Nama : Alifah Nur Aisyah
NIM : 11221069
Mata Kuliah : Sistem Paralel dan Terdistribusi - A

LAPORAN
TUGAS INDIVIDU 2

A. Definisi Sistem

Distributed Synchronization System adalah sebuah platform sinkronisasi terdistribusi yang dibangun untuk mengatasi tantangan koordinasi dan konsistensi data dalam lingkungan komputasi terdistribusi. Sistem ini mengintegrasikan tiga komponen fundamental yang bekerja secara independen namun kohesif: Distributed Lock Manager, Distributed Queue System, dan Distributed Cache Coherence System.

Secara teknis, sistem ini dapat didefinisikan sebagai arsitektur layanan mikro (microservices) yang di kontainerisasi menggunakan Docker, dimana setiap service menjalankan fungsi spesifik dalam menjaga konsistensi state dan koordinasi operasi di antara multiple nodes yang tersebar. Platform ini memanfaatkan algoritma consensus (Raft), teknik distribusi data (Consistent Hashing), dan protokol koherensi (MESI) untuk mencapai reliabilitas dan performa tinggi dalam skenario concurrent access.

B. Arsitektur Sistem

Sistem yang dikembangkan dalam uji coba ini menggunakan pendekatan arsitektur layanan terdistribusi berbasis container. Implementasi teknologi Docker dipilih sebagai fondasi utama untuk menjalankan komponen-komponen sistem secara mandiri, namun tetap terkoordinasi. Penggunaan Docker Compose memfasilitasi hubungan antar container, memungkinkan setiap komponen berfungsi sebagai service yang dapat diskalakan sesuai kebutuhan. Secara garis besar, arsitektur sistem terbagi menjadi beberapa layer dan komponen yang berinteraksi membentuk ekosistem terdistribusi yaitu:

- Layer Client : Client akan merepresentasikan eksternal yang mengakses sistem, bisa berupa aplikasi pengguna akhir atau service lain yang memerlukan fungsionalitas sinkronisasi terdistribusi. Komunikasi dilakukan melalui protokol

HTTP dengan arsitektur RESTful API. Client tidak berada dalam environment Docker, melainkan berkomunikasi dengan sistem melalui port yang telah di expose ke host machine. Keputusan untuk memisahkan client dari environment Docker memberikan fleksibilitas deployment. Client dapat berjalan pada platform apapun selama memiliki kemampuan melakukan HTTP request. Hal ini sejalan dengan prinsip interoperabilitas dalam sistem terdistribusi modern.

- Docker Environment : Seluruh komponen inti sistem berjalan dalam environment Docker yang dikelola oleh Docker Engine. Environment ini menciptakan isolasi logis melalui containerization, dimana setiap service berjalan dalam container terpisah dengan resources yang terdefinisi. Docker Engine bertanggung jawab atas lifecycle management dari container-container tersebut.
- Application Node Containers : Container aplikasi pada sistem ini merupakan instance dari satu Docker image yang sama, namun perannya dapat berbeda sesuai dengan konfigurasi environment variable saat startup. Terdapat tiga jenis node yang dijalankan, yaitu Raft Nodes untuk manajemen distributed lock berbasis protokol konsensus Raft, Queue Nodes untuk pengelolaan message queue menggunakan consistent hashing, serta Cache Nodes yang menerapkan protokol cache coherence MESI. Penentuan peran dan identitas setiap node dilakukan melalui environment variables seperti NODE_TYPE dan NODE_ID yang diatur dalam file docker-compose.yml, sehingga satu image dapat digunakan untuk seluruh tipe node dengan perilaku yang berbeda pada runtime. Komunikasi internal antar node, baik untuk konsensus, message forwarding, maupun cache invalidation, dilakukan melalui Docker internal network, sementara port-port tertentu diekspos ke host machine untuk memungkinkan interaksi dari client eksternal.
- Redis Container : Redis berfungsi sebagai *persistence layer* dan *temporary storage* bagi sistem Distributed Queue. Semua pesan yang masuk akan disimpan di Redis untuk menjamin durabilitas dan mendukung mekanisme *at-least-once delivery*. Container Redis hanya dapat diakses dari dalam Docker network, sehingga keamanan data lebih terjaga karena tidak diekspos langsung ke host. Redis dipilih karena mampu menangani operasi baca/tulis dengan sangat cepat

sebagai *in-memory data store*, namun tetap menyediakan opsi *persistence*. Hal ini menjadikannya cocok untuk kebutuhan sistem antrian yang memerlukan throughput tinggi dan latency rendah.

