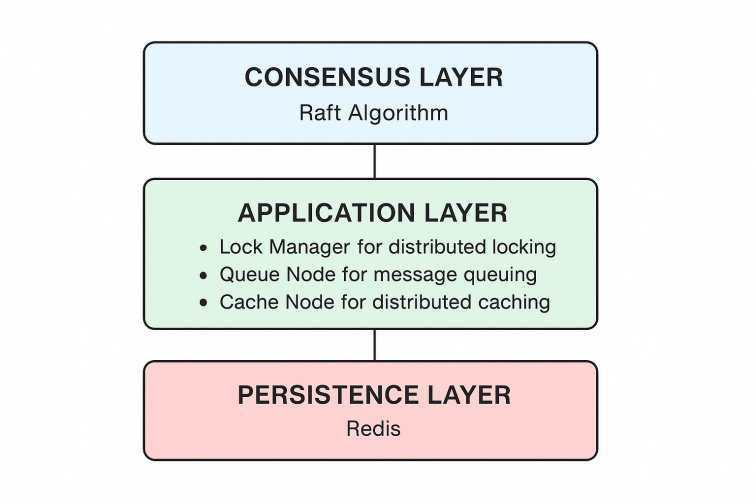
Nama : Febriani
NIM : 11221008

Tugas Individu 2 (Distributed Synchronization System)
Sistem Paralel dan Terdistribusi

LINK Youtube : https://youtu.be/9F4VpUZspRU

1. Arsitektur
   Tiga lapisan utama yang membentuk sistem ini adalah:



a. CONSENSUS LAYER (Lapisan Konsensus)
   Lapisan Konsensus merupakan fondasi dari sistem terdistribusi ini.
   Fungsi Utama: Bertanggung jawab penuh untuk menjaga konsistensi data (strong consistency) di antara semua Node (server) dalam cluster.
   Implementasi: Menggunakan Raft Algorithm. Raft memastikan bahwa setiap perintah yang diterima sistem (seperti operasi lock, enqueue, atau cache set) dicatat sebagai Log dan direplikasi ke mayoritas Node sebelum dianggap "komit" dan dieksekusi. Hal ini menjamin bahwa seluruh cluster selalu berada dalam state yang sama dan disepakati.

b. APPLICATION LAYER (Lapisan Aplikasi)
   Lapisan Aplikasi dibangun di atas Lapisan Konsensus dan menyediakan fungsionalitas inti yang diperlukan untuk sinkronisasi terdistribusi.. Lapisan ini menerima permintaan dari client (seperti operasi lock atau enqueue) dan menerjemahkannya menjadi perintah yang dikirim ke Lapisan Konsensus. Lapisan ini terdiri dari tiga komponen utama:
   - Lock Manager: Bertugas untuk distributed locking (penguncian terdistribusi). Fungsinya adalah mengelola hak akses ke resource bersama, memastikan hanya satu atau beberapa client yang dapat memanipulasi data pada waktu tertentu (misalnya, exclusive atau shared lock).

- Queue Node: Bertugas untuk message queuing (antrian pesan). Komponen ini memungkinkan Producer untuk mengirim pesan ke antrian dan Consumer untuk mengambilnya, menjamin pesan terkirim secara andal di lingkungan terdistribusi.
- Cache Node: Bertugas untuk distributed caching (caching terdistribusi). Komponen ini menyediakan penyimpanan data sementara berkecepatan tinggi yang konsisten di seluruh cluster.

c. PERSISTENCE LAYER (Lapisan Persistensi)

