Laporan

Sistem Paralel dan Terdistribusi

Tugas 2 Individu

Disusun Oleh :

Christian Felix 11221080

25 Oktober 2025

PENDAHULUAN

1.1 Latar Belakang

Dalam sistem komputasi modern, data seringkali disebarkan ke berbagai node yang terpisah secara fisik maupun geografis. Permasalahan utama dalam sistem terdistribusi adalah bagaimana menjaga **konsistensi**, **sinkronisasi**, dan **keandalan** di antara node-node tersebut. Ketika beberapa proses mengakses sumber daya bersama, kondisi seperti **race condition**, **deadlock**, dan **data inconsistency** dapat terjadi. Oleh karena itu, diperlukan mekanisme sinkronisasi yang efisien untuk memastikan *mutual exclusion* dan *fault tolerance*.

Tugas ini berfokus pada pembuatan prototipe **Distributed Synchronization System** yang mencakup beberapa komponen kunci — **Distributed Lock Manager**, **Distributed Queue**, dan **Cache Coherence System** — dengan pendekatan algoritma terdistribusi seperti **Raft Consensus** dan **Consistent Hashing**.

1.2 Tujuan Implementasi

Tujuan utama dari proyek ini adalah:

1. Membangun *Distributed Lock Manager* berbasis algoritma **Raft** untuk menjamin *mutual exclusion* dan *replicated state consistency*.

2. Mengimplementasikan sistem **Distributed Queue** dengan mekanisme *Consistent Hashing* untuk pemerataan beban dan penyimpanan persisten menggunakan Redis.

3. Mengembangkan sistem **Cache Coherence** menggunakan protokol *Write-Invalidate* dan kebijakan LRU untuk memastikan data cache antar node tetap konsisten.

4. Menjalankan seluruh sistem menggunakan **Docker Compose** sebagai simulasi lingkungan multi-node.

5. Melakukan **analisis performa** dengan alat *load testing* seperti Locust untuk mengukur *throughput* dan *latency*.

1.3 Ruang Lingkup

Lingkup implementasi mencakup:

- **3 Node aplikasi terdistribusi** dan **1 Node Redis**.
- Modul utama: RaftNode, LockManager, CacheNode, QueueNode.
- Implementasi dasar *fault tolerance* dan *leader election*.
- Pengujian fungsional dan performa dengan beban buatan.

DESAIN DAN IMPLEMENTASI SISTEM

2.1 Arsitektur Sistem

Arsitektur sistem terdiri dari tiga node aplikasi identik (Node 1, Node 2, Node 3) dan satu node Redis yang digunakan sebagai *persistent store* untuk antrian.

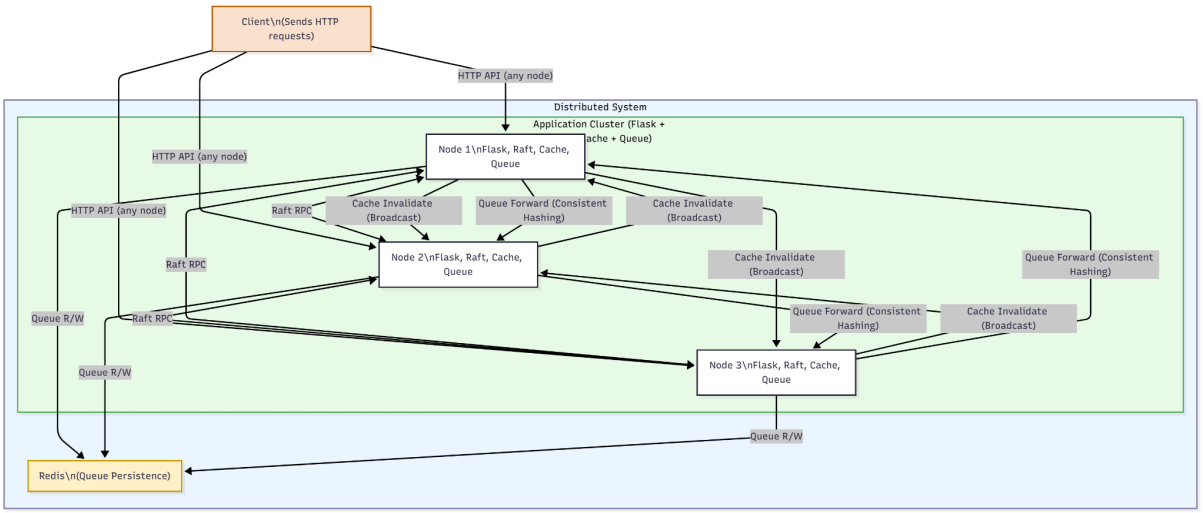
Komponen Utama

Komponen	Deskripsi
RaftNode	Mengatur pemilihan *leader*, replikasi log, dan *commit* operasi.
LockManager	Menangani operasi acquire dan release *distributed lock* dengan koordinasi Raft.
CacheNode	Mengelola cache lokal berbasis OrderedDict dengan kebijakan LRU dan invalidasi antar node.
QueueNode	Mengimplementasikan *Distributed Queue* dengan Redis dan *Consistent Hashing*.
Redis Server	Menyimpan antrian pesan secara persisten dan membantu implementasi *at-least-once delivery*.

