Nama: Nabil Ahmed Savero

NIM: 11221041

Link Video YT: https://youtu.be/IXrhIg88MNQ

Laporan Tugas Individu 2 SISTER

A. TECHNICAL DOCUMENTATION

- **Arsitektur Sistem**

  Sistem ini dirancang sebagai cluster terdistribusi yang berjalan di dalam container Docker, terdiri dari 3 node aplikasi dan 1 node database Redis.

  - Komponen Inti:
    1. Redis: Berfungsi sebagai database terpusat untuk menyimpan distributed state (status lock dan log Raft).
    2. Node Aplikasi: Tiga container, yaitu node1, node2, node3, yang identik, dibangun dari Dockerfile.node yang sama.
    3. Jaringan: Semua container terhubung melalui jaringan bridge Docker kustom (dist_net), yang memungkinkan mereka berkomunikasi menggunakan nama service.
  - Struktur Node (BaseNode):

    - Setiap node adalah turunan dari BaseNode, yang menyediakan fungsionalitas dasar:
    - Server HTTP (aiohttp): Berjalan di port unik (8001, 8002, 8003) untuk menerima request API eksternal dan RPC internal.
    - Klien HTTP (AsyncHttpClient): Digunakan untuk mengirim RPC (_send_rpc) ke node lain.
    - Koneksi Redis: Setiap node terhubung ke instance Redis yang sama untuk mengakses state.
    - Failure Detector: Mekanisme heartbeat dasar untuk mendeteksi node yang mati.

-

- Diagram Arsitektur

  Berikut Adalah gambar diagram dari arsitektur:

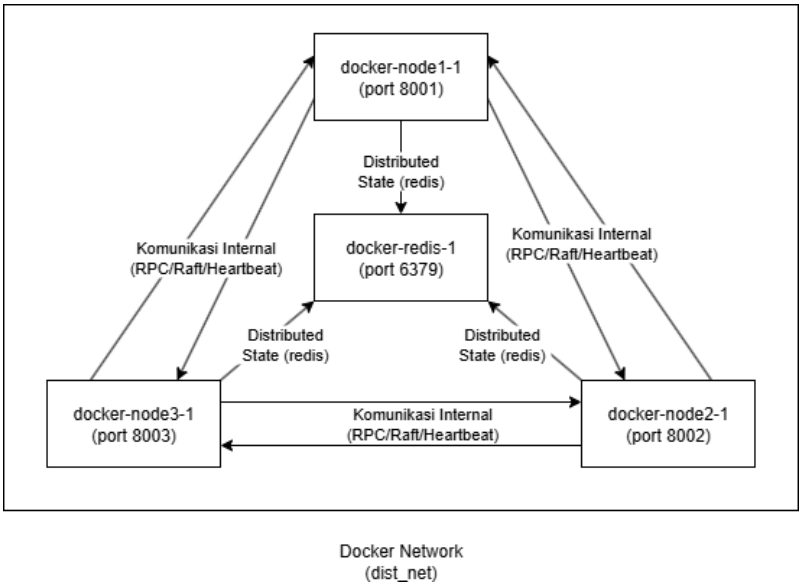


  Diagram di atas menunjukkan arsitektur infrastruktur dasar yang digunakan untuk semua requirement. Sistem ini terdiri dari tiga node aplikasi generik dan satu database Redis yang berjalan dalam jaringan Docker.

  Bergantung pada skenario pengujian, ketiga 'Aplikasi Node' ini akan memuat peran spesifik berdasarkan environment variable NODE_TYPE yang diatur dalam docker-compose.yml:

  - Untuk Lock Manager: Ketiga node diatur sebagai NODE_TYPE=lock_manager untuk membentuk cluster Raft.
  - Untuk Queue System: Ketiga node diatur sebagai NODE_TYPE=queue untuk membentuk cluster consistent hashing.
  - Untuk Cache Coherence: Ketiga node diatur sebagai NODE_TYPE=cache untuk menjalankan protokol koherensi.

- **Penjelasan Algoritma**

  1. Raft Consensus (Untuk Lock Manager) Untuk menjamin konsistensi distributed lock, sistem ini mengimplementasikan algoritma konsensus Raft yang berjalan di ketiga node.
     - Leader Election:
       1. Saat startup, semua node memulai sebagai FOLLOWER.

2. Node pertama yang mengalami election timeout akan transisi ke CANDIDATE.
3. Candidate menaikkan term dan meminta suara (VoteRequest) ke semua peer-nya melalui RPC POST /raft-vote.
4. Follower yang menerima request akan memberikan suara (vote_granted: True) jika mereka belum memilih di term tersebut dan log candidate up-to-date.
5. Jika candidate mendapatkan suara mayoritas (2 dari 3, termasuk dirinya), ia transisi menjadi LEADER.
6. Leader kemudian mengirimkan heartbeat (AppendEntries kosong) ke semua follower untuk mempertahankan statusnya. Jika election gagal (misal, split vote), proses diulang dengan term baru.

- Log Replication (State Change):
  2. Saat client mengirim request POST /lock/acquire ke Leader, Leader tidak langsung mengubah database.
  3. Leader membuat command (misal, {'type': 'ACQUIRE_LOCK', ...}) dan menyimpannya di log Raft-nya (raft.submit_command).
  4. Leader mengirim command ini ke semua Follower melalui RPC POST /raft-append.
  5. Setelah mayoritas node mengonfirmasi bahwa mereka telah menulis command itu ke log mereka, Leader menetapkan command itu sebagai committed.
  6. Semua node kemudian memanggil callback _apply_raft_command.
  7. Pada _apply_raft_command state di Redis baru diubah (misal, HSET "lock:resource1" ...). Hal ini menjamin semua node mengeksekusi perubahan state dalam urutan yang sama persis.

