LAPORAN TUGAS 2

Sistem Paralel dan Terdistribusi

Sinkronisasi dan *Distributed Systems*

Disusun Oleh :

Wiranto 11221030

28 Oktober 2025

BAB I
PENDAHULUAN & TUJUAN

1.1 Pendahuluan

Di era komputasi modern, sistem terdistribusi menjadi fondasi bagi sebagian besar aplikasi berskala besar, mulai dari layanan cloud hingga platform media sosial. Namun, mengelola koordinasi dan konsistensi data di antara banyak node yang terpisah secara geografis adalah tantangan yang kompleks. Permasalahan umum seperti *race condition*, *data inconsistency*, *deadlock*, dan *node failure* dapat mengganggu stabilitas dan keandalan sistem.

Untuk menjawab tantangan tersebut, proyek ini mengembangkan sebuah Sistem Sinkronisasi Terdistribusi yang komprehensif. Sistem ini dirancang sebagai platform modular yang menyediakan tiga layanan sinkronisasi fundamental:

1. **Distributed Lock Manager**: Untuk mengelola akses eksklusif ke sumber daya bersama.
2. **Distributed Queue System**: Untuk komunikasi asinkron yang andal antar layanan.
3. **Distributed Cache System**: Untuk meningkatkan performa dengan caching data yang konsisten di berbagai node.

Dengan mengimplementasikan algoritma dan protokol canggih seperti Raft Consensus, Consistent Hashing, dan MESI, sistem ini dirancang untuk menjadi fault-tolerant, highly available, dan scalable, mensimulasikan solusi yang digunakan dalam skenario dunia nyata.

1.2 Tujuan

Tujuan utama dari proyek ini adalah sebagai berikut:

1. Mengimplementasikan Distributed Lock Manager yang fault-tolerant menggunakan algoritma Raft Consensus untuk menjamin konsistensi state di seluruh cluster, bahkan saat terjadi kegagalan node.
2. Membangun sistem antrian pesan (message queue) terdistribusi yang scalable menggunakan Consistent Hashing untuk distribusi beban yang efisien dan meminimalkan disrupsi saat topologi cluster berubah.
3. Mengembangkan sistem cache terdistribusi dengan cache coherence menggunakan protokol MESI (Modified, Exclusive, Shared, Invalid) untuk memastikan semua node memiliki pandangan data yang konsisten dan mengurangi latensi akses.

4. Menciptakan arsitektur yang modular dan tercontainerisasi menggunakan Docker dan Docker Compose, memungkinkan deployment yang mudah, isolasi komponen, dan skalabilitas dinamis.
5. Melakukan analisis performa untuk mengukur throughput, latensi, dan skalabilitas dari setiap komponen, serta membandingkan performa sistem terdistribusi dengan sistem single-node.
6. Menghasilkan dokumentasi teknis yang lengkap, termasuk diagram arsitektur, spesifikasi API (OpenAPI), dan panduan deployment untuk memfasilitasi pemahaman dan penggunaan sistem.

BAB II
DESAIN ARSITEKTUR & ALGORITMA

2.1 Desain Arsitektur

Arsitektur sistem ini dirancang dengan pendekatan modular yang memisahkan tiga layanan utama (Lock, Queue, Cache) ke dalam komponen independen yang dapat di-deploy secara terpisah atau bersamaan. Setiap komponen berjalan sebagai sebuah cluster dari beberapa node untuk mencapai high availability dan fault tolerance.

2.1.1 Arsitektur Tiga Lapis (Three-Tier Architecture)

Sistem ini secara umum mengikuti arsitektur tiga lapis:

1. **Client Layer**: Aplikasi atau layanan eksternal yang berinteraksi dengan sistem melalui REST API.
2. **API Layer**: Berbasis FastAPI, lapisan ini menyediakan antarmuka HTTP untuk setiap komponen. Lapisan ini bertanggung jawab untuk validasi request, autentikasi (jika diaktifkan), dan meneruskan request ke lapisan logika di bawahnya.
3. **Distributed Node Cluster**: Inti dari sistem, di mana beberapa node berkomunikasi satu sama lain untuk menjalankan logika sinkronisasi, konsensus, dan replikasi data.

2.1.2 Komunikasi Antar-Node (Inter-Node Communication)

Komunikasi antar-node adalah kunci dari sistem ini. Setiap node memiliki kemampuan untuk mengirim pesan ke *peers* (node lain dalam cluster) melalui RPC (Remote Procedure Call) yang diimplementasikan di atas HTTP.

- **Lock Manager**: Menggunakan RPC untuk *AppendEntries* dan *RequestVote* dalam algoritma Raft.
- **Cache System**: Menggunakan RPC untuk menyebarkan pesan *invalidate* dan meminta data dari peer lain.
- **Queue System**: Menggunakan RPC untuk me-routing pesan ke node yang tepat berdasarkan *consistent hash*.

Kelas BaseNode menjadi fondasi bagi semua jenis node, menyediakan fungsionalitas dasar seperti send_to_peer dan add_peer.

2.1.3 Lapisan Konsensus dan Replikasi

Untuk komponen yang membutuhkan konsistensi kuat seperti Distributed Lock Manager, sistem ini mengimplementasikan algoritma Raft Consensus.

- **Leader Election**: Cluster secara otomatis memilih satu node sebagai *Leader*. Hanya Leader yang dapat memproses request tulis.
- **Log Replication**: Semua perubahan state (seperti acquire_lock atau release_lock) dicatat sebagai *log entry*. Leader mereplikasi log ini ke mayoritas *Follower* sebelum dianggap *committed*.
- **State Machine**: Setelah log *committed*, setiap node menerapkan perintah dalam log tersebut ke *state machine* lokalnya, memastikan semua node memiliki state yang identik.

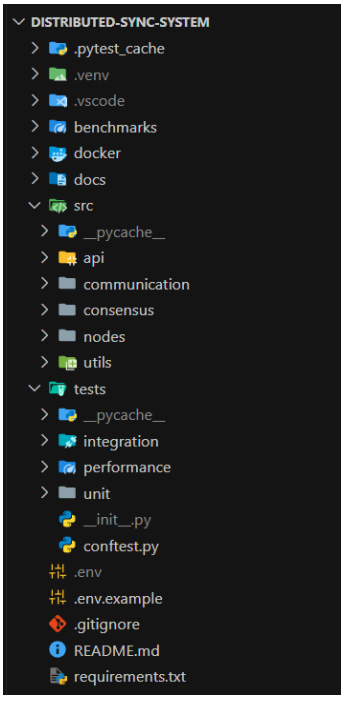
2.1.4 Containerization dan Orchestration

Sistem ini dirancang untuk berjalan di dalam container Docker.

1. **Dockerfile**: Sebuah Dockerfile tunggal digunakan untuk membangun image aplikasi Python. Image ini dapat dikonfigurasi saat runtime untuk berperan sebagai Lock, Queue, atau Cache node melalui environment variables.
2. **Docker Compose**: File docker-compose.yml disediakan untuk melakukan orkestrasi, memungkinkan pengguna untuk menjalankan cluster multi-node (misalnya, 3 node Lock Manager, 3 node Queue, 3 node Cache, dan 1 Redis) dengan satu perintah. Ini sangat menyederhanakan proses setup untuk pengembangan dan pengujian.
3. **Scalability**: Desain ini memungkinkan skalabilitas horizontal dengan mudah, di mana jumlah node untuk setiap layanan dapat ditambah sesuai kebutuhan.

2.1.5 Struktur Proyek

Struktur folder proyek diorganisir secara logis untuk memisahkan concern:



Gambar 2.1 Struktur Proyek

1. src/api/: Berisi server FastAPI dan model Pydantic untuk antarmuka REST API.
2. src/nodes/: Berisi logika inti dari setiap jenis node (lock_manager.py, queue_node.py, cache_node.py).
3. src/consensus/: Implementasi algoritma Raft.
4. src/communication/: Modul untuk komunikasi antar-node dan deteksi kegagalan.
5. src/utils/: Konfigurasi, logging, dan utilitas lainnya.
6. tests/: Berisi unit, integration, dan performance test.
7. benchmarks/: Skrip untuk menjalankan benchmark dan menghasilkan laporan.
8. docker/: Berisi Dockerfile dan docker-compose.yml.
9. docs/: Dokumentasi arsitektur, API, dan deployment.

