berdasarkan kunci pesan.

Komponen utama yang ada antara lain:

A. QueueRouter (Consistent Hashing), menggunakan library uhashring untuk menentukan node tujuan dari setiap pesan. Pemilihan node berbasis hash ring memastikan distribusi yang seimbang serta konsistensi saat node bertambah atau berkurang.

```
self.ring = uhashring.HashRing(nodes=ring_nodes)
```

Metode get_node_for_message() menentukan node tujuan pesan berdasarkan key, sedangkan get_node_address() mengembalikan alamat node tersebut.

B. QueueNode (Distributed Queue Server)

Setiap node merupakan instance dari kelas QueueNode yang memiliki tiga endpoint utama:

- /produce → menerima pesan dari producer dan menyimpannya ke antrian lokal serta log file.
- /consume → mengirim pesan ke consumer dan menandainya sebagai in-flight.
- /ack → menerima konfirmasi dari consumer dan menghapus pesan dari log.

Semua pesan disimpan dalam file log (queue_logs/{node_id}.log) agar sistem tetap dapat memulihkan data jika terjadi kegagalan node (crash recovery).

Implementasi sistem antrian terdistribusi ini menggunakan *consistent hashing* untuk memastikan pesan dikirim ke node yang konsisten berdasarkan kunci pesan. Dengan pustaka uhashring, sistem dapat menjaga distribusi beban yang stabil dan toleransi kegagalan meskipun terjadi perubahan jumlah node. Setiap node dijalankan menggunakan aiohttp dan asyncio, memungkinkan dukungan multiple producers dan multiple consumers melalui endpoint `/produce`, `/consume`, dan `/ack`.

Untuk menjamin message persistence, setiap pesan disimpan ke dalam file log sebelum dimasukkan ke antrian utama. Saat node direstart, fungsi `_recover_from_log()` membaca ulang log untuk memulihkan pesan yang belum diproses. Proses acknowledgment dari consumer akan menghapus pesan yang telah berhasil diproses dan memperbarui log agar hanya menyimpan pesan aktif. Mekanisme ini mendukung jaminan at-least-once delivery.

Ketahanan sistem diuji melalui simulasi crash dan recovery. Setelah node dihentikan secara paksa, sistem dapat memulihkan semua pesan dari file log, dan setelah acknowledgment, pesan dihapus permanen. Hasil pengujian membuktikan

sistem mampu memulihkan data tanpa kehilangan pesan serta mempertahankan integritas proses distribusi.

3.3. Distributed cache coherence

Arsitektur cache terdistribusi diimplementasikan menggunakan kelas CacheNode. Setiap node berfungsi sebagai cache lokal (in-memory) yang berada di depan sumber kebenaran utama, yaitu database Redis. Tujuannya adalah untuk mengurangi latensi baca dan beban pada Redis dengan melayani permintaan dari cache secepat mungkin. Setiap CacheNode memiliki empat endpoint utama:

- /read/{key} untuk melayani permintaan baca dari klien. Jika data ada di cache lokal (cache hit), data akan langsung dikembalikan. Jika tidak (cache miss), node akan mengambil data dari Redis, menyimpannya di cache lokal, lalu mengembalikannya ke klien.
- /write untuk menerima permintaan tulis dari klien. Node akan memperbarui nilai di Redis terlebih dahulu, kemudian memperbarui cache lokalnya dan mengirimkan pesan invalidasi ke semua node peer.
- /invalidate merupakan endpoint internal yang dipanggil oleh node peer untuk membatalkan (invalidate) entri cache yang sudah usang.
- /metrics akan menyediakan data kinerja seperti jumlah cache hit, miss, dan operasi lainnya.

Untuk menjaga konsistensi data antar cache node, sistem ini menggunakan protokol **write-invalidation** berbasis konsep **MESI (Modified, Exclusive, Shared, Invalid)**. Saat sebuah node melakukan operasi tulis, data ditandai sebagai *Modified* dan node tersebut mengirim permintaan `/invalidate` ke seluruh node peer, sehingga entri pada cache lain berubah menjadi *Invalid*. Akibatnya, saat node lain membaca data tersebut, mereka mengalami *cache miss* dan mengambil versi terbaru dari Redis. Setiap CacheNode juga menerapkan kebijakan *Least Recently Used (LRU)* dengan **OrderedDict** untuk menghapus data yang paling lama tidak diakses. Pengujian menunjukkan bahwa pembaruan pada satu node berhasil memicu invalidasi di node lain, memastikan konsistensi dan mencegah pembacaan data basi (*stale data*).