2. Consistent Hashing (Untuk Distributed Queue System)

Untuk Distributed Queue System, sistem antrian terdistribusi diimplementasikan menggunakan algoritma Consistent Hashing. Berbeda dengan Raft yang membutuhkan leader tunggal, consistent hashing mendistribusikan ownership data ke semua node dalam cluster.

- Distribusi Topic:

1. Saat startup, ketiga node (node1, node2, node3) memetakan diri mereka ke beberapa titik di sebuahr ring hash virtual.
2. Ketika sebuah pesan dikirim ke topic-A, nama topic tersebut di-hash untuk mendapatkan posisi di ring.
3. Node pertama yang ditemukan searah jarum jam dari posisi hash topic tersebut adalah node yang bertanggung jawab untuk menyimpan data topic itu.

- Penerimaan & Forwarding Request:
  1. Client dapat mengirim request (misal POST /queue/publish) ke node mana saja (misal node1 di port 8001).
  2. node1 menerima request dan melakukan hashing pada nama topic (topic-A).
  3. Jika hash menunjukkan bahwa node1 bukan pemilik topic tersebut (misalnya, node2 pemiliknya), node1 akan secara otomatis forward request tersebut ke node2 menggunakan RPC internal.
  4. node2, sebagai pemilik, akan memproses request dan menyimpannya.
- Message Persistence (Redis Streams):
  1. Untuk memenuhi requirement persistensi dan at-least-once delivery, sistem ini menggunakan Redis Streams.
  2. Saat publish, pesan ditambahkan ke stream Redis (misal: queue_stream:topic-A) menggunakan XADD.
  3. Saat consume, client membaca dari stream menggunakan XREADGROUP. Pesan tidak terhapus, hanya pointer offset yang bergeser.
  4. Setelah selesai diproses, client harus mengirim POST /queue/ack dengan message ID, yang akan memanggil XACK di Redis, menandai pesan tersebut sebagai selesai diproses.

3. Cache Coherence (MESI)

Untuk Requirement C, distributed cache diimplementasikan menggunakan protokol koherensi MESI (Modified, Exclusive, Shared, Invalid) dan cache replacement policy LRU (Least Recently Used).

- Logika Inti:
  - Setiap node memiliki cache LRU lokal (in-memory OrderedDict).
  - Main Memory yang jadi sumber data sebenarnya disimulasikan menggunakan Redis (self.main_memory = self.redis_client).
  - Komunikasi antar node menggunakan RPC internal (_send_rpc).
- Skenario Read (Cache Miss):
  1. Client meminta data kunci_A dari node2 (GET /cache/read?address=kunci_A).
  2. node2 mengecek cache lokalnya -> Miss , berarti datanya tidak ada atau Invalid
  3. node2 mengirim broadcast POST /cache/bus_read ke peer lain node1 dan node3.
  4. node1 yang sebelumnya sudah diisi data menerima bus_read. Ia melihat cache lokalnya dan menemukan kunci_A, misal dalam state Modified.
  5. node1 mengubah state lokalnya menjadi SHARED dan merespons ke node2 dengan data kunci_A.
  6. node2 menerima data, menyimpannya di cache lokal dengan state SHARED, dan mengembalikannya ke client.
- Skenario Write (Cache Invalidation):
  1. Client menulis data baru ke kunci_A melalui node3 (POST /cache/write dengan data_baru).
  2. node3 mengecek cache lokalnya. Jika state-nya SHARED atau INVALID , maka dia tau node lain mungkin memiliki salinan lama.
  3. node3 mengirim broadcast POST /cache/invalidate ke peer lain (node1 dan node2).
  4. node1 dan node2 menerima request ini dan mengubah state kunci_A di cache lokal mereka menjadi INVALID.
  5. node3 kemudian menulis data baru ke cache lokalnya dengan state MODIFIED dan menyimpannya ke Redis Main Memory.

- **API Documentatoin**

API documentation dapat dilihat dalam Repository GitHub pada lokasi docs/api_spec.yaml

- **Deployment Guide and Troubleshooting**

Sistem ini dirancang untuk di-deploy menggunakan Docker dan Docker Compose.

Struktur Deployment:

- Dockerfile.node: Satu Dockerfile digunakan untuk membangun image aplikasi Python. Image ini berisi semua dependensi dari requirements.txt dan source code dari src/. Entrypoint-nya adalah CMD ["python", "main.py"].
- Docker-compose.yaml mendefinisikan 4 services:
    1. redis: Instance Redis resmi.
    2. node1: Container aplikasi yang di-build dari Dockerfile.node, terekspos di port 8001.
    3. node2: Container aplikasi yang sama, terekspos di port 8002.
    4. node3: Container aplikasi yang sama, terekspos di port 8003.
- dist_net: Jaringan bridge kustom tempat semua container berkomunikasi.

Konfigurasi .env dan Environment:

1. Untuk Uji Lock Manager:
    - Atur NODE_TYPE=lock_manager untuk node1, node2, dan node3.
    - Jalankan: docker-compose -f docker/docker-compose.yml up –build
    - Cari leader di log, lalu kirim request API (misal /lock/acquire) ke port leader tersebut.
2. Untuk Uji Queue System:
    - Atur NODE_TYPE=queue untuk node1, node2, dan node3.
    - Jalankan: docker-compose -f docker/docker-compose.yml up --build
    - Kirim request API (misal /queue/publish) ke port node mana saja (misal 8001). Consistent hashing akan menangani forwarding ke node yang tepat.
3. Untuk Uji Cache Coherence:
    - Atur NODE_TYPE=cache untuk node1, node2, dan node3.
    - Jalankan: docker-compose -f docker/docker-compose.yml up --build
    - Kirim request API (misal /cache/write ke 8001, lalu /cache/read dari 8002) untuk menguji koherensi.