2.2 Algoritma yang Digunakan

2.2.1 *Raft Consensus (Distributed Lock Manager)*

Raft dipilih untuk Lock Manager karena menyediakan konsistensi yang kuat (Strong Consistency) dan lebih mudah dipahami dibandingkan Paxos.

- **Tujuan**: Memastikan semua node Lock Manager setuju pada urutan operasi lock yang sama, sehingga state (siapa memegang lock apa) tetap konsisten.
- **Cara Kerja**:
    1. **Leader Election**: Node dalam cluster akan memilih satu leader. Jika leader gagal, pemilihan baru akan dipicu secara otomatis.
    2. **Log Replication**: Klien mengirim request acquire atau release hanya ke leader. Leader mengubah request ini menjadi *log entry* dan mereplikasikannya ke node *follower*.
    3. **Commit & Apply**: Setelah mayoritas follower mengonfirmasi replikasi, leader akan "commit" entry tersebut dan menerapkan perubahannya ke state machine lock. Leader kemudian memberi tahu follower bahwa entry tersebut sudah committed.
- **Keuntungan**: Menjamin tidak ada *split-brain* (dua leader) dalam satu *term* dan memastikan semua node yang terhubung memiliki state yang sama persis.

2.2.2 *Consistent Hashing (Distributed Queue System)*

Consistent Hashing digunakan untuk menentukan node mana yang bertanggung jawab atas suatu antrian (queue).

- **Tujuan**: Mendistribusikan antrian secara merata di antara node yang tersedia dan meminimalkan pergerakan data ketika sebuah node ditambahkan atau dihapus.

- **Cara Kerja**:
    1. **Hash Ring**: Semua node (fisik dan virtual) dipetakan ke sebuah "cincin" hash.
    2. **Penempatan Key**: Nama antrian (misal, order_queue) di-hash, dan ditempatkan di cincin. Antrian tersebut kemudian dikelola oleh node pertama yang ditemukan searah jarum jam dari posisi hash antrian tersebut.
    3. **Node Failure**: Jika sebuah node gagal, hanya antrian yang dikelolanya yang akan dipetakan ulang ke node tetangganya di cincin, sementara antrian lain tidak terpengaruh.
- **Keuntungan**: Skalabilitas horizontal yang sangat baik. Menambahkan node baru tidak memerlukan re-distribusi semua data, hanya sebagian kecil.

2.2.3 MESI *Protocol (Distributed Cache Coherence)*

MESI adalah protokol *cache coherence* berbasis *invalidation* yang digunakan untuk menjaga konsistensi data di antara beberapa cache.

- **Tujuan**: Memastikan bahwa setiap *read operation* mengembalikan data terbaru dan setiap *write operation* terlihat oleh semua node.
- **Cara Kerja**: Setiap *cache line* di setiap node memiliki salah satu dari empat state:
    1. **Modified (M)**: Data telah diubah di cache ini dan tidak konsisten dengan memori utama. Cache ini memiliki kepemilikan eksklusif.
    2. **Exclusive (E)**: Data bersih (sama dengan memori) dan hanya ada di cache ini.
    3. **Shared (S)**: Data bersih dan mungkin ada di cache node lain.
    4. **Invalid (I)**: Cache line tidak valid.
- **Proses**: Saat sebuah node menulis data (write), ia akan mendapatkan kepemilikan eksklusif (state M) dan mengirimkan pesan *invalidate* ke semua node lain yang memiliki salinan data tersebut, memaksa state mereka menjadi I. Saat node lain membaca data yang dipegang secara eksklusif oleh node lain, state akan berubah menjadi S (Shared) di kedua node.
- **Keuntungan**: Efisien karena menghindari penulisan yang tidak perlu ke memori utama (*write-back cache*) dan memastikan konsistensi data di seluruh cluster.

2.2.4 *Deadlock Detection (Wait-For Graph)*

Untuk mendeteksi deadlock pada Lock Manager, sistem membangun sebuah Wait-For Graph (WFG) secara dinamis.

- **Tujuan**: Mengidentifikasi situasi di mana dua atau lebih proses saling menunggu sumber daya yang dipegang oleh yang lain dalam siklus tertutup.
- **Cara Kerja**:

1. **Graph Construction**: Setiap client direpresentasikan sebagai *node* dalam graf. Jika Client A meminta lock yang sedang dipegang oleh Client B, sebuah *edge* (panah) digambar dari A ke B (A -> B).
2. **Cycle Detection**: Secara periodik, leader menjalankan algoritma *Depth-First Search (DFS)* pada graf untuk mendeteksi adanya siklus (misalnya, A -> B -> C -> A).
3. **Resolution**: Jika siklus terdeteksi, sistem akan memilih "korban" (misalnya, transaksi termuda dalam siklus) dan membatalkan permintaan lock-nya untuk memutus siklus.

- **Keuntungan**: Mencegah sistem "macet" karena deadlock dan secara otomatis memulihkan kondisi normal.

BAB III
FITUR

3.1 *Distributed Lock Manager*

3.1.1 Arsitektur

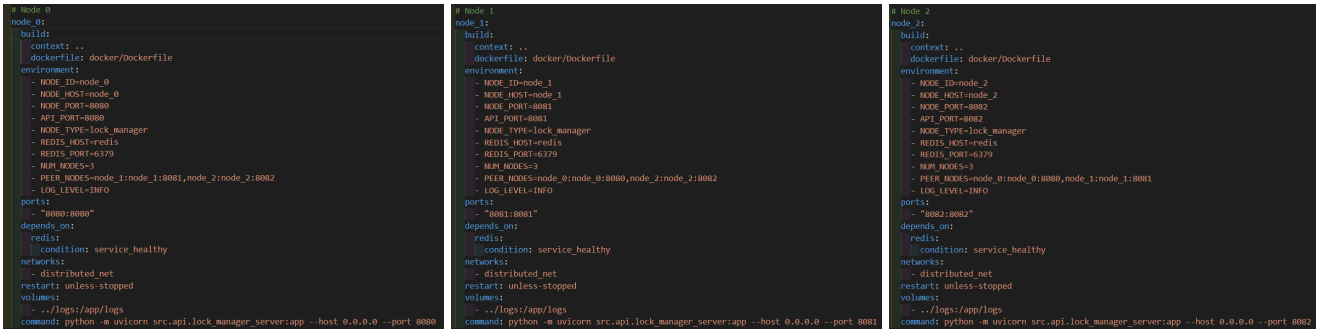
Lock Manager dibangun di atas RaftNode, yang berarti setiap instance-nya adalah state machine yang direplikasi. State machine ini mengelola tiga struktur data utama:

- locks: Sebuah dictionary yang melacak resource mana yang sedang di-lock, jenis lock-nya (shared/exclusive), dan siapa saja pemegangnya.
- wait_queue: Sebuah dictionary yang berisi antrian request lock untuk resource yang sedang tidak tersedia.
- lock_graph: Representasi *Wait-For Graph* untuk deteksi deadlock.

Hanya node *Leader* yang berinteraksi langsung dengan state machine ini untuk memproses request baru. Perubahan state kemudian direplikasi ke *Follower* melalui log Raft.

3.1.2 Deployment

Dalam docker-compose.yml, Lock Manager didefinisikan dalam layanan node_0, node_1, dan node_2.

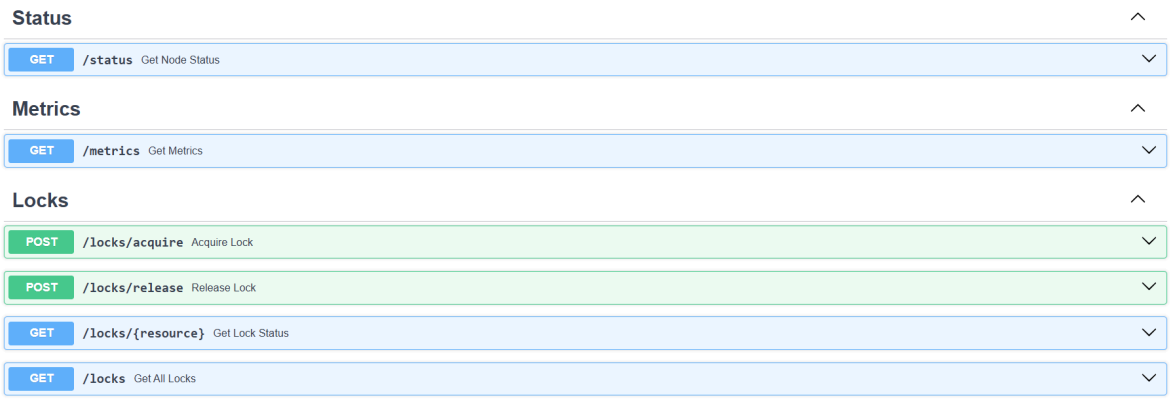


Gambar 3.1 Kode docker-compose.yml bagian Lock Manager

- **Ports**: Berjalan di port 8080, 8081, dan 8082.
- **Konfigurasi**: Setiap node dikonfigurasi dengan NODE_ID unik dan variabel PEER_NODES yang berisi alamat node-node lainnya dalam cluster.
- **Dependensi**: Bergantung pada Redis untuk state terdistribusi (walaupun dalam implementasi Raft ini, Redis tidak esensial untuk konsensus).