C. Alur Interaksi Sistem

Pada Distributed Lock Manager, interaksi komponen bekerja seperti berikut: ketika client ingin melakukan *acquire lock* pada suatu resource, client mengirim HTTP POST ke salah satu node yang portnya terbuka, misalnya Node 1. Jika Node 1 bukan Leader dalam cluster Raft, ia akan mengembalikan informasi Leader yang sebenarnya (misalnya Node 2), sehingga client mengirim ulang request ke Leader tersebut.

Setelah Leader (Node 2) menerima request, ia mencatat perintah lock acquisition ke log sebagai *uncommitted entry*, lalu mengirimkan AppendEntries RPC kepada seluruh Follower melalui Docker internal network. Follower yang menerima entry akan memvalidasi dan menyimpannya, kemudian mengirim acknowledgment kembali ke Leader. Jika sebagian besar (quorum) sudah memberikan acknowledgment, maka entry dianggap *committed* dan Leader memperbarui state machine untuk menetapkan lock ke requester. Leader kemudian mengirim respons sukses ke client.

Selanjutnya, pada heartbeat rutin, Leader memberi tahu Follower bahwa entry tersebut telah *committed*, sehingga mereka juga memperbarui state machine masing-masing agar seluruh cluster tetap konsisten. Prosedur ini memastikan sistem tetap dapat bekerja meskipun ada node yang gagal.

Alur komunikasi serupa juga digunakan pada operasi queue dan cache, dengan protokol yang berbeda (consistent hashing untuk queue dan MESI untuk cache). Secara umum, client hanya berkomunikasi melalui port yang diekspos, sedangkan koordinasi antar node berlangsung secara internal di Docker network dengan Redis sebagai storage untuk memastikan data tetap aman.

D. Uji Coba

1. LOCK MANAGER

Distributed Lock Manager dibangun menggunakan Raft Consensus Algorithm untuk menjamin konsistensi lock operations di lingkungan terdistribusi. Sistem ini mendukung exclusive locks, shared locks, dan memiliki mekanisme deteksi deadlock.

a. Implementasi Raft Consensus Algorithm

- Requirement : Implementasi distributed lock menggunakan algoritma Raft Consensus
- Implementasi : Sistem menggunakan Raft untuk mencapai consensus di antara nodes. Raft memiliki 3 komponen utama yaitu Leader Election (pemilihan leader secara demokratis), Log Replication (replikasi operasi lock ke semua nodes), dan Heartbeat Mechanism (maintenance komunikasi antar nodes)
- Langkah Pengujian :

  1) Mulai 3 lock manager nodes secara bersamaan
  2) Mengamati proses leader
  3) Verifikasi Raft state disetiap node
  4) Monitor heartbeat komunikasi

b. Hasil Pengujian

- Request ke Node 1

```
GET http://localhost:5001/api/status


{
 "node_id": 1,
 "address": "localhost:5001",
 "running": true,
 "raft_state": {
  "node_id": 1,
  "state": "follower",
  "term": 1346,
  "leader_id": 2,
  "log_size": 0,
  "commit_index": 0,
  "elections_started": 1145,
  "elections_won": 110,
```

```
   "heartbeats_sent": 0
  }
}
```