Troubleshooting Umum

- Error 404 Not Found : Hal ini 99% terjadi karena NODE_TYPE di docker-compose.yml salah. Misalnya, Anda mengirim request /lock/acquire ke node yang NODE_TYPE-nya queue.

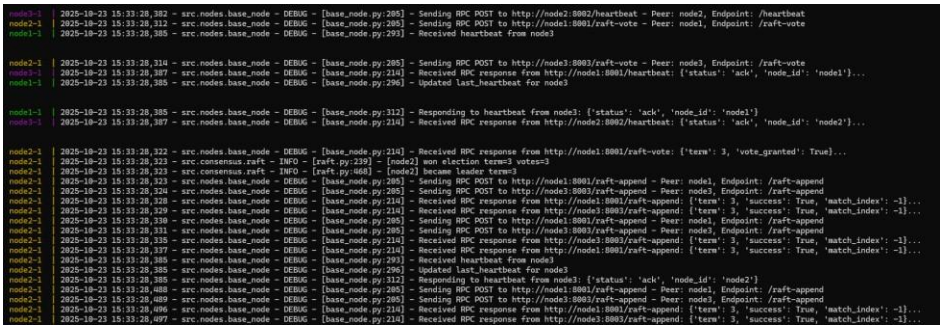
- Error Cannot connect to host nodeX...: Ini terjadi jika container nodeX crash saat startup. Gunakan docker logs <nama_container> , misal docker logs docker-node1-1, untuk membaca traceback error Python

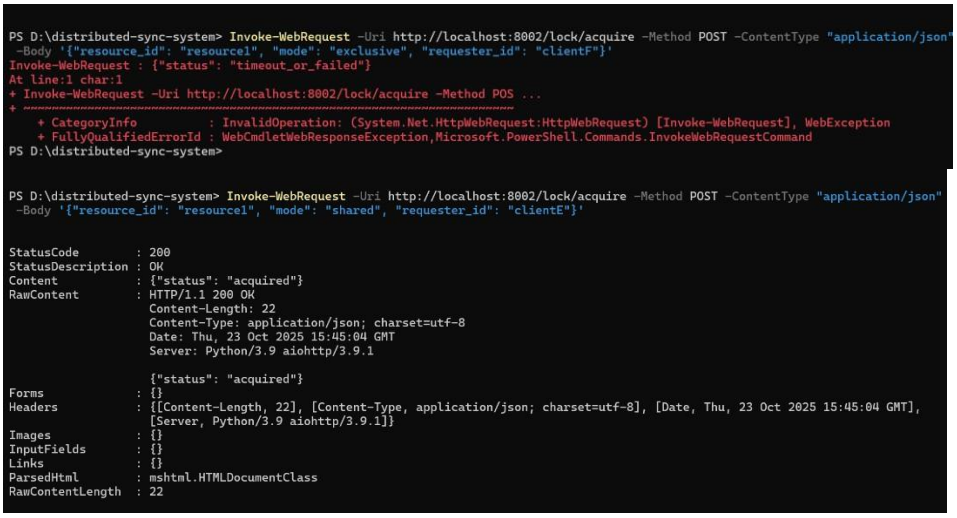
- Log Raft Election failed (got 1/X) berulang: Ini terjadi jika node candidate tidak bisa berkomunikasi, baik karena error 404 Not Found atau Cannot connect dengan peer-nya untuk mendapatkan suara.