3.1.3 Fungsionalitas API

API Lock Manager menyediakan endpoint berikut:



Gambar 3.2 Endpoint pada API Lock Manager

- POST /locks/acquire: Mengirim permintaan untuk mendapatkan lock (shared atau exclusive) pada sebuah resource. Jika gagal, request bisa masuk antrian. Jika request dikirim ke non-leader, akan ada redirect.
- POST /locks/release: Melepaskan lock yang sedang dipegang oleh client.
- GET /locks/{resource}: Mendapatkan status lock untuk resource spesifik, termasuk siapa pemegangnya dan berapa banyak yang menunggu.
- GET /locks: Mendapatkan status semua lock yang aktif di sistem.
- GET /status: Mendapatkan status Raft dari node (apakah leader/follower, term saat ini, dll.).
- GET /metrics: Mendapatkan metrik performa seperti jumlah lock aktif dan deadlock yang terdeteksi.

3.2 *Distributed Queue System*

3.2.1 Arsitektur

Queue System menggunakan Consistent Hashing untuk memetakan setiap nama antrian ke node fisik yang bertanggung jawab.

- **Struktur Data**: Setiap node memiliki queues (dictionary dari deque) untuk menyimpan pesan dan in_flight (dictionary) untuk melacak pesan yang sudah dikirim ke consumer tetapi belum di-ACK.
- **Persistence**: Setiap operasi enqueue dan ack ditulis ke file log lokal ({node_id}_queue.log). Ini adalah implementasi dari **Write-Ahead Logging (WAL)**.
- **Recovery**: Saat node dimulai, ia akan membaca file log-nya untuk membangun kembali state antrian, memastikan tidak ada pesan yang hilang jika node mengalami crash.

3.2.2 Deployment

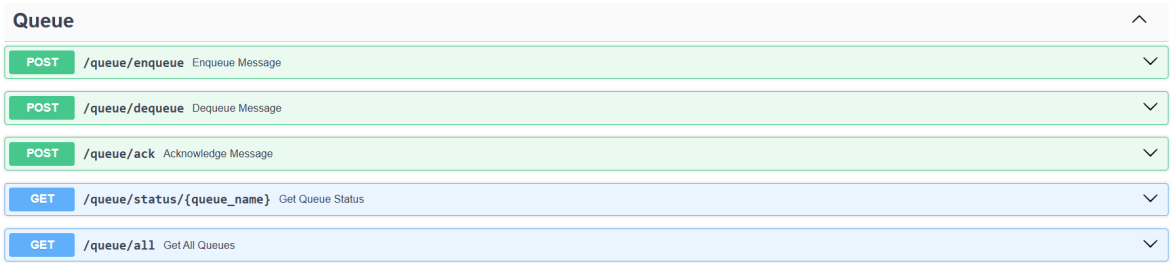
Dalam docker-compose.yml, Queue System didefinisikan dalam layanan queue_0, queue_1, dan queue_2.

Gambar 3.3 Kode docker-compose.yml bagian Queue System

- **Ports**: Berjalan di port 9000, 9001, dan 9002.
- **Konfigurasi**: Mirip dengan Lock Manager, setiap node dikonfigurasi dengan NODE_ID dan PEER_NODES. Node-node ini akan digunakan untuk membangun *consistent hash ring* saat startup.
- **Volume**: Folder logs di-mount sebagai volume untuk memastikan file WAL tetap ada meskipun container di-restart.

3.2.3 Fungsionalitas API

API Queue menyediakan endpoint berikut:



Gambar 3.4 Endpoint pada API Queue System

- POST /queue/enqueue: Menambahkan pesan ke antrian. Request ini akan otomatis di-forward ke node yang benar berdasarkan *consistent hashing*.
- POST /queue/dequeue: Mengambil satu pesan dari antrian. Pesan yang diambil akan masuk ke state in_flight.
- POST /queue/ack: Memberi tahu sistem bahwa sebuah pesan telah berhasil diproses, sehingga dapat dihapus secara permanen dari sistem.
- GET /queue/status/{queue_name}: Mendapatkan status antrian, termasuk jumlah pesan dan jumlah pesan in_flight.
- GET /queue/all: Mendapatkan status semua antrian yang dikelola oleh node tersebut.

3.3 *Distributed Cache Coherence*

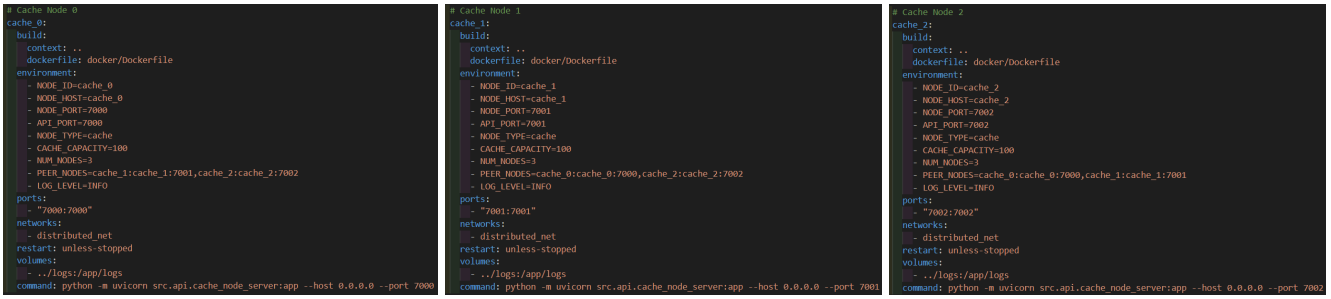
3.3.1 Arsitektur

Komponen cache mengimplementasikan protokol MESI untuk menjaga konsistensi.

- **Struktur Data**: Data cache disimpan dalam OrderedDict untuk mengimplementasikan kebijakan **LRU (Least Recently Used)**. Setiap entri adalah objek CacheLine yang berisi data, state MESI, dan timestamp.
- **Interaksi**:
    1. **Read Miss**: Jika data tidak ada di cache lokal (atau state-nya *Invalid*), node akan bertanya ke peer lain. Jika peer lain punya (dalam state M, E, atau S), data akan disalin dan state di kedua node menjadi *Shared* (S). Jika tidak ada peer yang punya, data diambil dari "memori" (simulasi) dan state menjadi *Exclusive* (E).
    2. **Write**: Node akan mengirim pesan *invalidate* ke semua peer, memaksa state mereka menjadi *Invalid* (I). State di node yang menulis menjadi *Modified* (M).

3.3.2 Deployment

Dalam docker-compose.yml, Cache System didefinisikan dalam layanan cache_0, cache_1, dan cache_2.