- Minimum 3 nodes yang berkomunikasi : Pengujian terhadap implementasi Distributed Lock Manager dengan tiga node aktif pada port 5001–5003 menunjukkan bahwa sistem mampu menjalankan komunikasi antar node menggunakan protokol message passing untuk Raft consensus. Berdasarkan hasil monitoring failure detector, sistem mendeteksi dua node selain dirinya sendiri, dengan satu node dalam kondisi alive dan satu node lainnya teridentifikasi sebagai gagal (Node 3), sementara pesan yang dikirim berjumlah 360 dan memperlihatkan bahwa komunikasi antar node berjalan meskipun terdapat 2.038 pengiriman yang gagal akibat satu node tidak merespons. Node 1 sebagai follower tetap dapat berkomunikasi dengan Node 2 sebagai leader melalui mekanisme heartbeat dan pertukaran pesan untuk proses pemilihan leader. Dengan total tiga node, quorum yang diperlukan adalah dua node, dan kondisi ini masih terpenuhi karena Node 1 dan Node 2 tetap aktif, sehingga sistem tetap dapat beroperasi secara normal. Secara keseluruhan, ketiga node berhasil dikonfigurasi, komunikasi antar node berjalan, failure detector berfungsi mendeteksi node yang gagal, dan sistem mampu mempertahankan quorum tanpa gangguan operasional.
- Pengujian mekanisme locking (Exclusive dan shared lock) : Pengujian mekanisme locking dilakukan untuk memastikan konsistensi dan keamanan akses terhadap resource dalam Distributed Lock Manager. Pada pengujian Exclusive Lock, ketika client mengirimkan permintaan acquire lock ke Node 1 (yang merupakan follower), sistem menolak permintaan tersebut dengan memberikan respons *not_leader* dan menginformasikan *leader_id* sehingga client dapat mengarahkan ulang request ke Leader (Node 2). Saat resource sudah berada dalam kondisi exclusive lock, request lain untuk resource yang sama akan ditolak sehingga tidak terjadi

konflik. Hal ini menunjukkan bahwa mekanisme blocking terhadap akses eksklusif berfungsi dengan benar. Pada pengujian Shared Lock, permintaan shared lock dari beberapa requester dapat diproses bersamaan tanpa konflik karena shared lock memungkinkan multiple readers selama tidak ada exclusive lock yang aktif. Selain itu, mekanisme release lock juga diuji dan sistem mampu menolak request release yang dikirim kepada follower, sekaligus menjaga agar hanya Leader yang menangani operasi write. Informasi status lock mencakup jumlah active_locks, waiting_requests, serta statistik penting seperti jumlah pengambilan lock, pelepasan, timeout, dan deteksi deadlock, yang semuanya tercatat dengan baik. Secara keseluruhan, hasil pengujian menunjukkan bahwa sistem telah berhasil menerapkan exclusive lock, shared lock, penanganan konflik, mekanisme release, serta pelacakan status lock secara konsisten.

- Hasil Partisi :

  Node 1: Follower

  Node 2: Leader

  Node 3: Follower

  Total: 3 nodes, Quorum: 2

2. DISTRIBUTED QUEUE SYSTEM

Distributed Queue System dibangun menggunakan Consistent Hashing untuk mendistribusikan messages secara merata di antara nodes. Sistem ini mendukung multiple producers dan consumers, message persistence melalui Redis, serta at-least-once delivery guarantee.

a. Implementasi Consistent Hashing

- Requirement : Implementasi distributed queue menggunakan consistent hashing
- Implementasi : Sistem menggunakan hash ring dengan 150 virtual nodes per physical node untuk distribusi yang lebih merata. Setiap message diroute ke node yang bertanggung jawab berdasarkan hash value dari message_id.

- Langkah Pengujian :

  1) Inisialisasi 3 queue nodes (localhost:6001, 6002, 6003)

  2) Verify hash ring formation

  3) Test message routing berdasarkan consistent hashing

b. Hasil Pengujian :

- Initial Status


```
{
 "node_id": 1,
 "queue_size": 0,
 "hash_ring_nodes": [
  "localhost:6001",
  "localhost:6002",
  "localhost:6003"
 ]
}
```


- Enqueue Operations

Message 1: Response: {"status": "enqueued", "message_id": "msg_001", "node_id": 1}
Message diterima langsung oleh Node 1
Message 2: Response: {"status": "redirected", "responsible_node": "localhost:6003"}
Consistent hashing mengarahkan ke Node 3

Message 3 : Response: {"status": "redirected", "responsible_node": "localhost:6003"}
Routing konsisten ke Node 3