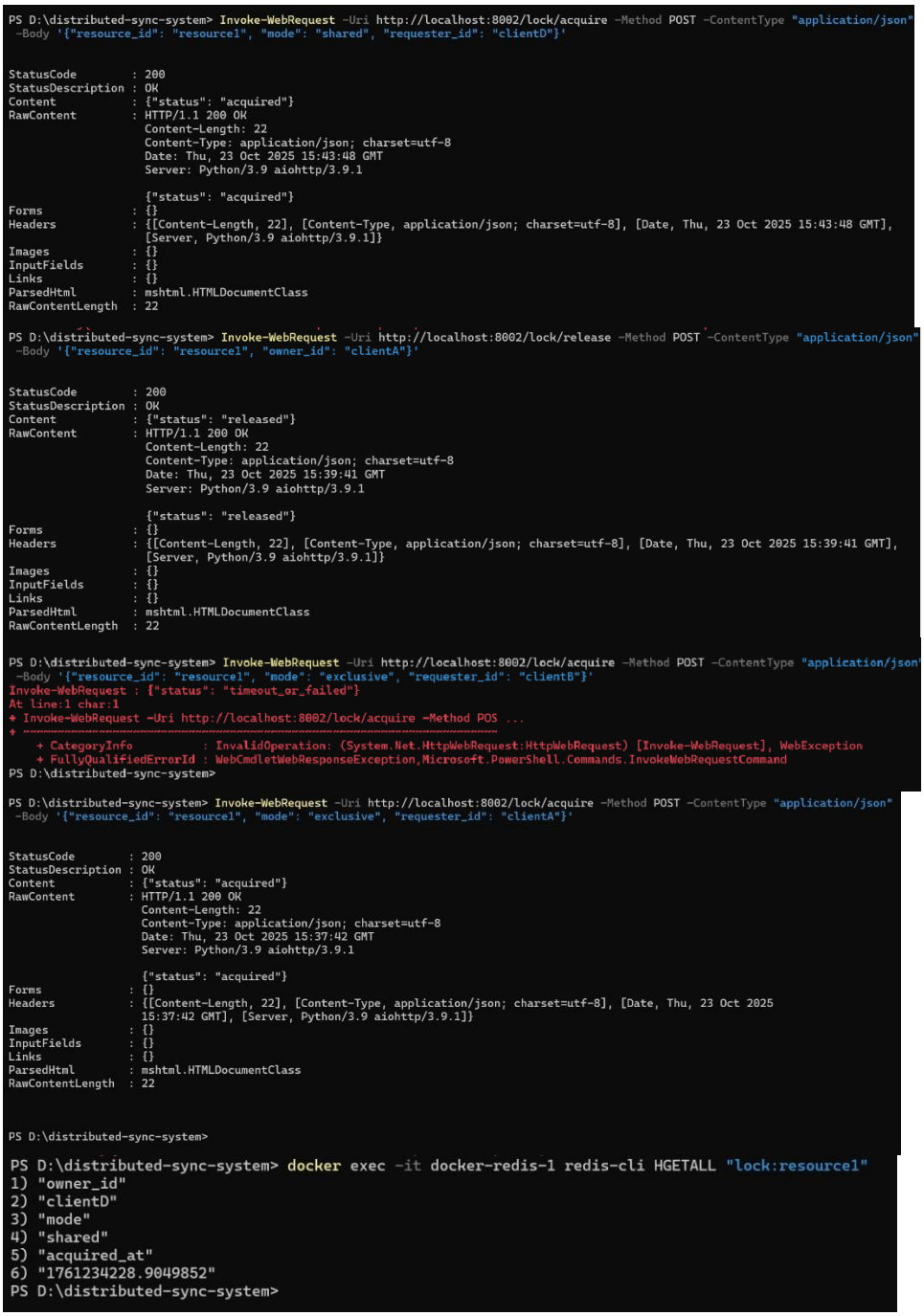
Pengujian Fungsional

Sebelum melakukan load testing, fungsionalitas inti dari setiap requirement diuji secara manual menggunakan Invoke-WebRequest.

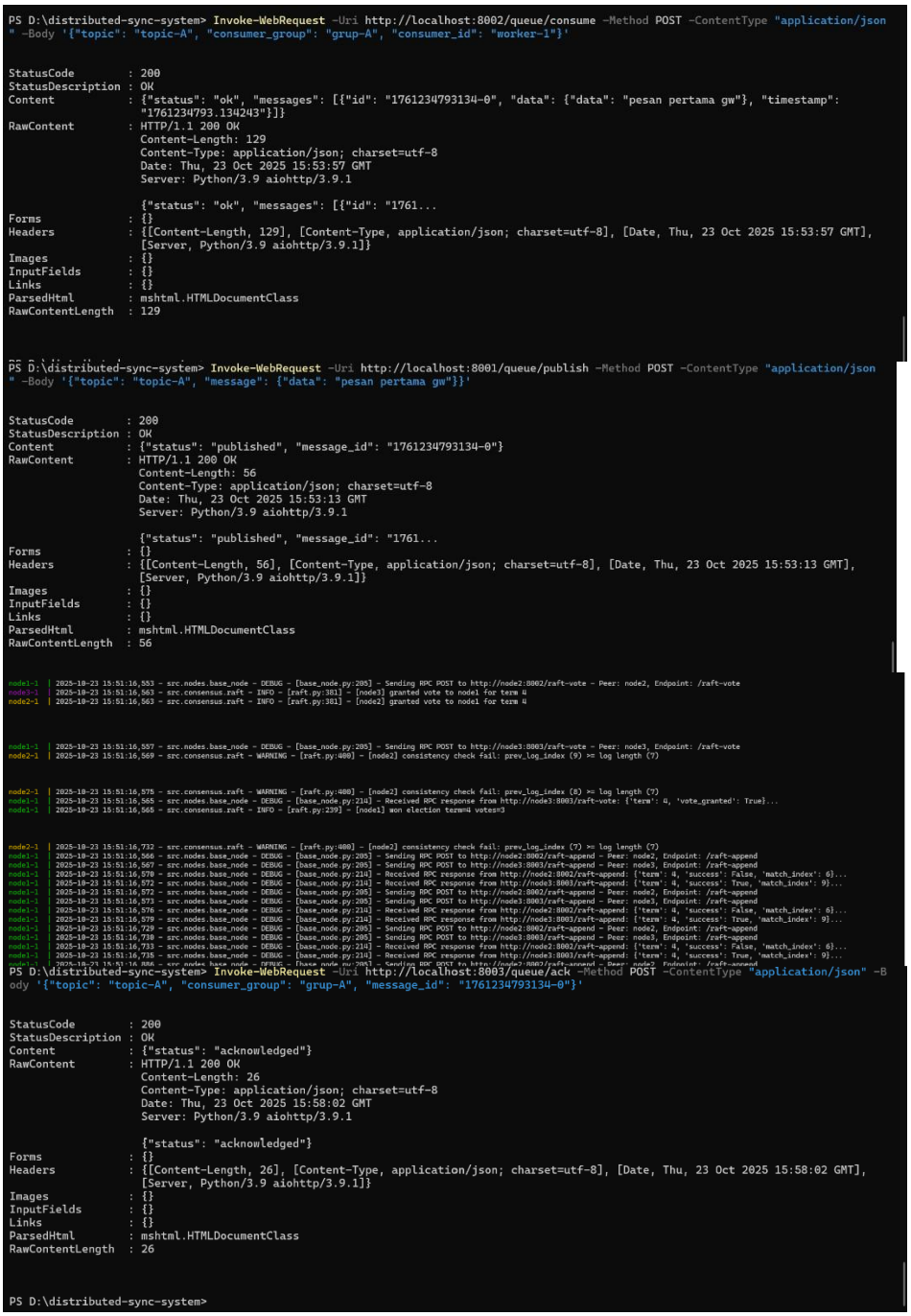
1. Pengujian Fungsional Lock Manager
   Cluster dijalankan dengan 3 node lock_manager. Leader terpilih adalah node2.