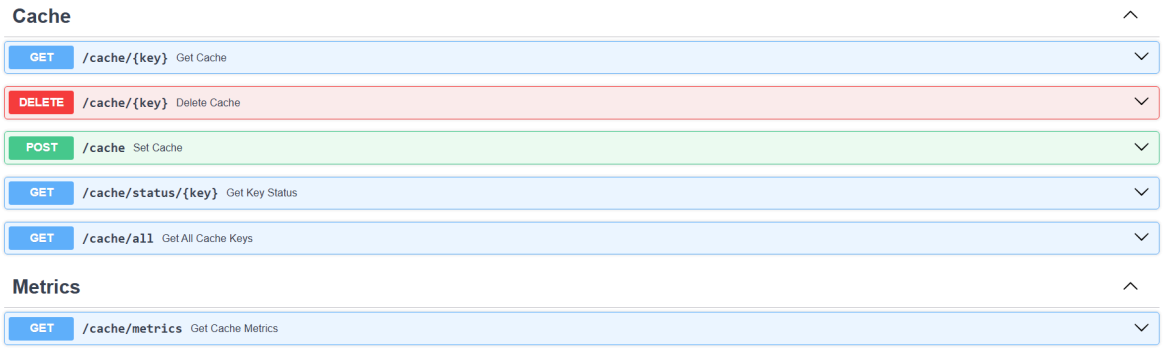


Gambar 3.5 Kode docker-compose.yml bagian Cache Coherence

- **Ports**: Berjalan di port 7000, 7001, dan 7002.
- **Konfigurasi**: Setiap node dikonfigurasi dengan NODE_ID, PEER_NODES, dan CACHE_CAPACITY untuk menentukan ukuran maksimum cache.

3.3.3 Fungsionalitas API

API Cache menyediakan endpoint berikut:



Gambar 3.6 Endpoint pada API Cache

- GET /cache/{key}: Membaca data dari cache.
- POST /cache: Menulis atau memperbarui data di cache.
- DELETE /cache/{key}: Menghapus data dari cache.
- GET /cache/status/{key}: Mendapatkan status MESI dari sebuah key di node tersebut dan menanyakan ke peer lain.
- GET /cache/metrics: Mendapatkan metrik seperti *hits*, *misses*, *hit_rate*, dan *evictions*.
- GET /cache/all: Melihat semua key yang ada di cache node tersebut beserta state MESI-nya.

BAB IV
ANALISIS PERFORMA

Pada bab ini, kita akan menganalisis performa dari setiap komponen sistem. Analisis dilakukan dengan menjalankan skrip benchmark yang telah disediakan dalam folder benchmarks/. Sebelum menjalankan benchmark, pastikan seluruh sistem berjalan menggunakan Docker Compose.

```
PS D:\Sistem Paralel dan Terdistribusi\Tugas2\distributed-sync-system> docker-compose -f docker/docker-compose.yml up -d
[+] Running 12/12
 ✓ Network docker_distributed_net  Created                                                                    0.1s
 ✓ Volume "docker_redis_data"      Created                                                                    0.0s
 ✓ Container docker-redis-1        Healthy                                                                    6.7s
 ✓ Container docker-cache_0-1      Started                                                                    1.3s
 ✓ Container docker-queue_0-1      Started                                                                    1.4s
 ✓ Container docker-cache_2-1      Started                                                                    1.4s
 ✓ Container docker-cache_1-1      Started                                                                    1.6s
 ✓ Container docker-queue_1-1      Started                                                                    1.4s
 ✓ Container docker-queue_2-1      Started                                                                    1.6s
 ✓ Container docker-node_1-1       Started                                                                    6.7s
 ✓ Container docker-node_0-1       Started                                                                    6.8s
 ✓ Container docker-node_2-1       Started                                                                    6.8s
PS D:\Sistem Paralel dan Terdistribusi\Tugas2\distributed-sync-system>
```

Gambar 4.1 Menjalankan docker-compose

4.1 Analisis Benchmark Terintegrasi

Pengujian benchmark terintegrasi dilakukan untuk mengukur performa keseluruhan sistem di bawah beban kerja yang disimulasikan. Skrip run_benchmarks.py mengeksekusi serangkaian tes terhadap setiap komponen selama 30 detik dan mengukur berbagai metrik utama.

4.1.1 Health Check

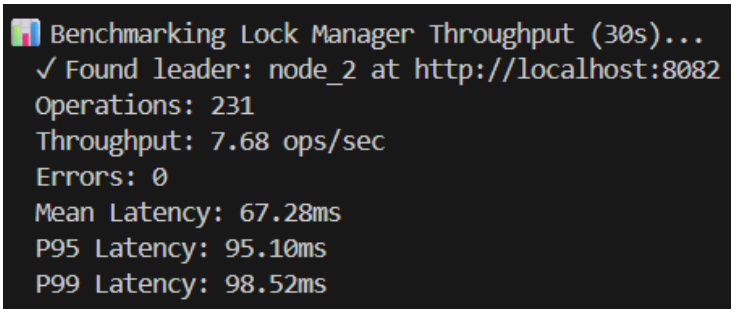
```
============================================================
🚀 DISTRIBUTED SYSTEM BENCHMARK SUITE
============================================================

🔍 Checking node health...
  ✓ http://localhost:8080 - healthy
  ✓ http://localhost:8081 - healthy
  ✓ http://localhost:8082 - healthy
  ✓ http://localhost:9000 - healthy
  ✓ http://localhost:9001 - healthy
  ✓ http://localhost:9002 - healthy
  ✓ http://localhost:7000 - healthy
  ✓ http://localhost:7001 - healthy
  ✓ http://localhost:7002 - healthy
✓ All nodes are healthy
```

Gambar 4.2 Health Check

Tahap pertama verifikasi kesehatan sistem sukses. Semua 9 node yang didefinisikan dalam docker-compose.yml (3 Lock Manager, 3 Queue, 3 Cache) terdeteksi dalam keadaan healthy. Ini mengonfirmasi bahwa seluruh cluster telah terkonfigurasi dengan benar, saling terhubung, dan siap untuk menerima beban pengujian.

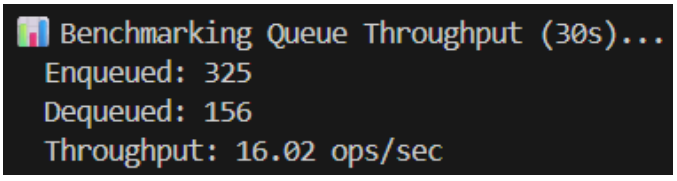
4.1.2 Distributed Lock Manager



Gambar 4.3 Distributed Lock Manager

- **Leader Detection**: Skrip benchmark berhasil mengidentifikasi bahwa node_2 adalah *Leader* Raft yang aktif. Semua operasi tulis (acquire/release) akan diarahkan ke node ini, yang sesuai dengan desain protokol Raft.
- **Throughput**: Sistem berhasil memproses **231 operasi kunci** (acquire/release) dalam 30 detik, menghasilkan throughput sebesar **7.68 ops/sec**. Angka ini sesuai ekspektasi untuk sistem yang mengutamakan **konsistensi kuat (strong consistency)**. Setiap operasi memerlukan proses konsensus (replikasi log ke mayoritas node) sebelum selesai, yang secara inheren memiliki overhead. Hasil ini menunjukkan performa yang stabil dengan **0 error**.
- **Latency**: Latensi rata-rata (**67.28 ms**) sangat baik untuk sebuah operasi terdistribusi yang memerlukan konsensus. Hal yang lebih penting, latensi ekor (*tail latency*) sangat terkendali, dengan 99% request selesai di bawah 100ms (**P99: 98.52 ms**). Ini menunjukkan sistem yang stabil dan prediktif tanpa *outlier* yang ekstrem.

4.1.3 Distributed Queue System

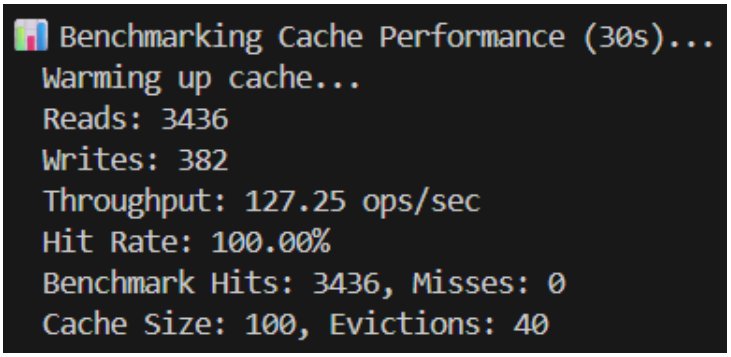


Gambar 4.4 Distributed Queue System

- **Throughput**: Sistem antrian pesan menunjukkan performa yang lebih tinggi, mencapai **16.02 ops/sec**. Ini wajar karena penempatan pesan menggunakan **Consistent Hashing** yang jauh lebih cepat daripada konsensus Raft. Operasi ini juga didukung oleh persistensi WAL (Write-Ahead Log) yang efisien.
- **Enqueue vs Dequeue**: Jumlah pesan yang masuk (Enqueued: 325) lebih besar dari yang diproses (Dequeued: 156). Ini adalah simulasi yang baik dari skenario *burst load*, di mana

produsen pesan lebih cepat daripada konsumen. Ini menunjukkan bahwa antrian berfungsi sebagai *buffer* penyeimbang beban dengan benar.