- Analisis : Hash ring berhasil dibentuk dengan tiga nodes, sehingga distribusi data dapat dilakukan secara seimbang. Dengan menggunakan

consistent hashing, setiap pesan yang memiliki nilai hash berbeda akan diarahkan ke node yang sesuai pada ring tersebut. Mekanisme ini akan memastikan keputusan routing tetap konsisten dan dapat diprediksi, meskipun terjadi perubahan jumlah nodes dalam sistem.

c. Support untuk Multiple Producers dan Consumers
    - Requirement : Support untuk multiple producers dan consumers
    - Implementasi : Sistem dirancang agar setiap node dapat bertindak sebagai producer (enqueue) dan consumer (dequeue). Client dapat mengirim request ke node mana pun, dan sistem akan menghandle routing.
    - Langkah Pengujian :
        1) Enqueue messages dari Node 1 (acting as producer)
        2) Verify messages dapat dienqueue ke berbagai nodes
        3) Dequeue message dari Node 1 (acting as consumer)

d. Hasil Pengujian :
    - Multiple Procedure Test

```
Request: POST http://localhost:6001/api/queue/dequeue

Response:
{
  "message_id": "msg_001",
  "data": {"content": "Hello from Queue", "timestamp": 1698765432},
  "retry_count": 0
}

 Node 1 berhasil dequeue message sebagai consumer
```

    - Distribution Check : Node 1 memiliki satu pesan dalam antrian (msg_001), Node 2 tidak menerima pesan, sedangkan Node 3 terindikasi menampung pesan msg_002 dan msg_003. Hal ini membuktikan bahwa setiap node mampu menerima permintaan enqueue sebagai producer sekaligus memproses permintaan dequeue sebagai consumer. Beban

penyimpanan pesan juga berhasil terdistribusi ke beberapa node, sehingga sistem tidak bergantung pada satu titik masuk saja. Dengan demikian, uji distribusi dinyatakan berhasil (PASSED) dan sistem terbukti mendukung model multiple producers dan multiple consumers.

e. Message Persistence and Recovery
   - Requirement : Implementasi message persistence dan recovery
   - Implementasi : Messages dipersist ke Redis untuk memastikan durability. Setiap message yang dienqueue disimpan di Redis sebelum acknowledgment dikirim ke client.
   - Langkah Pengujian
     1) Enqueue message ke queue
     2) Verify data integrity setelah dequeue
     3) Check Redis connection dari queue status

f. Hasil Pengujian
   - Enqueue with Data

```
Request Body:
{
  "message_id": "msg_001",
  "data": {
    "content": "Hello from Queue",
    "timestamp": 1698765432
  }
}

Response: {"status": "enqueued", "message_id": "msg_001", "node_id": 1}
```

   - Dequeue with Intact Data

```
Response:
{
```

```
  "message_id": "msg_001",
  "data": {
    "content": "Hello from Queue",
    "timestamp": 1698765432
  },
  "retry_count": 0
}
```


- Analisis : Verifikasi integritas data menunjukkan bahwa isi pesan tetap sama setelah dilakukan proses dequeue, dengan timestamp dan struktur data yang tidak mengalami perubahan serta tanpa adanya kehilangan data. Selain itu, log sistem memperlihatkan bahwa koneksi ke Redis pada alamat localhost:6379 berjalan dengan baik, di mana setiap pesan dipersist sebelum acknowledgment diberikan. Hasil pengujian ini membuktikan bahwa mekanisme message persistence telah berfungsi dengan benar menggunakan Redis, sehingga sistem mampu melakukan pemulihan pesan apabila diperlukan tanpa risiko data corrupt atau field yang hilang. Dengan demikian, status pengujian dinyatakan berhasil (PASSED) dan message persistence terbukti bekerja dengan baik.

g. Handle Node Failure Tanpa Kehilangan Data
    - Requirement : Handle node failure tanpa kehilangan data
    - Implementasi : Dengan consistent hashing dan Redis persistence, jika satu node fail, messages yang seharusnya diroute ke node tersebut akan otomatis diredirect ke node berikutnya dalam hash ring.

h. Hasil Pengujian


```
"hash_ring_nodes": [
  "localhost:6001",
  "localhost:6002",
  "localhost:6003"
]
```