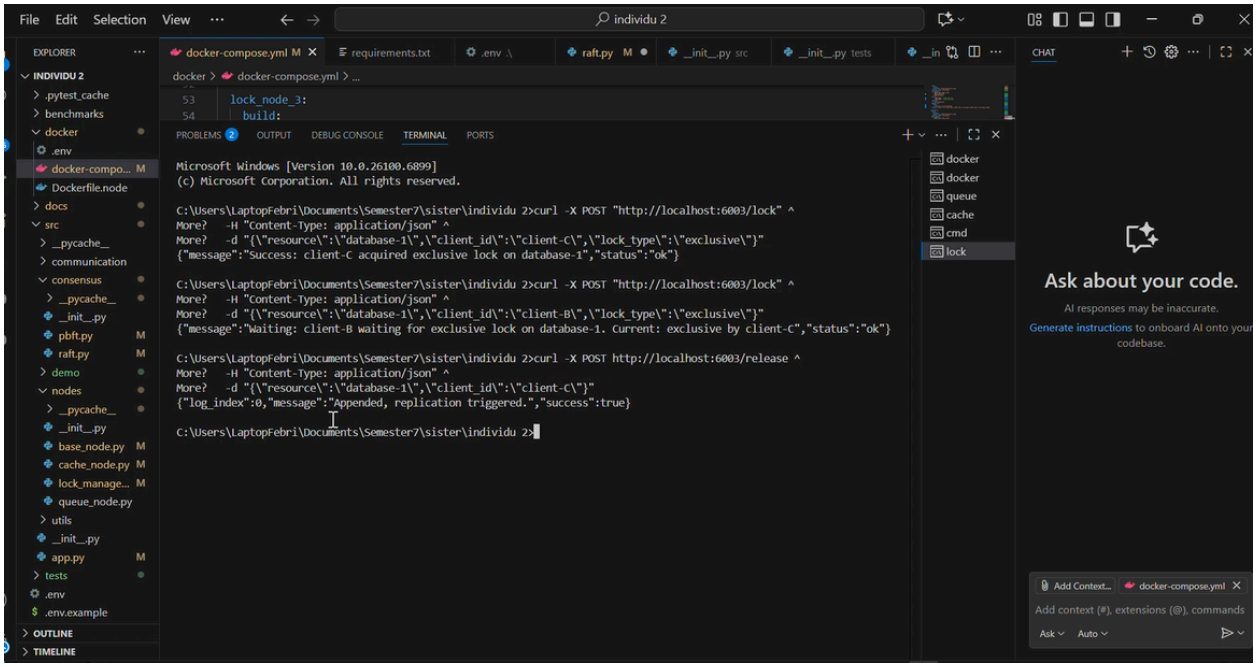
Lapisan Persistensi adalah tempat data aktual disimpan secara tahan lama dan cepat.

Implementasi: Lapisan ini menggunakan Redis. Redis dipilih karena performanya yang tinggi sebagai in-memory data structure store, ideal untuk mendukung kebutuhan kecepatan tinggi dari Lock Manager, Queue Node, dan Cache Node

2. Distributed Lock Manager

Distributed Lock Manager adalah salah satu sistem terdistribusi yang merupakan proses pengambilan lock. Dengan menggunakan Distributed Lock Manager dapat mengatur hak akses terhadap suatu sumber daya agar tidak terjadi konflik ketika beberapa node mencoba mengakses sumber daya secara bersamaan.

Gambar dibawah ini adalah tampilan kode basic lock Acquisition.



Pada gambar diatas adalah exclusive lock, yang mana hanya satu node atau client yang dapat memiliki hak akses secara penuh terhadap suatu resource pada satu waktu. Jika terdapat node yang ingin mengakses resource yang sudah diakses lebih dulu oleh node lain, maka node yang lain tidak dapat mengakses dan statusnya akan menunggu. Seperti pada gambar diatas, bahwa client-c ingin mengakses exclusive lock pada resource database-1. Setelah itu dikirimkan, response menunjukkan succes yang berarti client-C mendapatkan akses dari database-1. Kemudian client-B ingin mengakses exclusive lock pada database-1, namun statusnya menjadi waiting karena pada database-1 sudah diakses pada client-C. Dengan menggunakan exclusive

lock, data tetap konsisten dan aman dari konflik karena tidak ada dua proses yang mengubah data secara bersamaan.