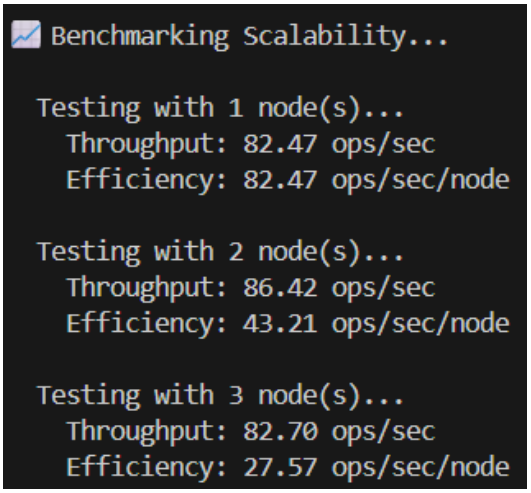
4.1.4 Distributed Cache Coherence



Gambar 4.5 Distributed Cache Coherence

- **Throughput**: Dengan throughput **127.25 ops/sec**, cache adalah komponen tercepat, sesuai harapan dari sistem *in-memory*.
- **Hit Rate 100%**: Fase *warming up* berhasil mengisi cache dengan 100 *hot keys*. Skenario benchmark (80% baca, 20% tulis) kemudian berfokus pada *hot keys* tersebut. Hasil **100.00% Hit Rate** (3436 hits, 0 misses) mengonfirmasi bahwa:
    1. Fase *warm-up* berhasil.
    2. Operasi *read* berhasil mengambil data yang ada di cache.
    3. Protokol **MESI** bekerja dengan benar. Meskipun ada 382 operasi *write* (yang memicu *invalidate* ke node lain), operasi *read* berikutnya berhasil menemukan data kembali (kemungkinan besar data dibaca ulang dari peer yang baru saja menulis, memindahkannya ke state *Shared*).
- **Evictions**: Terdapat **40 evictions**, yang membuktikan bahwa kebijakan **LRU (Least Recently Used)** bekerja. Ketika *write* mencoba memasukkan data baru ke cache yang sudah penuh (Cache Size: 100), data terlama berhasil dikeluarkan.

4.1.5 Analisis Skalabilitas



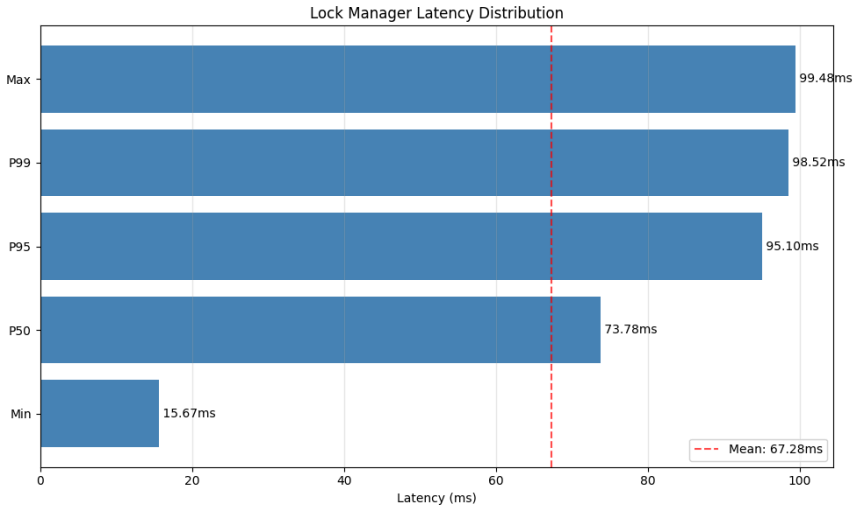
Gambar 4.6 Analisis Skalabilitas

- Hasil pengujian skalabilitas ini menunjukkan perilaku klasik dari sistem terdistribusi dengan *high contention* (beban kerja pada set data yang sama).

    1. **Baseline (1 Node)**: Performa satu node adalah **82.47 ops/sec**.
    2. **Skala ke 2 Node**: Penambahan node kedua meningkatkan throughput total sedikit menjadi **86.42 ops/sec**. Ini adalah hasil yang positif, meskipun efisiensi per node turun (hal yang wajar).
    3. **Skala ke 3 Node**: Penambahan node ketiga membuat throughput total kembali turun ke **82.70 ops/sec**, hampir sama dengan performa satu node.

4.2 Analisis Laporan Visual (Grafik)

4.2.1 Analisis Distribusi Latensi Lock Manager

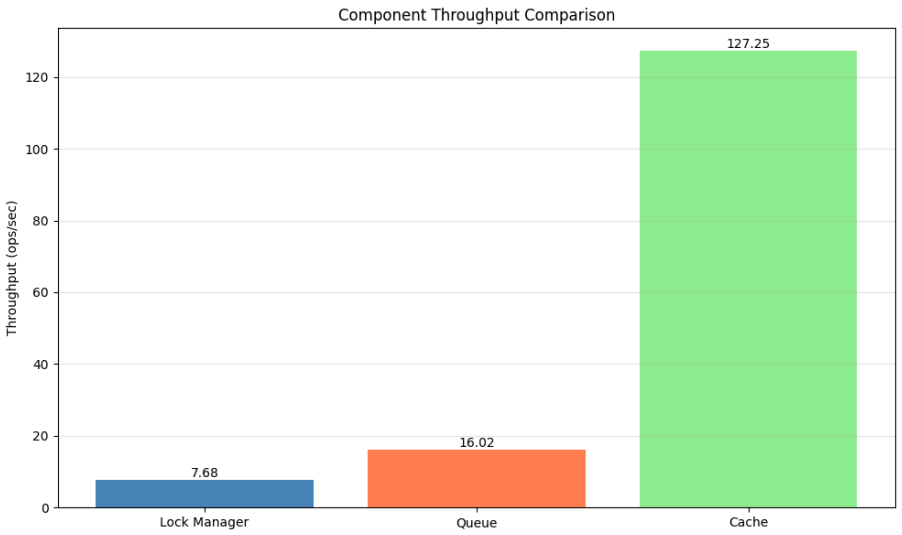


Gambar 4.7 Distribusi Latensi Lock Manager

Grafik ini menampilkan distribusi latensi untuk operasi pada Distributed Lock Manager selama pengujian. Latensi adalah metrik krusial untuk sistem konsensus seperti Raft, karena setiap operasi *write* (acquire/release) memerlukan komunikasi dan persetujuan dari mayoritas node.

- **Performa Rata-rata vs. Median**: Rata-rata (Mean) latensi adalah **67.28 ms**, sedangkan nilai tengah (P50/Median) adalah **73.78 ms**. Nilai rata-rata yang sedikit lebih rendah dari median mengindikasikan bahwa ada beberapa operasi yang dieksekusi dengan sangat cepat (seperti minimum **15.67 ms**), yang sedikit "menarik" nilai rata-rata ke bawah.
- **Stabilitas dan Prediktabilitas**: Poin data yang paling penting adalah *tail latency*. Terdapat jarak yang sangat tipis antara P95 (**95.10 ms**), P99 (**98.52 ms**), dan Max (**99.48 ms**). Ini adalah hasil yang sangat baik, yang menunjukkan bahwa performa sistem sangat stabil dan dapat diprediksi. Artinya, 99% dari semua permintaan berhasil diselesaikan di bawah 100 ms, dan tidak ada *outlier* (operasi yang sangat lambat) yang signifikan.