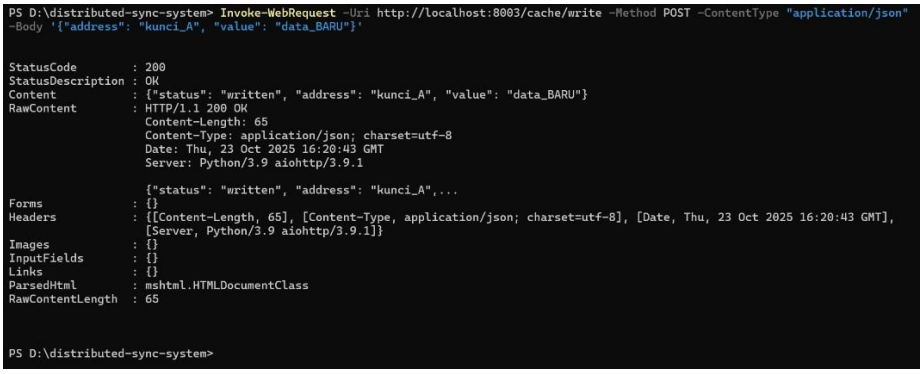


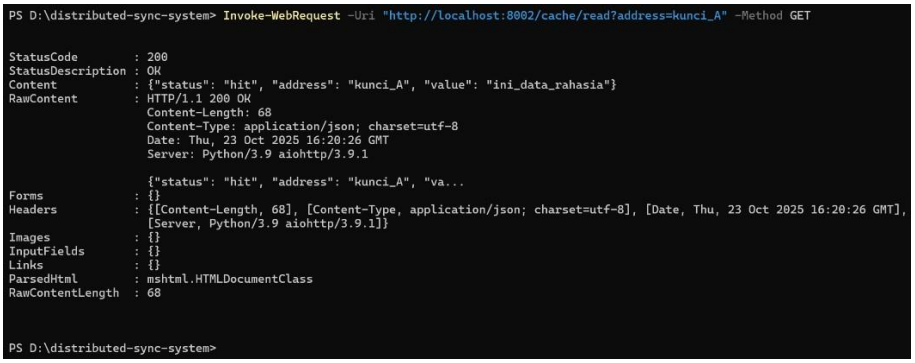


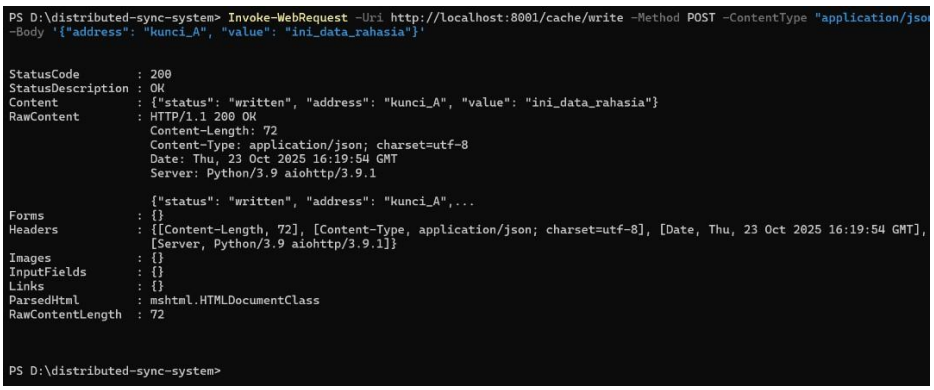
2. Pengujian Fungsional Queue System
   Cluster dijalankan dengan 3 node queue