4. API Documentation

4.1. Daftar endpoint

Dokumentasi API ini mencakup tiga komponen utama dalam sistem terdistribusi: Lock Manager, Queue, dan Cache.

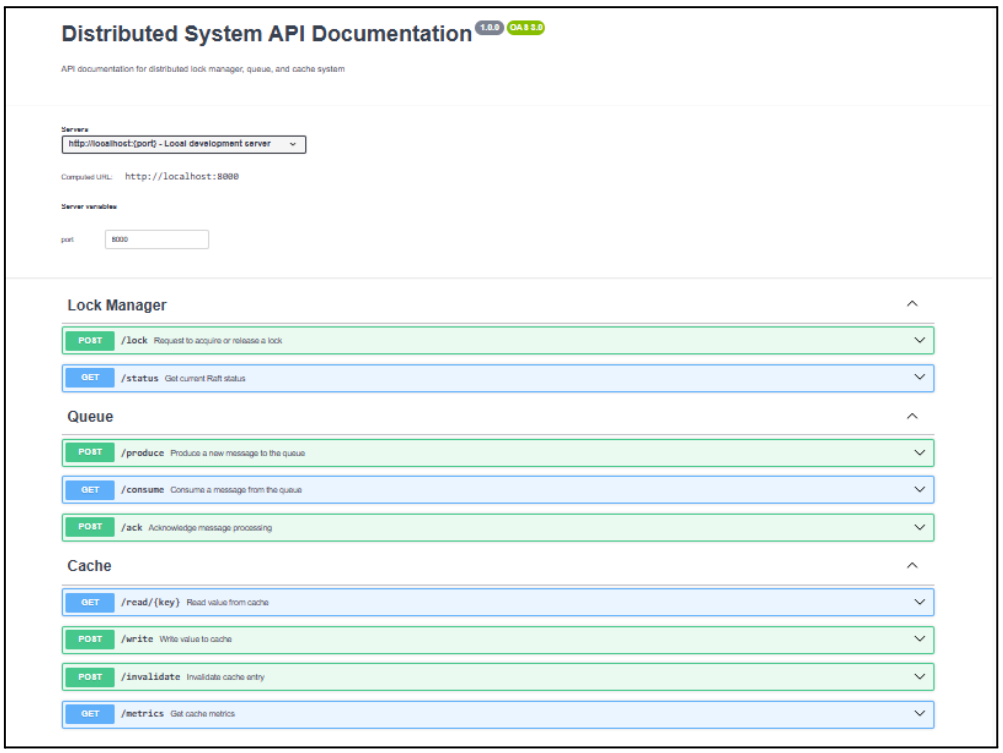
A. Lock Manager, untuk mengelola distributed locks yang mendukung shared dan exclusive mode dengan mekanisme konsensus Raft. Pada komponen

Lock Manager, terdapat dua endpoint utama: /lock serta /status dengan metode GET.

- POST /lock yang menggunakan metode POST untuk melakukan permintaan akuisisi atau pelepasan kunci
- GET /status untuk menampilkan status terkini dari konsensus Raft antar node.

B. Queue, Menyediakan message queue terdistribusi yang mendukung produksi, konsumsi, dan acknowledgement pesan. Komponen Queue memiliki tiga endpoint, yakni:
   - POST /produce untuk menambahkan pesan baru ke dalam queue.
   - GET /consume untuk mengambil pesan dari queue untuk diproses.
   - POST /ack untuk memberi tanda bahwa pesan telah berhasil diproses.

C. Cache. menyediakan mekanisme penyimpanan sementara (in-memory cache) dengan kemampuan baca, tulis, dan invalidasi entri. komponen Cache terdapat empat endpoint yang mendukung operasi penyimpanan data sementara secara terdistribusi.
   - GET /read/{key} untuk membaca nilai dari cache berdasarkan kunci tertentu.
   - POST /write untuk menulis atau memperbarui nilai dalam cache.
   - POST /invalidate untuk menghapus entri cache tertentu atau seluruh cache.
   - GET /metrics untuk menampilkan metrik cache seperti jumlah hit, miss, dan kapasitas saat ini.