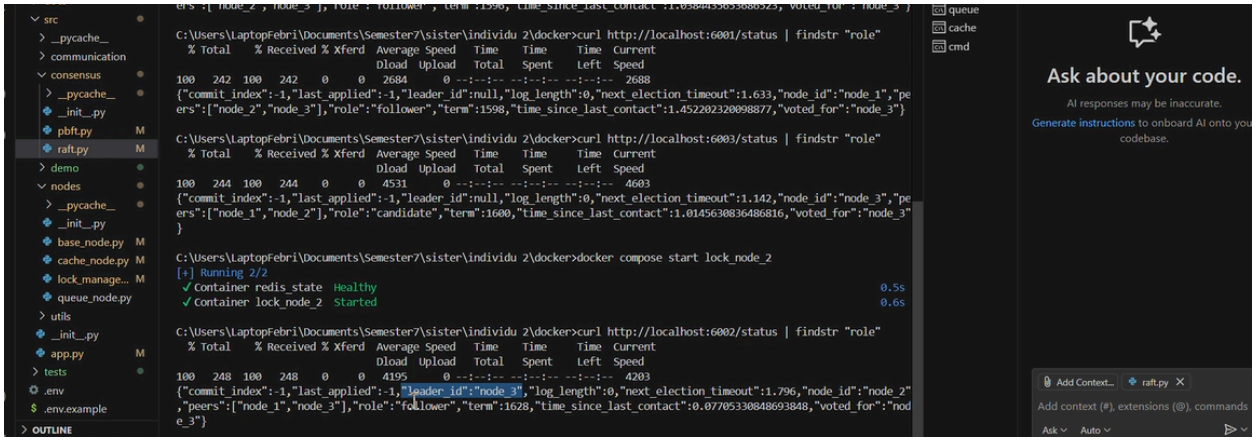
Setelah Client-C selesai menggunakan resource dan melakukan release lock, maka hak akses secara otomatis berpindah ke Client-D, yang sebelumnya berada dalam antrean tunggu. Dengan cara ini, sistem dapat menjaga agar proses berjalan secara bergantian, serta mencegah terjadinya konflik atau kerusakan data akibat akses bersamaan.

Gambar dibawah ini adalah tampilan kode shared lock.

Pada gambar di atas ditunjukkan skenario shared lock antara Client-A dan Client-D. Kedua client tersebut sama-sama mengajukan permintaan untuk mendapatkan akses ke resource yang sama, namun dengan jenis lock shared. Sistem memberikan status “success” kepada keduanya karena shared lock memungkinkan lebih dari satu client untuk mengakses resource secara bersamaan, selama akses yang dilakukan hanya bersifat membaca (read-only) dan tidak melakukan perubahan data.

Selanjutnya, Client-E juga mengajukan permintaan untuk mengakses resource yang sama, tetapi dengan jenis exclusive lock. Karena pada saat itu resource masih digunakan oleh Client-A dan Client-D dengan shared lock, maka permintaan dari Client-E tidak dapat langsung disetujui dan statusnya menjadi “waiting

Gambar dibawah ada tampilan kode fault tolerance.



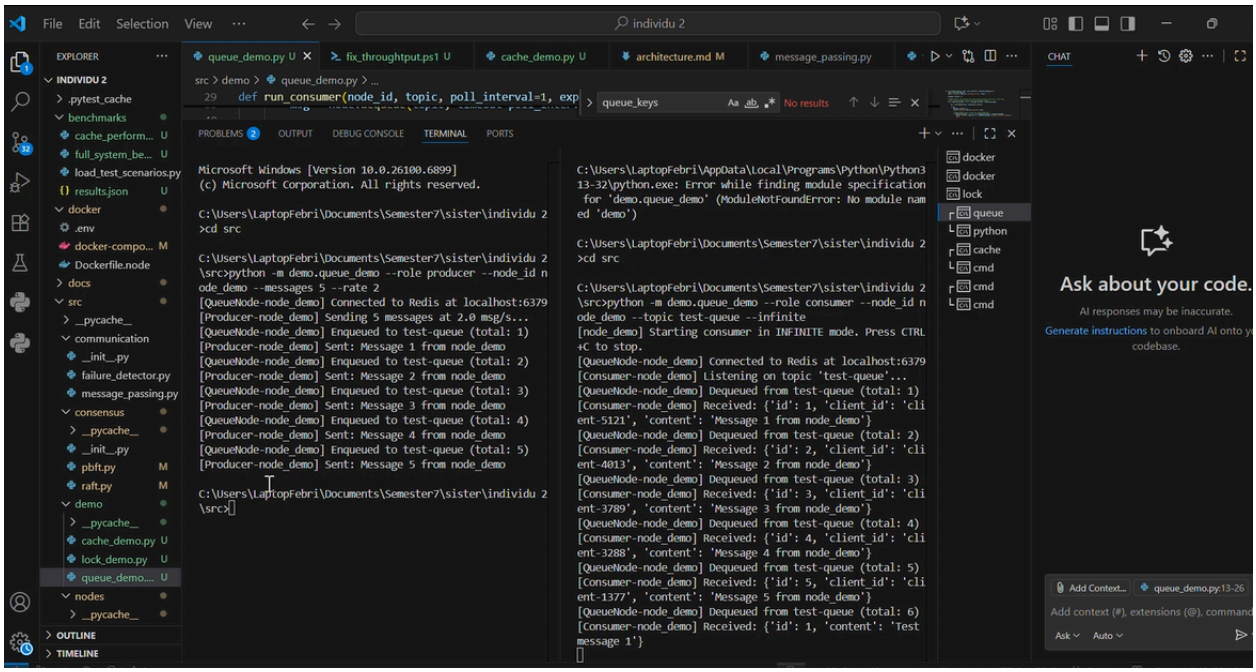
Pada gambar di atas ditunjukkan skenario node failure pada sistem terdistribusi yang menggunakan mekanisme Raft Consensus. Pada awalnya, sistem memiliki tiga node, yaitu