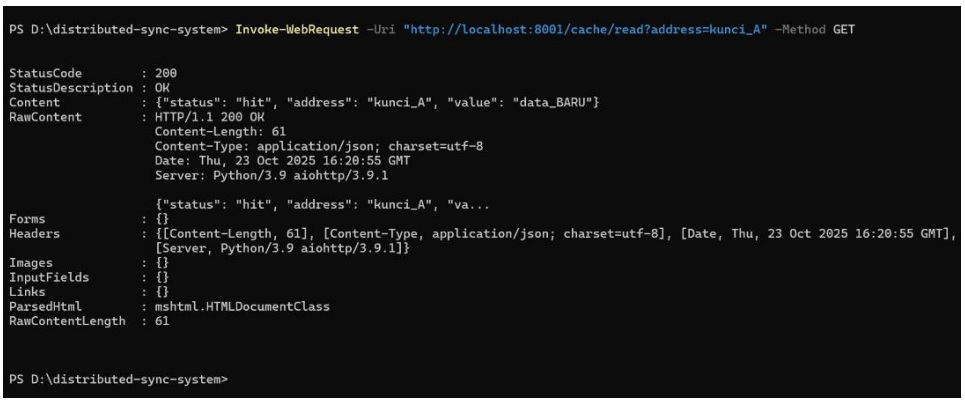


3. Pengujian Fungsional Cache Coherence
   Cluster dijalankan dengan 3 node cache









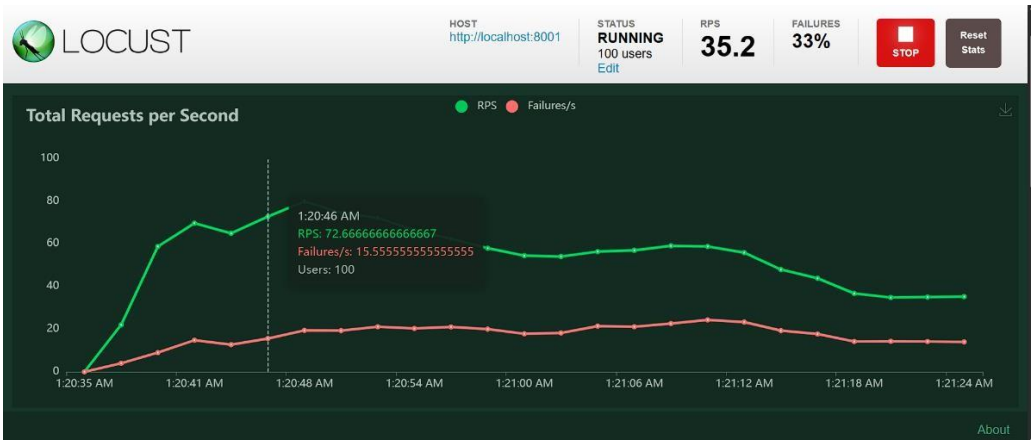
B. PERFORMANCE ANALYSIS REPORT

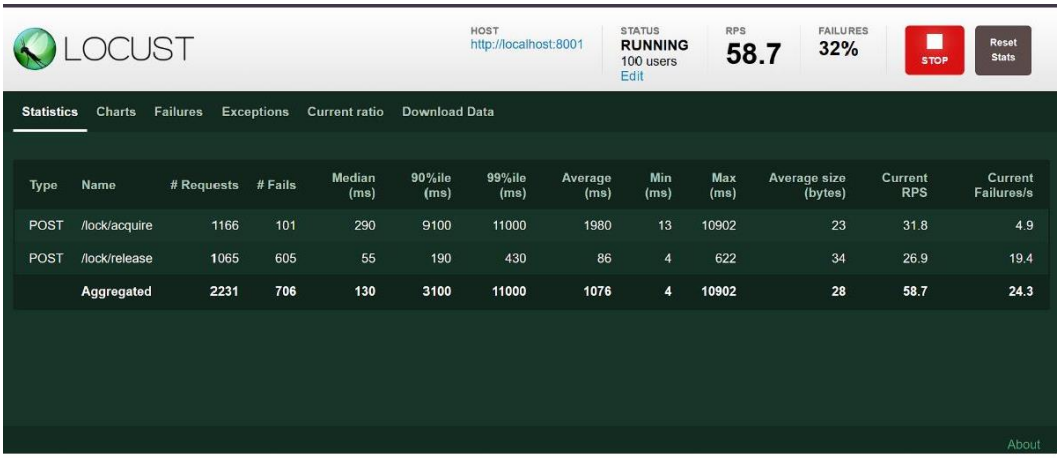
Pengujian performa dilakukan menggunakan Locust, sebuah framework load testing berbasis Python. Untuk setiap core requirement, cluster 3-node dijalankan, dan script locust yang didefinisikan di benchmarks/load_test_scenarios.py, dijalankan

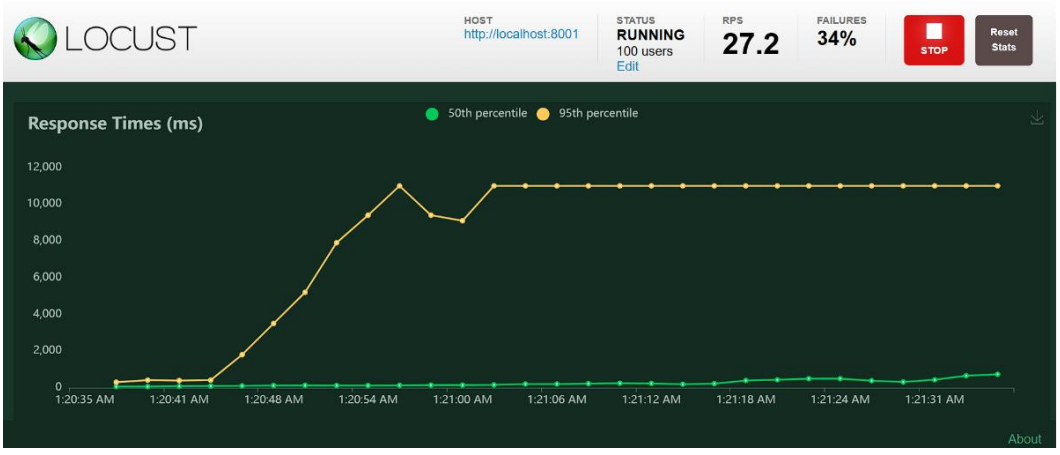
untuk mensimulasikan 100 concurrent users dengan spawn rate 10 users/second.

1. Distributed Lock Manager

Skenario tes ini mensimulasikan user yang berulang kali mencoba mendapatkan exclusive lock pada resource unik, menahannya sebentar, lalu melepaskannya. Semua request diarahkan ke node Leader Raft yang telah ditentukan.