4.2. Parameter dan respons

Setiap endpoint memiliki struktur permintaan dan tanggapan yang dirancang agar mudah diintegrasikan antar komponen sistem seperti Lock Manager, Queue, dan Cache. Pada komponen Lock Manager, endpoint /lock menerima parameter dalam bentuk objek JSON yang berisi action (jenis operasi seperti ACQUIRE_SHARED, ACQUIRE_EXCLUSIVE, atau RELEASE), lock_name (nama kunci yang ingin dioperasikan), dan client_id (identitas unik klien). Respons yang dihasilkan berupa status operasi, dengan kode 202 jika permintaan diterima oleh leader, 400 untuk permintaan yang tidak valid, 408 bila waktu tunggu habis, dan 423 jika terjadi deadlock. Endpoint /status mengembalikan status terkini dari konsensus Raft, termasuk state (follower, candidate, atau leader) dan current_term.

Pada komponen Queue, endpoint /produce menerima parameter message dalam format JSON dan mengembalikan status keberhasilan penyimpanan pesan. Endpoint /consume tidak memerlukan parameter dan memberikan pesan yang tersedia di antrian, dengan kode 200 jika ada pesan dan 204 jika antrian kosong. Endpoint /ack memerlukan parameter msg_id untuk mengonfirmasi bahwa pesan telah berhasil diproses, dengan respons 200 jika berhasil dan 404 jika ID tidak ditemukan.

Untuk komponen Cache, endpoint /read/{key} menggunakan parameter path key untuk membaca nilai dari cache. Responsnya mencakup key, value, dan source yang menunjukkan apakah data berasal dari cache atau Redis. Endpoint /write membutuhkan parameter key dan value untuk menulis data baru ke cache, sedangkan /invalidate menerima key untuk menandai entri sebagai tidak valid. Endpoint /metrics tidak memerlukan parameter dan mengembalikan statistik performa cache seperti jumlah cache_hits, cache_misses, serta operasi writes dan invalidations.

5. Deployment dan Troubleshooting

5.1. Setup environment

Sebelum melakukan deployment, pastikan sistem telah memenuhi prasyarat utama. Perangkat harus memiliki Git untuk mengkloning repositori dan Docker Desktop (atau Docker Engine & Docker Compose pada Linux) yang sudah dalam kondisi aktif (running). Kedua perangkat ini digunakan untuk membangun dan menjalankan sistem terdistribusi di dalam kontainer Docker.

5.2. Langkah menjalankan sistem

Proses deployment dilakukan melalui dua tahap utama, yaitu memperoleh kode proyek dan menjalankan sistem menggunakan Docker Compose. Pertama, kloning repositori dengan perintah git clone <URL_REPOSITORI> dan masuk ke direktori proyek. Selanjutnya, buka file docker-compose.yml untuk memilih sistem yang akan dijalankan: Distributed Lock Manager, Queue System, atau Cache System. Aktifkan hanya layanan yang diperlukan dengan menghapus tanda komentar (#) pada bagian yang relevan.

Untuk menjalankan sistem, gunakan perintah: *docker-compose up --build.* Perintah ini akan membangun image dan menjalankan seluruh node dalam klaster. Setelah aktif, sistem dapat diverifikasi menggunakan perintah curl untuk menguji endpoint masing-masing. Misalnya, untuk Lock Manager dapat mengirim request POST /lock, untuk Queue System menggunakan POST /produce, dan untuk Cache System dapat diuji melalui POST /write dan GET /read/{key}.