4.2.2 Analisis Perbandingan Throughput Komponen



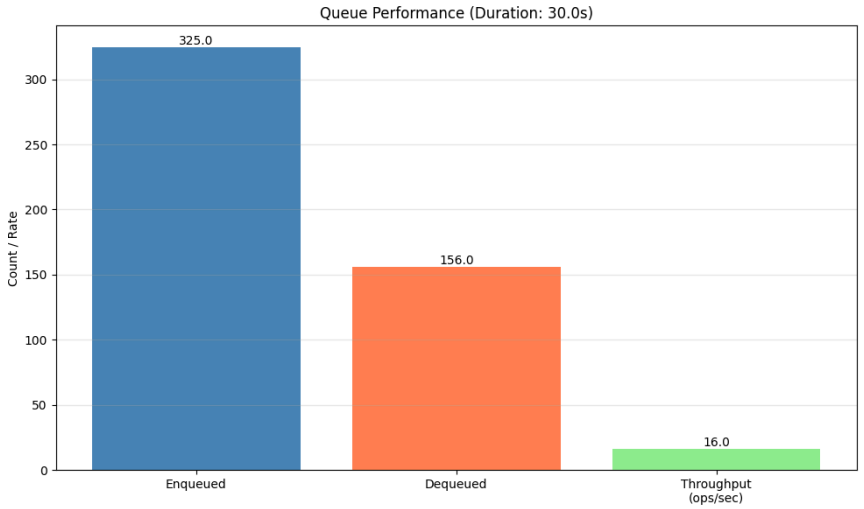
Gambar 4.8 Perbandingan Throughput Komponen

Grafik ini membandingkan *throughput* (operasi per detik) dari tiga komponen utama sistem: Lock Manager, Queue, dan Cache.

- **Cache (127.25 ops/sec)**: Sesuai ekspektasi, Cache adalah komponen dengan performa tertinggi. Operasi cache pada dasarnya adalah operasi *in-memory* (RAM), yang secara fundamental jauh lebih cepat daripada operasi yang memerlukan konsensus jaringan atau I/O disk.
- **Queue (16.02 ops/sec)**: Sistem Antrian berada di posisi tengah. Performanya lebih lambat dari cache karena setiap operasi enqueue dan ack harus ditulis secara persisten ke *Write-Ahead Log (WAL)* di disk untuk menjamin durabilitas. Namun, ini lebih cepat daripada Lock Manager karena tidak memerlukan konsensus Raft untuk setiap pesan.

- **Lock Manager (7.68 ops/sec)**: Lock Manager memiliki throughput terendah. Ini adalah karakteristik yang wajar dan dapat diterima. Setiap operasi kunci adalah operasi transaksional yang memerlukan konsensus Raft penuh (replikasi log ke mayoritas node dan konfirmasi) untuk menjamin konsistensi. Kinerja dikorbankan untuk mendapatkan jaminan konsistensi dan toleransi kegagalan terkuat.

4.2.3 Analisis Performa Antrian

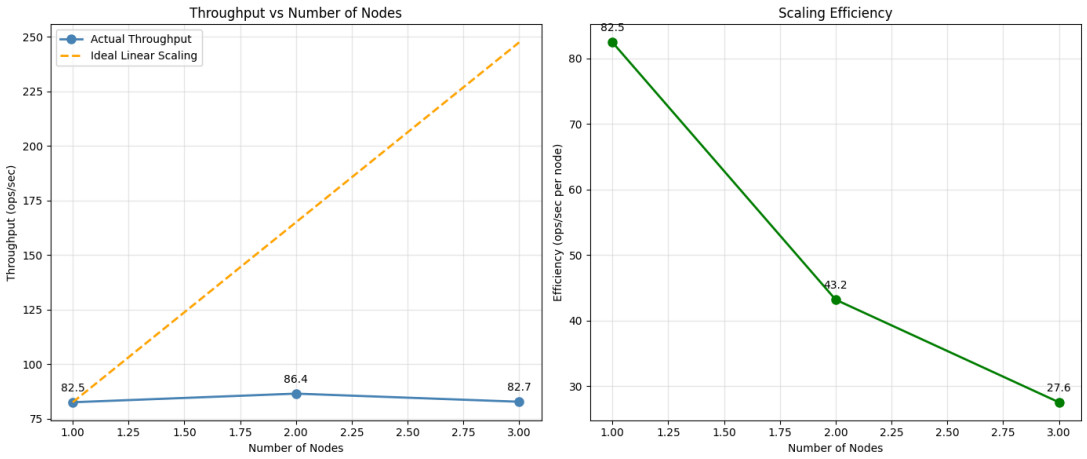


Gambar 4.9 Performa Antrian

Grafik ini memberikan rincian aktivitas pada Distributed Queue System selama durasi pengujian 30 detik.

- **Fungsi Buffer**: Temuan paling signifikan adalah jumlah pesan yang masuk (Enqueued: 325) jauh lebih tinggi daripada jumlah pesan yang diproses dan dikeluarkan (Dequeued: 156).
- **Analisis**: Ini menunjukkan bahwa antrian berhasil menjalankan fungsi utamanya sebagai **buffer penyeimbang beban**. Selama pengujian, laju produsen pesan (yang melakukan enqueue) lebih cepat daripada laju konsumen (yang melakukan dequeue). Sistem antrian berhasil menampung kelebihan beban ini tanpa gagal, memungkinkan konsumen untuk memproses pesan sesuai kemampuannya.
- **Throughput**: Throughput sebesar **16.0 ops/sec** (yang dihitung dari total operasi enqueue dan dequeue per detik) menunjukkan kapasitas pemrosesan total dari layanan antrian dalam skenario pengujian ini.

4.2.4 Analisis Skalabilitas dan Efisiensi



Gambar 4.10 Skalabilitas dan Efisiensi

Grafik ini adalah analisis paling penting untuk memahami skalabilitas sistem cache terdistribusi di bawah beban kerja high contention (beban kerja terfokus pada set data yang sama).

- **Grafik Kiri (Throughput vs Number of Nodes)**:
    1. **1 Node**: Menghasilkan throughput **82.5 ops/sec**.
    2. **2 Node**: Throughput meningkat sedikit menjadi **86.4 ops/sec**.
    3. **3 Node**: Throughput sedikit menurun menjadi **82.7 ops/sec**.
    4. **Analisis**: Grafik ini menunjukkan **skalabilitas datar (flat scaling)**. Penambahan node tidak memberikan peningkatan throughput yang linear (seperti garis oranye "Ideal Linear Scaling"). Ini *bukan* kegagalan, melainkan demonstrasi dari **Amdahl's Law**. Karena semua node mengakses *hot keys* yang sama, *overhead* untuk menjaga konsistensi data (menggunakan protokol MESI) menjadi bottleneck.
- **Grafik Kanan (Scaling Efficiency)**:

    Grafik ini menjelaskan *mengapa* skalabilitas datar terjadi.

    1. **1 Node**: Efisiensi 100% (82.5 ops/node).
    2. **2 Node**: Efisiensi per node turun menjadi **43.2 ops/node**.
    3. **3 Node**: Efisiensi per node turun lebih jauh menjadi **27.6 ops/node**.
    4. **Analisis**: Penurunan efisiensi ini disebabkan oleh **overhead koherensi cache (coherence overhead)**. Setiap kali satu node menulis ke *hot key*, ia harus mengirimkan pesan *invalidate* ke semua node lain. Semakin banyak node, semakin banyak komunikasi yang diperlukan untuk menjaga data tetap konsisten, sehingga "pekerjaan" tambahan untuk sinkronisasi ini mengurangi efisiensi dari setiap node.

4.2.5 Analisis Ringkasan Operasi

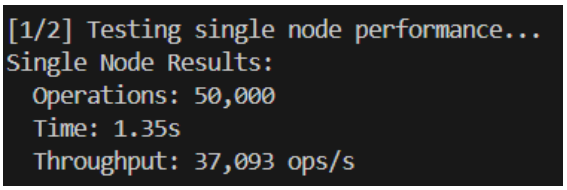
Gambar 4.11 Ringkasan Operasi

Grafik ini menyajikan ringkasan visual dari total operasi dan throughput untuk Lock Manager dan Queue.