Node-1, Node-2, dan Node-3, dengan Node-2 berperan sebagai leader. Ketika Node-2 dihentikan (stop), sistem akan kehilangan leader. Setelah itu dilakukan pengecekan status pada node lainnya, yaitu Node-1 dan Node-3. Hasilnya menunjukkan bahwa Node-1 berstatus follower, sedangkan Node-3 berstatus candidate. Hal ini diakibatkan karena pada mekanisme Raft, agar sebuah node dapat menjadi leader, node tersebut harus memperoleh mayoritas suara dari seluruh node dalam cluster, yaitu lebih dari setengah jumlah total node. Namun, dengan hanya dua node yang aktif, tidak ada node yang bisa mendapatkan mayoritas suara, sehingga proses pemilihan tidak dapat diselesaikan.Kondisi tanpa leader ini bersifat sementara, dan ketika Node-2 dijalankan kembali, sistem kembali memiliki tiga node yang lengkap. Setelah itu, Node-3 berhasil mendapatkan mayoritas suara dan resmi terpilih sebagai leader baru, sementara Node-2 kembali berperan sebagai follower dan melakukan sinkronisasi log dengan leader yang baru.

3. Distributed Queue System

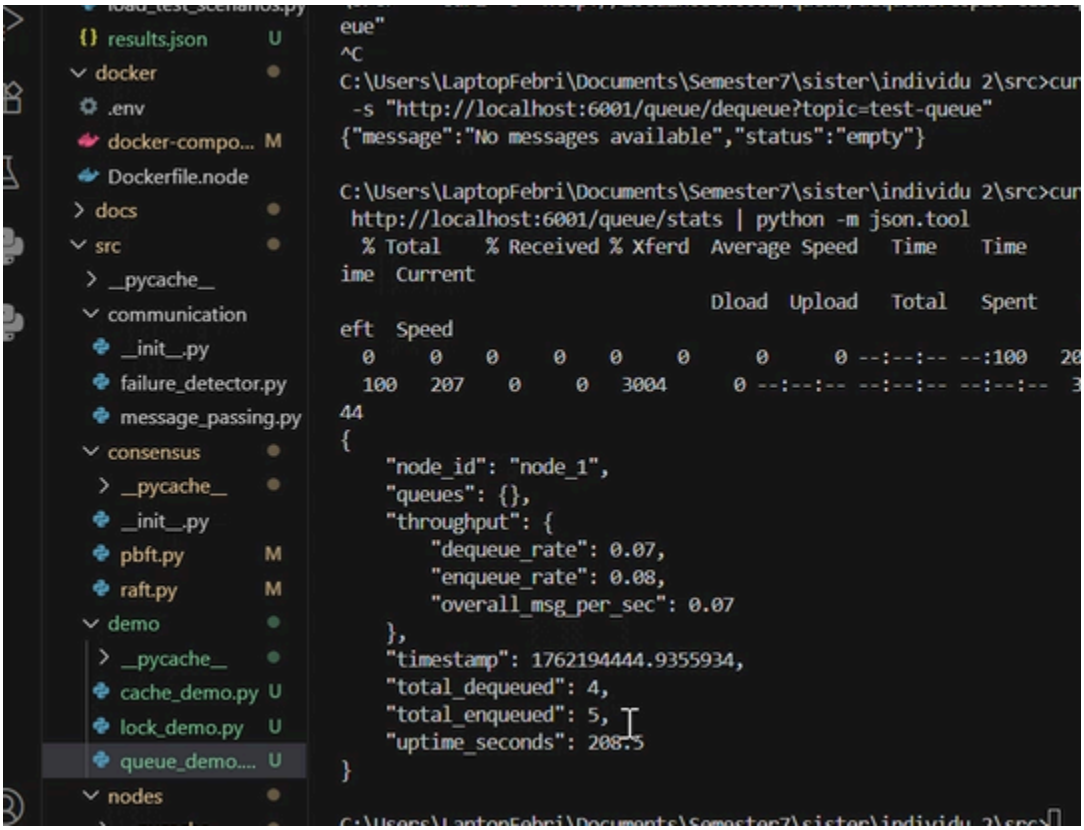
Distributed Queue System adalah mekanisme antrian pesan yang digunakan untuk mengatur proses pengiriman dan penerimaan data antara beberapa komponen dalam sistem terdistribusi. Sistem ini bekerja dengan konsep producer-consumer, di mana producer mengirim pesan ke antrian dan consumer memprosesnya secara bergantian. Dengan cara ini, komunikasi antar komponen menjadi lebih efisien, teratur, dan tetap berjalan meskipun terjadi perbedaan kecepatan proses antar node.

Gambar dibawah adalah proses pengiriman dari Producer-Consumen.



Gambar di atas menunjukkan proses interaksi antara producer dan consumer. Pada terminal sebelah kiri terlihat aktivitas producer yang mengirimkan pesan ke dalam antrian, sedangkan terminal sebelah kanan menampilkan consumer yang menerima dan memproses pesan tersebut.Kode diatas, consumer berhasil menerima sebanyak 5 pesan yang dikirim oleh producer secara berurutan.

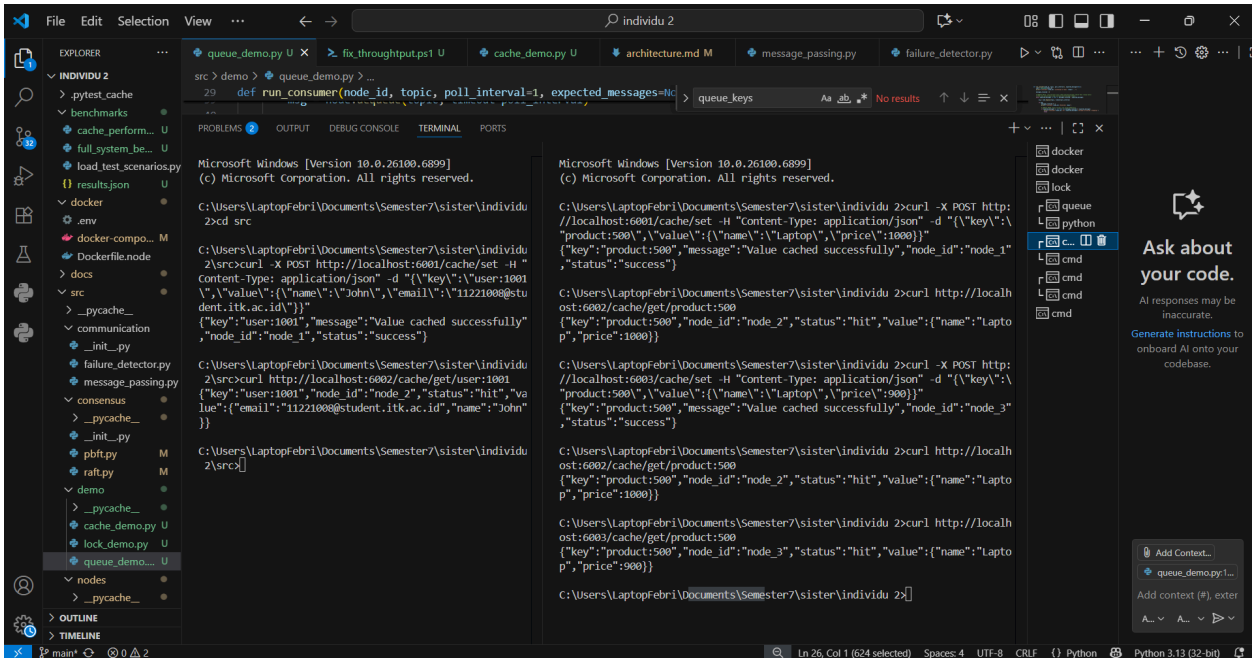
Gambar dibawah adalah tampilan dari kode Queueu/Stats