5.3. Troubleshooting umum

Beberapa permasalahan umum yang mungkin muncul saat deployment meliputi:

A. Port sudah digunakan: muncul pesan port is already allocated. Solusinya, hentikan semua kontainer lama dengan docker-compose down sebelum menjalankan ulang.
B. Docker daemon tidak aktif: menyebabkan error Cannot connect to the Docker daemon. Pastikan Docker Desktop sudah berjalan sepenuhnya sebelum memulai sistem.
C. Kesalahan koneksi antar node: error Connection refused biasanya disebabkan karena layanan yang salah dikonfigurasi dalam docker-compose.yml. Periksa kembali port dan layanan yang aktif.
D. Kode tidak diperbarui: jika perubahan pada kode Python tidak terlihat, jalankan perintah dengan opsi --build untuk memaksa Docker membangun ulang image dengan kode terbaru.
E. Dengan langkah-langkah di atas, sistem dapat berjalan secara konsisten dan dapat dipulihkan dari sebagian besar error umum selama proses deployment.

6. Pengujian & Analisis Kinerja

Bagian ini membahas metodologi pengujian kinerja yang dilakukan pada tiga komponen utama sistem terdistribusi: Distributed Lock Manager, Distributed Queue System, dan Distributed Cache System. Setiap komponen diuji menggunakan pendekatan beban (load testing) untuk mengukur throughput, latency, serta stabilitas sistem di bawah kondisi beban tinggi.

6.1. Metodologi pengujian

Pengujian dilakukan menggunakan Locust (versi 2.42.1) untuk mensimulasikan pengguna secara bersamaan. Tujuan utama pengujian adalah membandingkan performa sistem dalam dua mode operasi: Single-Node dan Distributed (3-Node Cluster). Parameter umum pada pengujian antara lain:

- Jumlah pengguna virtual: 10
- Tingkat kelahiran pengguna (spawn rate): 10 pengguna/detik
- Durasi pengujian: 2 menit setelah semua pengguna aktif
- Target host: http://localhost:<port> menyesuaikan layanan yang diuji

A. Distributed Lock Manager

Skenario: Pengguna melakukan siklus acquire dan release terhadap beberapa sumber daya (resource-A, B, C) secara acak.

B. Distributed Queue System

Skenario: Pengguna menghasilkan pesan menggunakan endpoint /produce, kemudian mengambilnya dengan /consume. Pengujian fokus pada stabilitas antrian dan waktu pemrosesan pesan.

C. Distributed Cache System

Skenario: Pengguna menulis nilai baru menggunakan /write dan membaca kembali menggunakan /read/{key}. Tujuannya adalah mengevaluasi waktu respon dan konsistensi antar node cache.