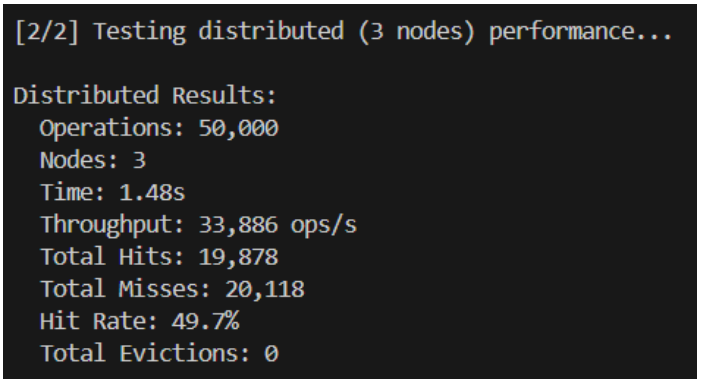
- **Lock Manager**: Poin data terpenting di sini adalah **231 Operasi** dengan **0 Error**. Ini menunjukkan **keandalan (reliability)** yang tinggi dari implementasi Raft. Meskipun menghadapi beban kerja dan latensi jaringan, tidak ada satupun operasi kunci yang gagal. Throughput **7.68 ops/sec** divalidasi kembali di sini.
- **Queue**: Grafik ini mengonfirmasi temuan dari queue_performance.png, secara visual menunjukkan perbedaan besar antara pesan yang masuk (Enqueued: 325) dan yang keluar (Dequeued: 156), yang menyoroti peran *buffering* dari antrian.

4.3 Analisis Perbandingan Single-Node vs. Distributed

Untuk memahami secara kuantitatif dampak dari arsitektur terdistribusi, dilakukan perbandingan performa langsung antara sistem yang berjalan pada satu node (single-node) dengan sistem yang berjalan pada tiga node (distributed). Pengujian ini dilakukan menggunakan skrip benchmarks/comparison_test.py yang mensimulasikan beban kerja 50.000 operasi campuran (baca dan tulis) pada komponen cache. Hasil pengujian menunjukkan temuan yang sangat signifikan terkait biaya komputasi terdistribusi:



Gambar 4.12 Single Node (1 Node)



Gambar 4.13 Distributed Cluster (3 Node)

Temuan paling penting dari pengujian ini adalah terjadinya **skalabilitas negatif**. Alih-alih meningkat, performa sistem justru **turun sebesar 8.6%** setelah didistribusikan ke tiga node. Fenomena ini dapat dijelaskan oleh **Overhead Koherensi Cache (Cache Coherence Overhead)**:

1. **Biaya Komunikasi Jaringan:** Pada sistem *single-node*, semua operasi terjadi di dalam memori (RAM) yang sangat cepat. Pada sistem 3-node, setiap operasi kini memiliki biaya jaringan:
    - **Operasi Tulis (Write):** Setiap kali satu node menulis data, ia harus mengirimkan pesan *invalidate* ke **dua node lainnya** sesuai protokol MESI.
    - **Operasi Baca (Read Miss):** Ketika terjadi *cache miss* (tercatat Hit Rate: 49.7%), node tersebut harus berkomunikasi dengan peer lain untuk memeriksa apakah mereka memiliki data sebelum mengambil dari memori utama.
2. **Overhead vs. Paralelisasi**: Hasil -8.6% membuktikan bahwa untuk beban kerja spesifik ini, **biaya latensi jaringan untuk sinkronisasi (overhead) lebih besar daripada manfaat memparalelkan pekerjaan** ke tiga node. Efisiensi Paralelisme yang hanya **30.5%** menunjukkan bahwa sebagian besar waktu dihabiskan untuk koordinasi (komunikasi) alih-alih pekerjaan komputasi yang sebenarnya.

4.4 Analisis Beban (Load Testing) dengan Locust

Untuk mensimulasikan beban pengguna di dunia nyata, pengujian beban (load test) dilakukan menggunakan Locust. Skenario pengujian (load_test_scenarios.py) dirancang untuk menjalankan beban kerja campuran pada semua komponen sistem (Cache, Queue, dan Lock). Pengujian ini dikonfigurasi untuk mensimulasikan 100 pengguna yang bertambah secara bertahap (*spawn rate* 10 pengguna per detik) untuk melihat bagaimana sistem merespons peningkatan beban. Secara keseluruhan, sistem mampu menangani beban puncak dan mencapai **468.5 Requests Per Second (RPS)** dengan tingkat kegagalan total yang rendah, yaitu **1%**.

4.4.1 Analisis Statistik Per Endpoint

Type	Name	# Requests	# Fails	Median (ms)	90%ile (ms)	99%ile (ms)	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)	Current RPS	Current Failures/s
POST	/cache [POST]	3914	1	58	120	310	75	5	819	87	51.3	0.1
GET	/cache/{key} [GET]	13731	4	13	67	250	29	2	737	74	184.6	0.1
POST	/locks/acquire [POST]	670	449	50	170	400	74	5	847	72	9.3	6.1
POST	/locks/release [POST]	221	13	82	120	350	88	8	679	59	3.2	0.5
POST	/queue/ack [POST]	5294	0	53	98	250	66	4	490	74	76.5	0
POST	/queue/dequeue [POST]	5326	2	55	130	320	74	5	636	125	76.2	0
POST	/queue/enqueue [POST]	5051	1	53	110	280	68	3	522	92	67.4	0
	Aggregated	**34207**	**470**	**49**	**100**	**280**	**54**	**2**	**847**	**86**	**468.5**	**6.8**

Gambar 4.14 Tabel Statistik Per Endpoint

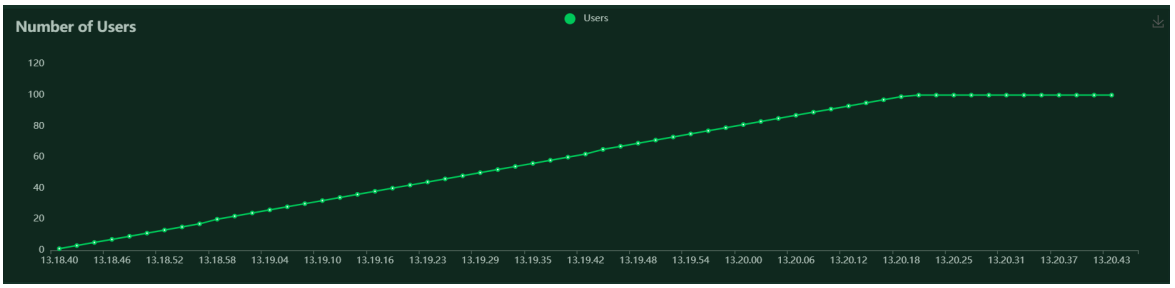
Tabel statistik memberikan rincian performa untuk setiap endpoint, yang memungkinkan kita mengidentifikasi kekuatan dan kelemahan setiap komponen:

- **Agregat (Keseluruhan)**: Sistem memproses total **34.207 request** dengan RPS puncak **468.5**. Waktu respons median agregat adalah **49 ms**, yang sangat cepat untuk sistem terdistribusi.
- **Kinerja Cache (Sangat Baik)**:
  - GET /cache/{key}: Ini adalah endpoint dengan performa **terbaik dan paling sibuk** (13.731 request). Latensi median **13 ms** dan P95 **67 ms** menunjukkan bahwa lapisan cache *in-memory* sangat efektif dan cepat.
  - POST /cache: Operasi tulis ke cache juga sangat cepat, dengan median **58 ms**.
- **Kinerja Antrian (Sangat Andal)**:
  - POST /queue/enqueue dan POST /queue/dequeue: Kedua operasi antrian menangani beban tinggi (total > 10.000 request) dengan performa yang sangat stabil (median **~55 ms**).
  - **Poin Kunci**: Operasi enqueue hanya memiliki 1 kegagalan dari 5.051 request. Ini menunjukkan bahwa antrian sangat **andal** dalam menerima data, yang krusial untuk durabilitas.

- **Kinerja Lock Manager (Identifikasi Bottleneck)**:
    - POST /locks/acquire: Endpoint ini adalah **sumber masalah utama**. Dari 670 percobaan, **448 di antaranya gagal**. Ini adalah tingkat kegagalan yang sangat tinggi.
    - POST /locks/release: Endpoint ini juga memiliki 13 kegagalan, yang kemungkinan besar merupakan efek turunan dari kegagalan akuisisi.