Gambar di atas menunjukkan hasil dari perintah queue stats yang menampilkan kinerja sistem antrian pada node_1. Berdasarkan data tersebut, sistem mencatat enqueue_rate sebesar 0.08 dan dequeue_rate sebesar 0.07, yang menunjukkan kecepatan pengiriman dan penerimaan pesan per detik. Total pesan yang berhasil dikirim oleh producer sebanyak 5 pesan, sementara yang telah diterima oleh consumer sebanyak 4 pesan, sehingga masih ada satu pesan yang menunggu untuk diproses. Selain itu, sistem memiliki waktu aktif (uptime) selama 2085 detik, yang menandakan bahwa node telah berjalan cukup stabil dalam memproses antrian pesan secara berkelanjutan.

4. Distributed Cache

Cache adalah tempat penyimpanan data sementara yang digunakan untuk mempercepat proses pengambilan data. Dengan menggunakan cache, sistem tidak perlu terus-menerus mengambil data dari database utama sehingga waktu akses menjadi lebih cepat dan efisien. Kegunaannya adalah meningkatkan kinerja sistem dan mengurangi beban pada server utama.

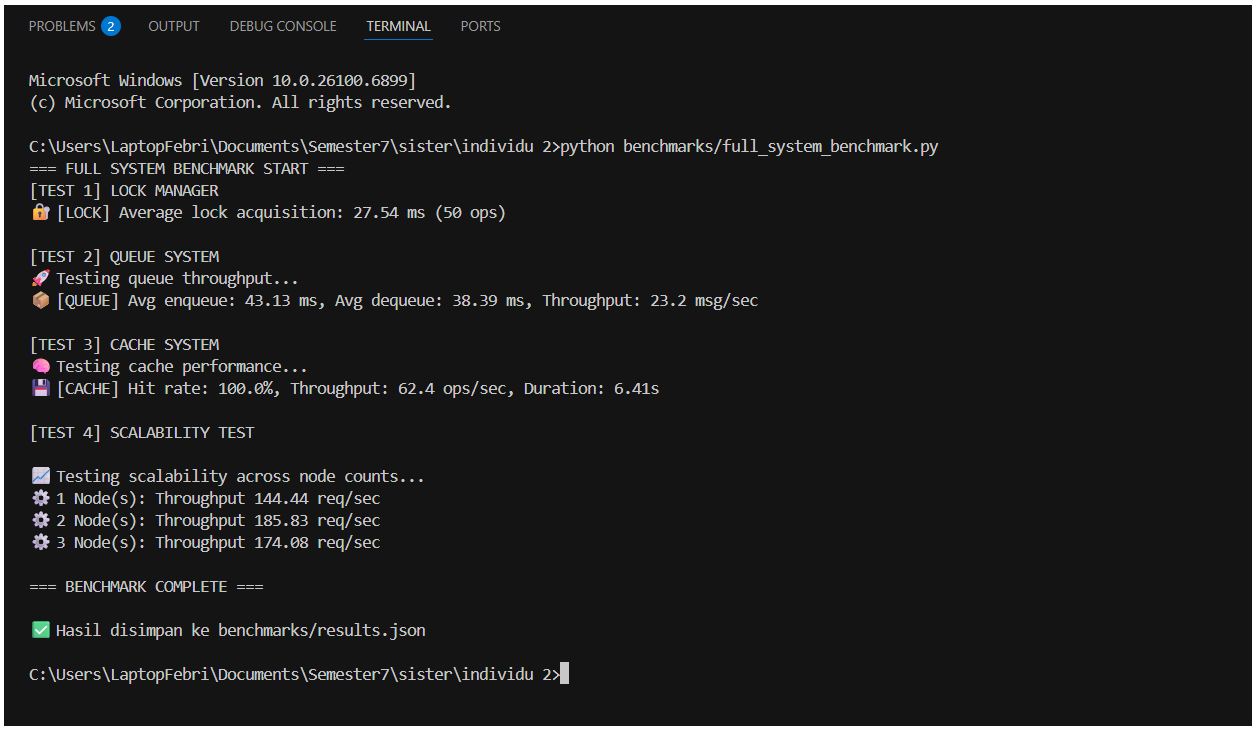


Pada gambar diatas menampilkan dimulai dengan melakukan proses write data kedalam cache yang mana menyimpan nama serta email pada node 1 pada terminal kiri. Response menunjukkan "Value cached successfully", yang berarti data berhasil disimpan di cache lokal Node-1. Setelah itu lakukan get user dengan node 2 hasilnya adalah cache hit yaitu data sudah tersedia di cache lokal node 2

Pada terminal kanan dilakukan proses cache coherence menggunakan protokol MESI. Awalnya, Node-1 menulis data produk dengan harga awal 1000 menggunakan perintah cache/set. Setelah itu, dilakukan pembacaan data pada Node-2 menggunakan perintah cache/get/product:500, dan hasilnya menunjukkan status hit, yang berarti data sudah tersimpan di cache lokal dengan harga 1000. Selanjutnya, Node-3 melakukan pembaruan (*update*) harga produk tersebut menjadi 900. Setelah update dilakukan, Node-3 