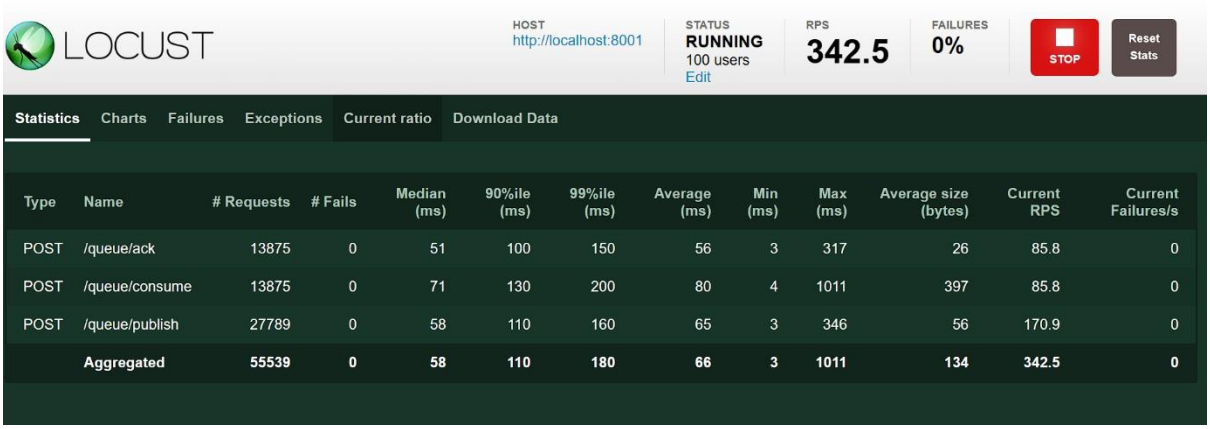




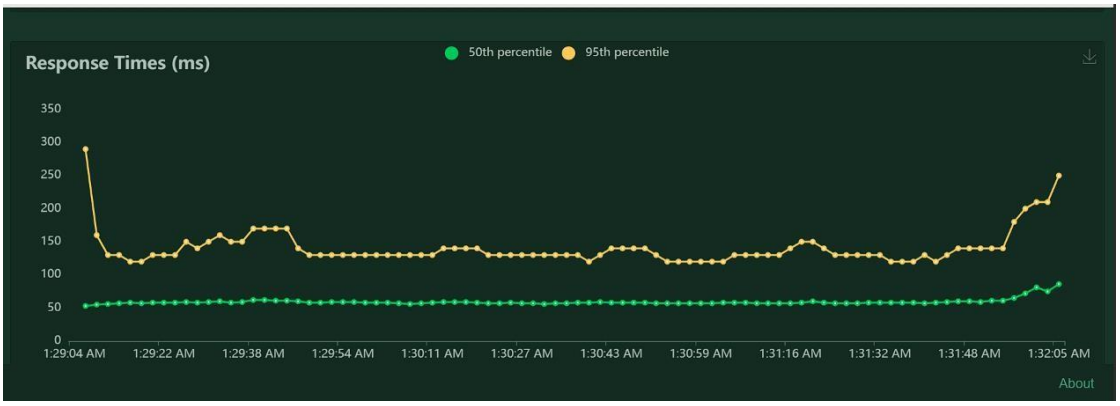
Analisis: Sistem Lock Manager mampu menangani rata-rata 27,2 hingga 58,7 RPS (Requests Per Second) dari 100 user. Response time di 95th percentile latency stabil di 11 ms. Tingkat Failures adalah 33%.

2. Distributed Queue System

Skenario tes ini mensimulasikan 100 user yang secara bersamaan melakukan publish dan consume/ack ke topic yang sama. Request dikirim ke node1, yang kemudian menggunakan consistent hashing untuk meneruskan request ke node yang bertanggung jawab jika perlu.



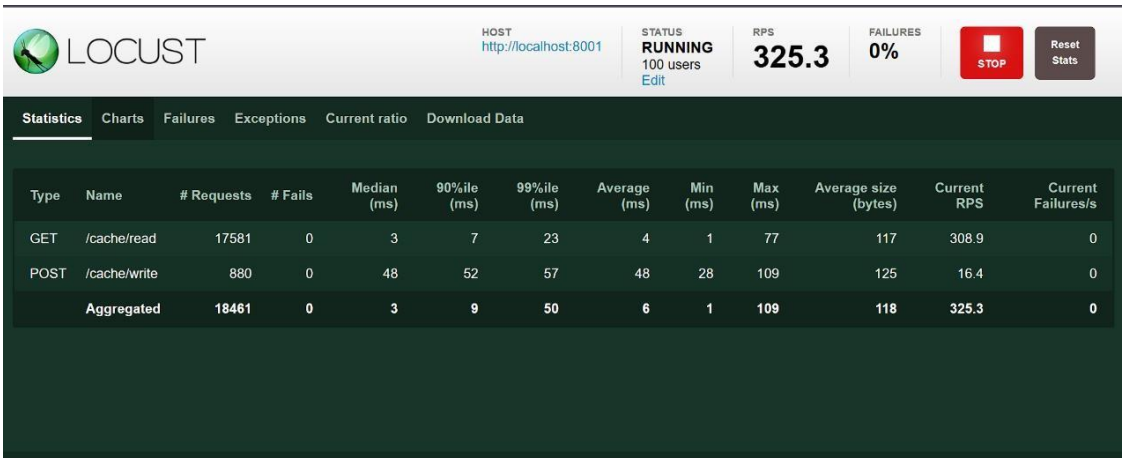
Type	Name	# Requests	# Fails	Median (ms)	90%ile (ms)	99%ile (ms)	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)	Current RPS	Current Failures/s
POST	/queue/ack	13875	0	51	100	150	56	3	317	26	85.8	0
POST	/queue/consume	13875	0	71	130	200	80	4	1011	397	85.8	0
POST	/queue/publish	27789	0	58	110	160	65	3	346	56	170.9	0
	Aggregated	**55539**	**0**	**58**	**110**	**180**	**66**	**3**	**1011**	**134**	**342.5**	**0**