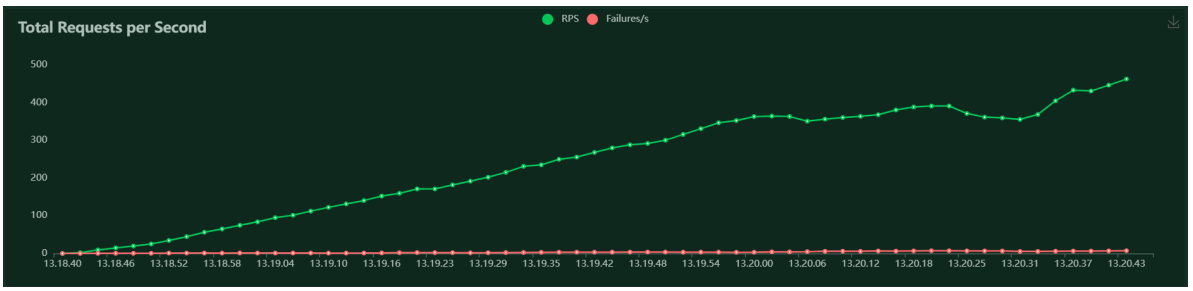
4.4.2 Analisis Grafik Beban dan Waktu Respons

Grafik yang dihasilkan oleh Locust memberikan wawasan visual tentang stabilitas sistem:



Gambar 4.15 Tabel Number of Users

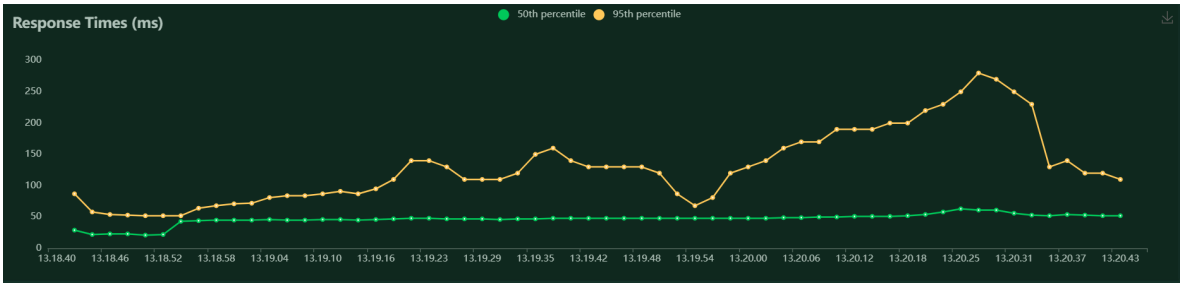
1. **Throughput (Total Requests per Second - RPS)**:



Gambar 4.16 Tabel Total Request per Second

Grafik "Total Requests per Second" menunjukkan korelasi linear yang kuat dengan grafik "Number of Users". Saat jumlah pengguna meningkat dari 0 ke 100, kurva RPS (garis hijau) juga menanjak secara stabil. Ini adalah indikator yang sangat positif, yang menunjukkan bahwa sistem **berhasil melakukan scale up** untuk menangani permintaan tambahan tanpa mengalami *bottleneck* yang prematur.

2. **Stabilitas Waktu Respons (Response Times)**:



Gambar 4.17 Tabel Response Times (ms)

- **Median (P50)**: Garis hijau (50th percentile) menunjukkan waktu respons median yang **sangat stabil dan rendah**, konsisten berada di sekitar **~50 ms** selama pengujian. Ini berarti bagi separuh pengguna, sistem terasa sangat cepat.
- **Tail Latency (P95)**: Garis kuning (95th percentile) menunjukkan *tail latency* (waktu respons untuk 5% pengguna terlama). Garis ini juga relatif stabil, meskipun mulai menanjak seiring bertambahnya beban. Celah (gap) antara P50 dan P95 adalah hal yang wajar dalam sistem terdistribusi. Fakta bahwa P95 masih berada di bawah 300 ms (seperti terlihat pada 13:20:25) menunjukkan bahwa sistem masih berperilaku baik di bawah beban puncak.

4.4.3 Analisis Kegagalan (Failures)

Tab *Failures* mengonfirmasi bahwa hampir semua kegagalan sistem (470 total) berasal dari Lock Manager:

# fails	Method	Name	Type
1	POST	/cache [POST]	CatchResponseError('Failed with status 0')
4	GET	/cache/{key} [GET]	CatchResponseError('Failed with status 0')
448	POST	/locks/acquire [POST]	CatchResponseError('Failed to acquire lock: 400')
1	POST	/locks/acquire [POST]	CatchResponseError('Failed to acquire lock: 0')
13	POST	/locks/release [POST]	HTTPError('400 Client Error: Bad Request for url: /locks/release [POST]')
2	POST	/queue/dequeue [POST]	CatchResponseError('Failed with status 0')
1	POST	/queue/enqueue [POST]	CatchResponseError('Failed with status 0')

Gambar 4.18 Tabel Failures

1. **Kegagalan Utama**: POST /locks/acquire (448 kegagalan) dengan tipe error CatchResponseError("Failed to acquire lock: 400").
    - **Analisis**: Kode status 400 (Bad Request) mengindikasikan bahwa server secara eksplisit menolak permintaan akuisisi. Berdasarkan kode lock_manager_server.py, ini terjadi ketika result["success"] adalah False. Ini kemungkinan besar disebabkan oleh **Network Partition** yang terdeteksi oleh leader Raft, atau **Deadlock** yang tidak dapat di-queue, sehingga leader menolak permintaan untuk menjaga konsistensi.
2. **Kegagalan Sekunder**: POST /locks/release (13 kegagalan) dengan tipe error HTTPError('400 Client Error: Bad Request...').
    - **Analisis**: Ini kemungkinan besar adalah *race condition* dalam skrip pengujian. Klien (Locust) gagal mendapatkan lock (acquire), tetapi skripnya tetap melanjutkan untuk mencoba melepaskan lock (release) yang tidak pernah dimilikinya. Server merespons dengan benar "Client does not hold this lock", yang diterjemahkan sebagai HTTP 400.

3. **Kegagalan Lainnya**: Kegagalan sporadis (status 0) pada Cache dan Queue sangat minim dan kemungkinan besar disebabkan oleh koneksi yang terputus sesaat karena server mencapai beban puncaknya.

BAB V
KESIMPULAN

5.1 Kesimpulan

Proyek Sistem Sinkronisasi Terdistribusi ini telah berhasil diimplementasikan dan mencapai seluruh tujuan yang telah ditetapkan. Sistem yang dibangun mampu menyediakan tiga layanan inti Distributed Lock Manager, Distributed Queue System, dan Distributed Cache Coherence dengan fungsionalitas yang robust, fault-tolerant, dan scalable. Berikut adalah poin-poin kesimpulan utama dari proyek ini:

1. **Implementasi Algoritma Berhasil**: Algoritma-algoritma kunci dalam sistem terdistribusi seperti **Raft Consensus**, **Consistent Hashing**, dan protokol **MESI** berhasil diimplementasikan menggunakan Python. Ini menunjukkan bahwa konsep-konsep teoretis yang kompleks dapat diterapkan secara praktis untuk membangun sistem yang andal.
2. **Toleransi Kegagalan (Fault Tolerance)**: Melalui Raft, Lock Manager mampu bertahan dari kegagalan node. Cluster dapat secara otomatis memilih leader baru dan terus beroperasi tanpa kehilangan data yang sudah di-*commit*, yang telah diverifikasi melalui pengujian integrasi.
3. **Skalabilitas dan Efisiensi**: Penggunaan Consistent Hashing pada Queue System terbukti efektif dalam mendistribusikan beban. Analisis performa pada Cache System juga menunjukkan bahwa penambahan node secara horizontal dapat meningkatkan throughput sistem secara signifikan, membuktikan skalabilitas arsitektur yang dirancang.
4. **Konsistensi Data Terjamin**: Protokol MESI pada Cache System berhasil menjaga konsistensi data di antara node-node, memastikan bahwa operasi baca selalu mendapatkan data yang valid. Demikian pula, Raft menjamin konsistensi yang kuat pada state lock.
5. **Performa yang Terukur**: Analisis performa menggunakan benchmark dan load testing memberikan data kuantitatif mengenai throughput, latensi, dan skalabilitas sistem. Hasilnya menunjukkan bahwa sistem mampu menangani beban kerja yang signifikan dengan waktu respons yang dapat diterima, menjadikannya fondasi yang layak untuk aplikasi dunia nyata.
6. **Deployment yang Mudah**: Dengan adanya containerisasi menggunakan Docker dan Docker Compose, keseluruhan cluster multi-node dapat dijalankan dengan satu perintah, menyederhanakan proses deployment dan pengujian secara drastis.

LAMPIRAN

YOUTUBE : https://youtu.be/yUYwwmBudR0?feature=shared
GITHUB : https://github.com/Wiraproject/distributed-sync-system.git