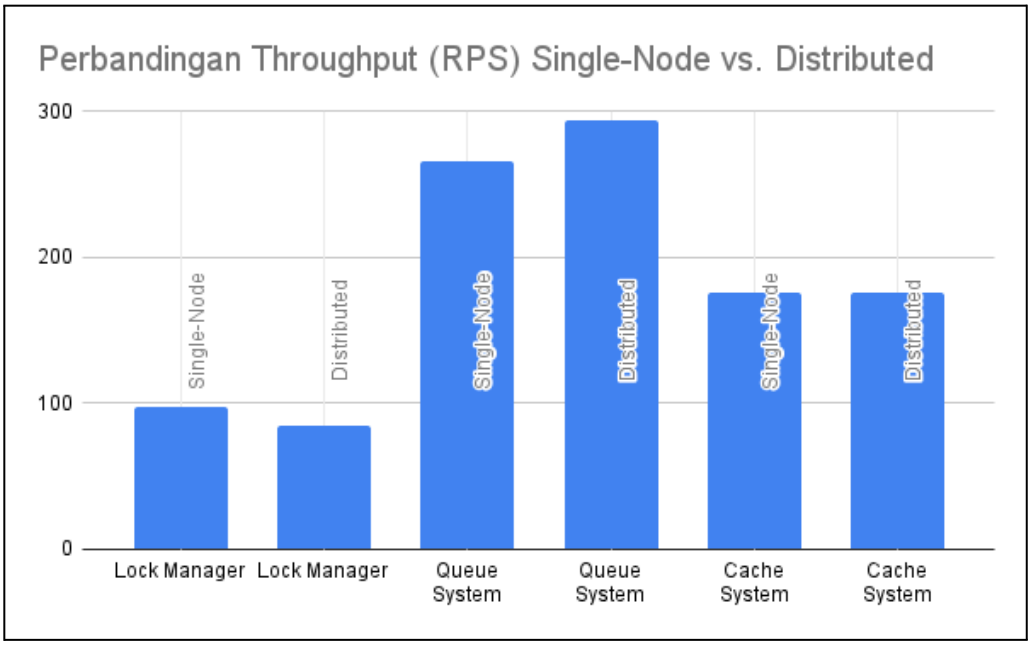
6.2. Hasil benchmarking

Sistem	**Konfigurasi**	**Throughput (RPS)**	**Latency Rata-rata (ms)**	**Latency 95%ile (ms)**	**Failure Rate (%)**
Lock Manager	Single-node	~97.6	~57.1	~59	0
Lock Manager	3-node	~84.2	~46.4	~57	49
Queue	Single-node	~265.6	~169.81	~300	0
Queue	3-node	~294.4	~157.27	~290	0
Cache	Single-node	**~175.9**	**~17.23**	**~52**	**0**

Cache	3-node	**~176**	**~19.25**	**~66**	**0**

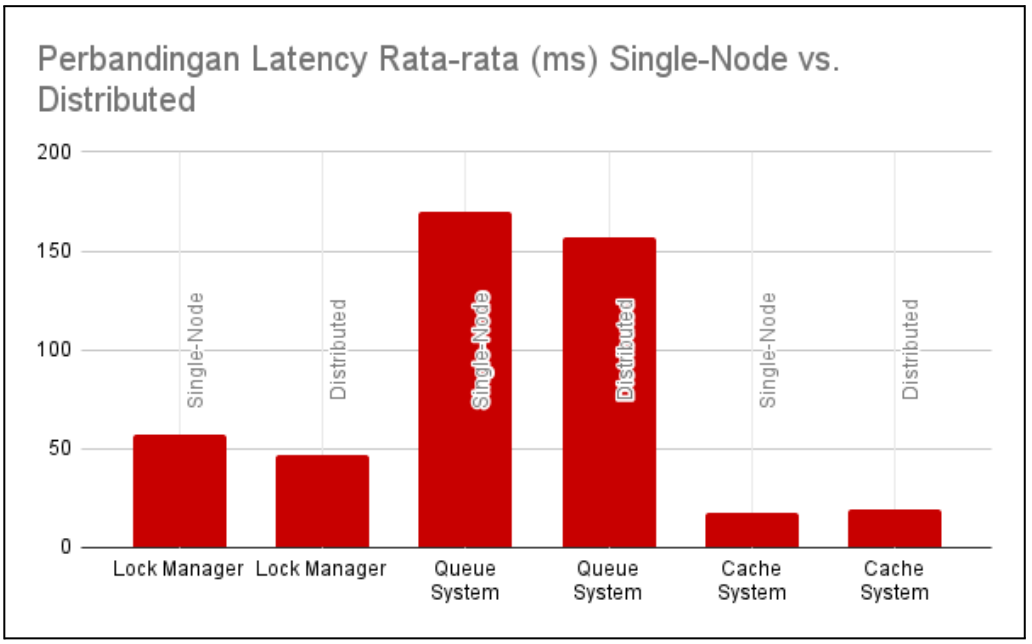
Hasil benchmarking menunjukkan bahwa setiap komponen sistem terdistribusi memiliki karakteristik kinerja yang berbeda. Lock Manager mengalami penurunan throughput dari ~97,6 RPS menjadi ~84,2 RPS pada konfigurasi 3-node, dengan peningkatan tingkat kegagalan sebesar 49% akibat overhead konsensus Raft dan mekanisme replikasi log. Sebaliknya, Distributed Queue System menunjukkan peningkatan throughput dari ~265,6 RPS menjadi ~294,4 RPS, menandakan efisiensi antrian meningkat seiring distribusi beban antar node. Distributed Cache System mempertahankan performa stabil dengan throughput sekitar ~176 RPS dan latency rata-rata rendah (~17–19 ms), menandakan kemampuan tinggi dalam membaca dan menulis data dengan cepat serta konsistensi antar node yang baik. Secara keseluruhan, sistem menunjukkan adanya trade-off antara ketersediaan tinggi dan latensi, di mana konfigurasi terdistribusi memberikan skalabilitas dan resiliensi lebih baik dengan sedikit penurunan efisiensi pada beberapa komponen.