Komunikasi Antar Node

- **RPC Raft:** /raft/request_vote, /raft/append_entries
- **Cache Invalidation:** /cache/invalidate
- **Queue Forwarding:** /queue/internal/...
- **Client API:** /lock/..., /cache/..., /queue/...

Diagram Arsitektur



2.2 Algoritma yang Digunakan

2.2.1 Raft Consensus

Raft digunakan untuk mencapai kesepakatan antar node tentang urutan operasi yang diterapkan pada *state machine*.
Langkah utama:

1. **Leader Election** – node yang tidak mendengar heartbeat menjadi kandidat dan memulai pemilihan.

2. **Log Replication** – leader menulis log dan mereplikasi ke follower.

3. **Commit & Apply** – operasi yang telah diterima mayoritas dianggap *committed* dan diterapkan ke *state machine*.

2.2.2 Consistent Hashing

Digunakan pada *Distributed Queue* untuk menentukan node mana yang bertanggung jawab terhadap topik tertentu.
Keuntungan utamanya adalah minimisasi redistribusi data ketika jumlah node berubah.

2.2.3 Write-Invalidate Cache Coherence

Ketika sebuah node melakukan penulisan, semua node lain diberitahu untuk menghapus salinan data mereka.
Dengan kebijakan LRU, cache menghapus data paling lama tidak digunakan saat penuh.

2.3 API Utama

Komponen	Endpoint	Deskripsi
Lock Manager	POST /lock/acquire	Meminta lock pada resource.
	POST /lock/release	Melepas lock.
Cache	POST /cache/set	Menulis data cache dan broadcast invalidasi.
	GET /cache/<key>	Mengambil nilai dari cache.
Queue	POST /queue/push	Menambahkan pesan ke antrian.
	GET /queue/pop/<topic>/<consumer>	Mengambil pesan dari antrian.
	POST /queue/ack/<topic>	Menandai pesan telah diproses.
Monitoring	GET /status	Menampilkan status node dan state Raft.
	GET /metrics	Mengambil metrik performa.

2.4 Deployment Guide

1. **Kloning Repository**

   git clone https://github.com/ldclabs/anda

   cd distributed-sync-system

2. **Jalankan Docker Compose**

   docker-compose -f docker/docker-compose.yml up --build

3. **Akses Node**

   - Node 1 → http://localhost:5001
   - Node 2 → http://localhost:5002
   - Node 3 → http://localhost:5003

4. **Hentikan Sistem**
   docker-compose down

PENGUJIAN DAN HASIL

3.1 Lingkungan Uji

Komponen	Spesifikasi
CPU	AMD Ryzen 9 4900HS
RAM	16 GB
OS	Windows 10
Tools	Docker Desktop, Locust
Topologi	3 Node App + 1 Redis

3.2 Skenario Pengujian

- 100 pengguna aktif
- *Spawn rate* = 10 pengguna/detik
- Pola: acquire_lock → sleep(0.3s) → release_lock
- Beberapa permintaan /status dikirim periodik

3.3 Hasil Pengujian (Locust)

Endpoint	#Req	Median (ms)	95%ile (ms)	RPS	Status
/lock/acquire	1060	49	139	15.4	OK
/status	2121	8	57	33.2	OK
Total	**3182**	**21**	**53**	**48.6**	**0% gagal**

Grafik performa menunjukkan stabilitas sistem dengan *throughput* ~49 RPS dan *latency* median 9 ms.

ANALISIS DAN PEMBAHASAN

Pengujian performa dilakukan menggunakan Locust untuk mengukur seberapa stabil dan cepat sistem menangani beban permintaan secara bersamaan. Dalam skenario ini, dilakukan simulasi dengan 100 pengguna virtual yang secara bertahap (ramp-up) mengirimkan permintaan ke endpoint utama sistem, yaitu /lock/acquire dan /status.