mengirimkan invalidation ke node lain agar cache mereka ikut diperbarui. Ketika Node-2 kembali melakukan *get* pada product:500, harga yang ditampilkan sudah berubah menjadi 900, menandakan bahwa mekanisme cache coherence berjalan dengan baik — semua node memiliki data yang konsisten dan terbaru.

5. Benchmark & Scalability

Gambar di atas menunjukkan hasil dari pengujian performa sistem terdistribusi menggunakan perintah:



Hasil menunjukkan bahwa Lock Manager memiliki waktu rata-rata pengambilan kunci sebesar 27.54 ms, menandakan respon yang cepat. Queue System memiliki *enqueue* dan *dequeue* yang stabil dengan throughput 23.2 msg/sec, menandakan pesan terkirim tanpa kehilangan. Cache System menunjukkan kinerja sangat baik dengan *cache hit rate* 100% dan throughput 62.4 ops/sec, yang mempercepat akses data.

Sementara itu, Scalability Test memperlihatkan peningkatan throughput seiring bertambahnya node, membuktikan bahwa sistem mampu melakukan *scaling* secara horizontal dengan performa yang tetap stabil.

6. Kesimpulan

Dalam demo ini, sistem Distributed Lock Manager dengan algoritma Raft Consensus berhasil menjaga konsistensi data dan tetap berfungsi meskipun terjadi kegagalan pada salah satu node. Distributed Queue System menunjukkan kinerja tinggi tanpa kehilangan pesan, sementara Distributed Cache meningkatkan performa sistem secara signifikan melalui mekanisme cache coherence. Tantangan utama mencakup penerapan algoritma konsensus, pengaturan invalidasi cache antar-node, serta konfigurasi container dan jaringan. Semua tantangan tersebut berhasil diatasi dengan penerapan Redis pub/sub, logging, dan mekanisme heartbeat. Ke depannya, sistem dapat ditingkatkan melalui penambahan monitoring, optimasi cache, serta otomatisasi pengujian. Sistem ini berpotensi diterapkan pada berbagai lingkungan terdistribusi seperti microservices, basis data terdistribusi, sistem antrian pesan, dan layanan cache.