6.3. Visualisasi performa



Grafik ini menunjukkan perbandingan throughput antar komponen sistem pada konfigurasi single-node dan distributed. Terlihat bahwa Queue System

memiliki throughput tertinggi, mencapai hampir 300 RPS dalam mode terdistribusi, menunjukkan efisiensi tinggi dalam pemrosesan pesan paralel. Cache System menunjukkan stabilitas performa dengan perbedaan kecil antara kedua konfigurasi, mengindikasikan mekanisme sinkronisasi antar node yang ringan. Sementara itu, Lock Manager memiliki throughput paling rendah karena overhead algoritma konsensus Raft yang menambah latensi komunikasi antar node.



Grafik ini memperlihatkan perbandingan rata-rata latency dari masing-masing sistem. Lock Manager menunjukkan latency rendah (<60 ms) pada kedua konfigurasi, menandakan efisiensi dalam pengelolaan kunci meskipun ada beban koordinasi. Queue System memiliki latency paling tinggi karena proses enqueue–dequeue dan replikasi log menambah waktu tunggu. Sebaliknya, Cache System menampilkan latency paling rendah (sekitar 17–19 ms), membuktikan performa cepat dalam operasi read/write serta efisiensi protokol konsistensi cache antar node.

6.4. Analisis dan pembahasan

Hasil pengujian menunjukkan bahwa performa sistem bervariasi tergantung pada karakteristik tiap komponen. Pada **Lock Manager**, throughput menurun sekitar 13% dalam konfigurasi 3-node akibat overhead proses konsensus Raft, meskipun latency sedikit membaik. Hal ini menegaskan bahwa peningkatan reliabilitas dan konsistensi global memerlukan kompromi pada efisiensi. Sebaliknya, **Queue System** menunjukkan peningkatan throughput saat dijalankan secara

terdistribusi karena beban kerja dapat dibagi merata antar node, meskipun latency masih cukup tinggi akibat komunikasi dan logging untuk menjaga *at-least-once delivery*.

Sementara itu, **Cache System** memperlihatkan performa paling stabil dengan throughput hampir identik antara single-node dan 3-node, serta latency rendah (~17–19 ms). Hasil ini menunjukkan bahwa mekanisme *cache coherence* bekerja efisien tanpa overhead signifikan. Secara keseluruhan, arsitektur terdistribusi yang dibangun berhasil menjaga keseimbangan antara konsistensi, ketersediaan, dan efisiensi, di mana Lock Manager menonjol pada konsistensi, Queue pada skalabilitas, dan Cache pada kecepatan akses data.

7. Kesimpulan dan Saran

7.1. Kesimpulan

Proyek ini berhasil mengimplementasikan tiga arsitektur sistem sinkronisasi terdistribusi dengan karakteristik dan trade-off berbeda. Distributed Lock Manager menggunakan algoritma Raft untuk mencapai konsensus, replikasi state, dan deteksi deadlock secara efektif, meskipun ada penurunan throughput. Distributed Queue System berbasis Consistent Hashing menunjukkan skalabilitas horizontal dan mendukung message persistence serta at-least-once delivery. Distributed Cache Coherence menerapkan protokol write-invalidation (MESI) untuk menjaga konsistensi antar node dengan sedikit peningkatan latensi tulis. Semua sistem berhasil dijalankan dalam Docker Compose, membuktikan portabilitas dan keandalan desain sistem terdistribusi.

7.2. Saran

Saran terhadap sistem ini kedepannya adalah sebagai berikut:

- Menambahkan mekanisme replikasi antar-node untuk mencegah kehilangan data saat terjadi kegagalan.
- Mengintegrasikan service discovery (misalnya Consul atau etcd) agar node dapat bergabung dan keluar secara otomatis tanpa konfigurasi manual.
- Menghubungkan sistem dengan Prometheus dan Grafana untuk pemantauan kinerja dan visualisasi metrik secara real-time.

- Mengembangkan klien Lock Manager agar dapat otomatis mendeteksi dan beralih ke leader yang aktif tanpa intervensi pengguna.

8. Link dan bukti

8.1. Link Github Repository: https://github.com/Gust4vooo/sister-synchronization.git

8.2. Link video demonstrasi : https://youtu.be/C_sFpbHBIR4