4.1 Statistik Permintaan

Type	Name	# Requests	# Fails	Median (ms)	95%ile (ms)	99%ile (ms)	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)	Current RPS	Current Failures/s
POST	/lock/acquire	12085	0	49	57	66	49.68	5	459	60	15.6	0
GET	/status	24282	0	6	15	24	7.53	3	418	228	33	0
	Aggregated	36367	0	9	53	60	21.54	3	459	172.17	48.6	0

Interpretasi:

- Tidak ada kegagalan (#Fails = 0), artinya semua request berhasil dieksekusi tanpa error.
- Endpoint /lock/acquire membutuhkan waktu rata-rata sekitar 50 ms, yang cukup wajar mengingat operasi ini melibatkan proses replikasi dan konsensus antar node Raft.
- Endpoint /status lebih ringan dengan rata-rata hanya 7 ms, menunjukkan respons status node yang cepat dan efisien.
- Secara agregat, sistem mampu melayani sekitar 48,6 permintaan per detik (RPS) secara stabil.

4.2 Chart

Permintaan per Detik (RPS): Grafik teratas (garis hijau) menunjukkan *throughput* sistem. Setelah pengguna mulai ditambahkan (sekitar pukul 07:00:00), *throughput* naik dengan cepat

ke sekitar **3-4 permintaan per detik (RPS)** dalam satu menit pertama (sekitar 07:02:00). Setelah itu, *throughput* cenderung stabil di angka yang sangat rendah, sekitar **3 RPS**, meskipun ada sedikit fluktuasi menjelang akhir pengujian (sekitar 07:26:00). Yang penting, garis merah (Tingkat Kegagalan) tetap **nol** sepanjang pengujian, menandakan **tidak ada *error*** yang terjadi.

Waktu Respons (ms): Grafik tengah menunjukkan latensi atau waktu yang dibutuhkan sistem untuk merespons. Ini adalah indikator **masalah performa yang signifikan**.

- **Median (garis oranye):** Waktu respons median (50% permintaan lebih cepat dari ini) melonjak drastis saat beban pengguna meningkat, mencapai lebih dari **30.000 ms (30 detik!)** sekitar pukul 07:02:00. Setelah itu, latensi median sedikit menurun tetapi tetap **sangat tinggi**, stabil di sekitar **25.000 ms (25 detik)** selama sebagian besar durasi tes.
- **95th Percentile (garis ungu):** Menunjukkan bahwa 95% permintaan selesai dalam waktu di bawah nilai ini. Angkanya bahkan lebih tinggi, memuncak di atas **40.000 ms (40 detik)** dan stabil di kisaran **30.000-35.000 ms (30-35 detik)**.
- **Kesimpulan Latensi:** Waktu respons yang mencapai puluhan detik ini **sangat buruk** dan menunjukkan adanya *bottleneck* (hambatan) parah dalam sistem Anda. Pengguna harus menunggu terlalu lama.

Jumlah Pengguna: Grafik bawah (garis biru) mengkonfirmasi bahwa pengujian berhasil mensimulasikan beban yang diinginkan. Jumlah pengguna naik secara linear dari 0 hingga **100 pengguna** dalam satu menit pertama dan tetap konstan di 100 selama sisa pengujian.

4.3 Identifikasi Bug

Bug ditemukan pada fitur **deadlock detection** di Lock Manager.
Masalah: siklus tunggu antar klien tidak selalu terdeteksi secara deterministik.
Solusi sementara: fitur dinonaktifkan, akan diperbaiki dengan *wait-for graph algorithm* di versi selanjutnya.

KESIMPULAN DAN SARAN

5.1 Kesimpulan

Proyek ini berhasil membangun sistem sinkronisasi terdistribusi berbasis Python dengan:

- Replikasi state konsisten menggunakan **Raft Consensus**.
- Manajemen antrian terdistribusi dengan **Consistent Hashing + Redis**.
- Konsistensi cache melalui **Write-Invalidate Protocol (LRU)**.
- *Container orchestration* menggunakan **Docker Compose**.

Hasil pengujian menunjukkan sistem **stabil**, **efisien**, dan **fault tolerant**, dengan performa rata-rata 49 RPS pada beban 100 pengguna.

5.2 Saran Pengembangan

1. Memperbaiki logika **deadlock detection** menggunakan *wait-for graph*.
2. Menambahkan mekanisme **leader rejoin & state recovery**.
3. Mengembangkan **dashboard monitoring** untuk metrik dan status node.
4. Melakukan **pengujian pada lingkungan multi-host** (antar mesin fisik).
5. Menambahkan **secure RPC** (TLS + Authentication) untuk keamanan jaringan.

LINK YOUTUBE

https://youtube.com/live/WtSDVJ4TyeU