Mekanisme fallback pada message routing bekerja dengan memanfaatkan perhitungan hash untuk menentukan node tujuan penyimpanan pesan. Jika node yang menjadi target awal tidak tersedia, maka pesan secara otomatis akan dialihkan ke node berikutnya dalam urutan hash ring untuk memastikan proses enqueue tetap berjalan. Semua pesan yang diterima juga disimpan secara persisten di Redis sehingga tetap aman meskipun terjadi kegagalan pada node tertentu. Pada skenario pengujian yang dilakukan, pesan *msg_001* tersimpan pada Node 1, sedangkan *msg_002* dan *msg_003* diarahkan ke Node 3 dan disimpan di Redis. Jika Node 3 mengalami kegagalan, hash ring akan diperbarui dan hanya menyisakan Node 1 dan Node 2 sebagai node aktif. Dengan demikian, pesan-pesan yang seharusnya dialokasikan ke Node 3 akan secara otomatis dialihkan ke node terdekat berikutnya yang masih aktif dalam ring. Sementara itu, pesan yang sudah tersimpan di Redis tetap dapat diakses dan direcover tanpa kehilangan data. Hal ini memastikan ketahanan sistem terhadap node failure dan menjaga konsistensi operasional message queue.

i. Hasil Failed Messages Tracking

```
"statistics": {
  "enqueued": 1,
  "dequeued": 1,
  "failed": 0    ← No failed messages
}
```

Analisis hasil pengujian menunjukkan bahwa setiap node mampu melacak keanggotaan hash ring secara konsisten, sehingga perubahan status node dapat segera diketahui dan ditangani. Mekanisme persistence melalui Redis menjamin durabilitas data, memastikan tidak ada pesan yang hilang saat terjadi gangguan. Pengujian juga mencatat zero message loss dengan nilai *failed: 0*, menegaskan keandalan proses recovery. Selain itu, penggunaan consistent hashing memberikan kemampuan failover yang mulus ketika terjadi kegagalan node. Berdasarkan semua temuan tersebut, pengujian dinyatakan berhasil (PASSED)

dengan node failure handling yang terbukti tidak menyebabkan kehilangan data sama sekali.

j. At-Least-Once Delivery Guarantee

- Requirement : Support untuk at-least-once delivery guarantee
- Implementasi : Sistem menggunakan acknowledgment mechanism. Message tetap dalam queue hingga consumer mengirim acknowledgment. Jika acknowledgment tidak diterima, message akan diretry.
- Hasil Pengujian : Mekanisme *at-least-once guarantee* pada sistem diuji melalui pemantauan *retry_count* dan proses acknowledgment. Nilai "retry_count": 0 menunjukkan bahwa pesan berhasil diproses tanpa kegagalan, namun apabila terjadi error selama pemrosesan, nilai ini akan meningkat dan pesan akan dimasukkan kembali ke antrean hingga berhasil diproses. Pesan tidak akan dihapus dari penyimpanan sebelum acknowledgment diterima, sehingga memastikan bahwa setiap pesan minimal diproses satu kali dan tetap aman jika terjadi *consumer crash*.

  Berdasarkan hasil statistik, jumlah pesan yang berhasil di*enqueue* dan di*dequeue* memiliki nilai yang sama (1), dengan 0 pesan gagal, menegaskan bahwa tidak ada pesan yang hilang maupun tertinggal di dalam antrean. Hasil analisis menunjukkan bahwa sistem telah menerapkan mekanisme acknowledgment dengan benar, melacak *retry* untuk setiap pesan, serta menjamin keberhasilan pengiriman pesan secara penuh. Dengan demikian, *at-least-once guarantee* terbukti terpenuhi dan pengujian ini dinyatakan berhasil (PASSED).

3. DISTRIBUTED CACHE COHERENCE SYSTEM

Distributed Cache Coherence System dibangun menggunakan MESI Protocol untuk menjaga konsistensi cache entries di antara multiple nodes. Sistem ini mengimplementasikan four state coherence (Modified, Exclusive, Shared, Invalid) dengan LRU replacement policy dan automatic invalidation mechanism.