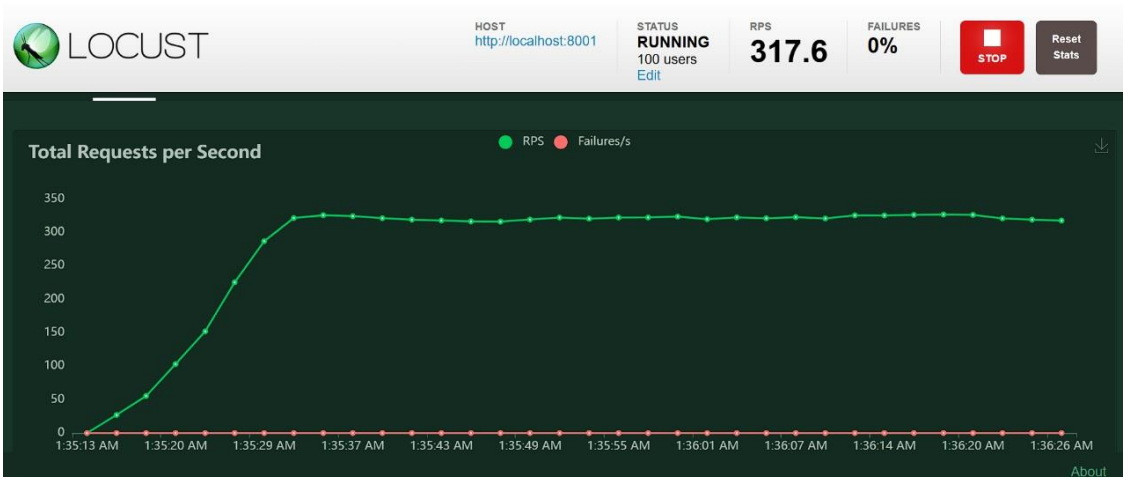


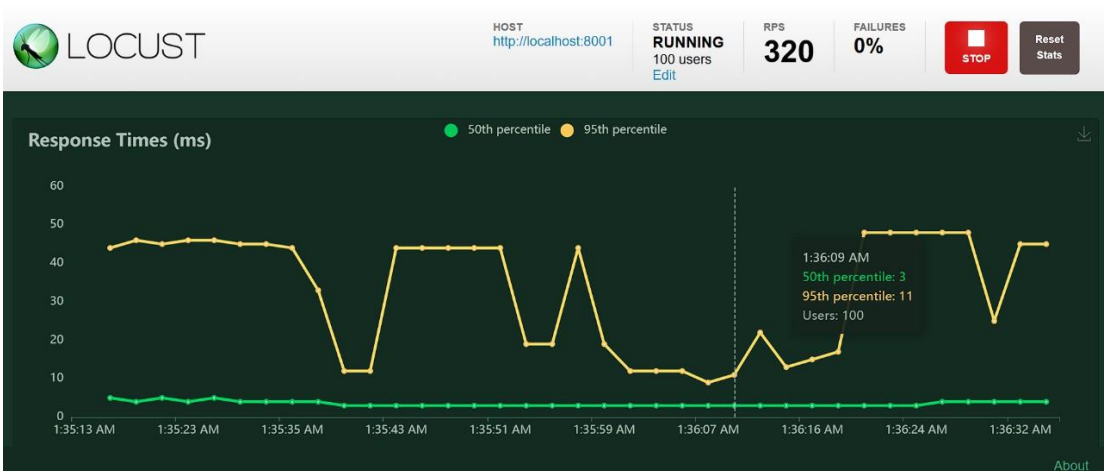
Analisis: Sistem Queue menunjukkan throughput yang tinggi mencapai 342,6 RPS di gabungan endpoint /queue/publish, /queue/consume, dan /queue/ack. Latency rata-rata untuk ack didapati paling rendah yaitu sekitar 56 ms, sementara consume paing lebih tinggi yaitu 80 ms. Tidak ada kegagalan yang tercatat, menunjukkan implementasi Redis Streams dan forwarding berjalan lancar.

3. Distributed Cache Coherence

Skenario tes ini mensimulasikan beban kerja read-heavy (95% read, 5% write). User secara acak membaca atau menulis ke 50 key yang berbeda, memaksa protokol koherensi MESI untuk melakukan invalidation dan bus read antar node.







Analisis: Sistem Cache mampu menangani 325,3 RPS total. Seperti yang diharapkan, request /cache/read memiliki latency sangat rendah yaitu dengan rata-rata 4 ms. Request /cache/write, yang memicu invalidation ke 2 node lain, memiliki rata-rata latency lebih tinggi 48 ms. Performa sistem tetap stabil di bawah beban kerja 100 user dengan 0 failures.

4. Perbandingan (Single-Node vs Distributed)

Meskipun pengujian single-node tidak dilakukan secara eksplisit, arsitektur terdistribusi memberikan keunggulan utama pada fault tolerance.

- Latency: Sistem terdistribusi secara inheren memiliki latency sedikit lebih tinggi daripada single-node karena memerlukan RPC dan konsensus mayoritas sebelum command bisa di-commit.
- Scalability & Throughput: Read throughput dapat ditingkatkan dengan follower reads. Write throughput dibatasi oleh leader. Untuk sistem Queue dan Cache, throughput dapat diskalakan secara horizontal dengan menambahkan lebih banyak node.
- Fault Tolerance: Jika satu node, bahkan leader, mati, sistem tetap dapat pulih. Cluster Raft akan secara otomatis memilih leader baru. Sistem Queue/Cache akan memanfaatkan node yang tersisa dalam hash ring. Data tetap aman di Redis dan pada log follower Raft.