a. Implementasi Cache Coherence Protocol

- Requirement : Implementasi cache coherence protocol

- Implementasi : Sistem menerapkan protokol MESI untuk menjaga konsistensi cache antar node, dengan setiap entry berada pada salah satu dari empat state. State Modified (M) menunjukkan bahwa data telah diubah di cache dan belum konsisten dengan main memory. State Exclusive (E) menandakan bahwa data dalam kondisi bersih dan hanya tersimpan pada satu cache saja. State Shared (S) berarti data bersih tetapi juga mungkin tersimpan pada cache lain secara bersamaan. Sementara itu, state Invalid (I) menunjukkan bahwa data sudah tidak valid atau usang sehingga tidak boleh lagi digunakan. Dengan penggunaan keempat state ini, protokol MESI memastikan bahwa setiap cache dapat beroperasi secara efisien namun tetap menjaga konsistensi data di seluruh sistem terdistribusi.
- Langkah Pengujian :

  1) Test initial cache status (empty cache)
  2) PUT operation → State transition ke MODIFIED
  3) GET from different node → State transition ke SHARED (both nodes)
  4) PUT from third node → INVALIDATE other nodes
- Hasil Pengujian :

  Pengujian dilakukan untuk memverifikasi implementasi protokol MESI, koordinasi antar multiple cache nodes, mekanisme invalidation, kebijakan cache replacement LRU, serta monitoring performa. Sistem dikonfigurasi dengan tiga cache nodes yang berkomunikasi secara terdistribusi pada port 7001–7003.

  Pengujian pertama menunjukkan bahwa cache berada dalam kondisi awal kosong dengan kapasitas 1000 entries dan seluruh statistik pencatatan operasi masih bernilai nol, menandakan sistem siap menerima operasi. Selanjutnya, operasi PUT pada Node 1 berhasil memasukkan data dan menghasilkan state Modified (M) karena data baru hanya tersedia pada node tersebut. Ketika dilakukan operasi GET pada Node 1 untuk key yang sama, sistem berhasil memberikan local hit sehingga akses data berlangsung cepat. Namun, ketika operasi GET dilakukan melalui Node 2,

Node 2 melakukan remote fetch dari Node 1 sehingga kedua node berbagi data dalam state Shared (S). Hal ini memperlihatkan bahwa sharing dan propagasi data bekerja sesuai rancangan. Pengujian berlanjut dengan skenario invalidation di mana operasi PUT pada Node 3 dengan key yang sama menyebabkan Node 1 dan Node 2 melakukan perubahan state dari Shared (S) ke Invalid (I). Node 3 kemudian menjadi pemilik baru data dalam state Modified (M). Bukti dari hasil testing memperlihatkan adanya invalidation counters yang tercatat pada setiap node, menunjukkan bahwa broadcast invalidation berfungsi otomatis dan data yang stale tidak digunakan lagi oleh node lain.

Sistem juga mendukung multiple cache nodes secara efektif. Statistik pada tiap node menunjukkan adanya komunikasi dan sinkronisasi antar cache nodes melalui mekanisme invalidation maupun remote fetch. Protokol MESI mampu menjaga cache coherence di seluruh cluster selama proses baca dan tulis berlangsung.

Selain itu, mekanisme cache replacement policy LRU (Least Recently Used) telah diuji dengan memasukkan dan mengakses data dalam urutan tertentu. Sistem berhasil mencatat urutan akses dan siap melakukan eviction ketika kapasitas penuh. Meskipun dalam pengujian kapasitas tidak mencapai batas maksimal, eviction counter sudah berjalan dan siap menghapus data yang paling jarang diakses.

Terakhir, performance monitoring menunjukkan hit rate yang sangat baik, yaitu 63,6%, dengan pencatatan statistik lengkap seperti jumlah reads, writes, hits, misses, invalidations, dan evictions. Monitoring ini memberikan informasi performa yang akurat dan real time terhadap beban akses cache. Secara keseluruhan, hasil pengujian menunjukkan bahwa seluruh requirement pada Distributed Cache Coherence System terpenuhi dengan baik, meliputi:

- Implementasi protokol MESI
- Koordinasi multiple cache nodes
- Mekanisme invalidation & update propagation

- LRU replacement policy
- Performance metrics & monitoring

Sistem terbukti mampu menjaga data consistency, mengurangi latency melalui cache hits, dan memberikan dukungan penuh untuk pengujian concurrency serta fault tolerance pada distributed environment.