LAPORAN TUGAS 2

Sistem Paralel dan Terdistribusi

Synchronization and Distributed Systems

Angger Karisma Deotama 11221010

BAB I
DESAIN ARSITEKTUR & RINGKASAN ALUR

1.1 Pendahuluan

Sistem terdistribusi modern menuntut skalabilitas, ketersediaan tinggi, dan toleransi terhadap kegagalan. Namun, distribusi komputasi dan data di berbagai node jaringan memperkenalkan tantangan fundamental dalam sinkronisasi, komunikasi, dan konsistensi data. Tanpa mekanisme yang andal, sistem berisiko mengalami race conditions, kehilangan pesan, dan latensi tinggi akibat akses data yang tidak efisien. Proyek ini bertujuan untuk mengembangkan sebuah sistem sinkronisasi terdistribusi yang mensimulasikan skenario dunia nyata, di mana beberapa node harus berkomunikasi dan menyinkronkan data secara konsisten.

1.2 Tujuan

Tujuan dari proyek ini adalah merancang, mengimplementasikan, dan menganalisis sebuah sistem terdistribusi yang mengintegrasikan tiga komponen inti untuk mengatasi tantangan utama dalam sinkronisasi dan konsistensi data:

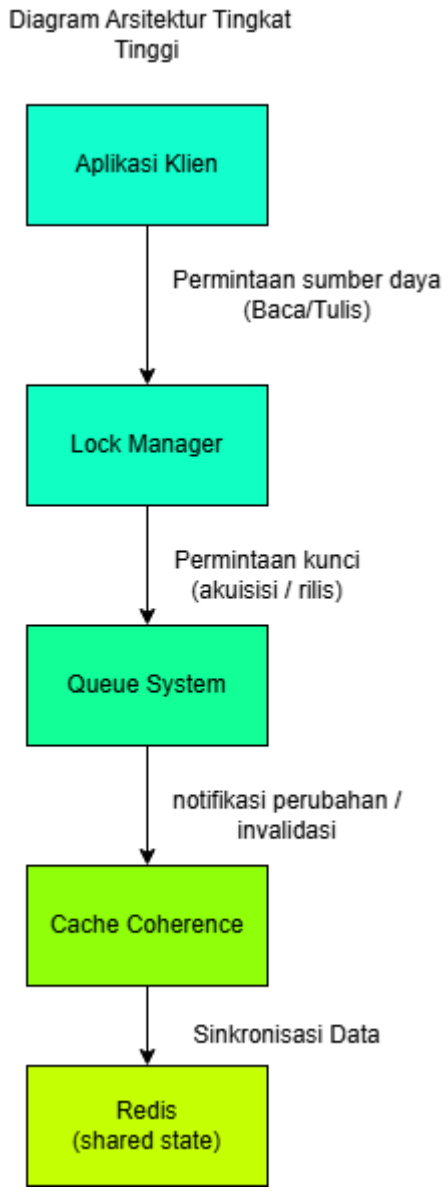
1. Distributed Lock Manager yang menyediakan mekanisme mutual exclusion untuk mengelola akses serentak ke sumber daya bersama, diimplementasikan dengan algoritma konsensus Raft.
2. Distributed Queue System yang memfasilitasi komunikasi asinkron yang andal dan dapat diskalakan antara produsen dan konsumen, diimplementasikan dengan Consistent Hashing.
3. Distributed Cache Coherence yang mengurangi latensi akses data dengan menyediakan lapisan cache lokal yang konsisten, diimplementasikan dengan protokol yang meniru logika MESI.

Laporan ini akan merinci arsitektur, algoritma yang digunakan, fungsionalitas, serta analisis kinerja dari sistem yang dibangun.

BAB II
DESAIN ARSITEKTUR

2.1 Desain Arsitektur

Arsitektur sistem dirancang sebagai platform kohesif yang terdiri dari beberapa komponen utama yang saling berinteraksi untuk menyediakan fondasi yang kuat bagi aplikasi terdistribusi.



Gambar 2.1 Desain Arsitektur.

2.1.1 Aplikasi Klien

Aplikasi Klien adalah entitas yang menggunakan layanan yang disediakan oleh infrastruktur terdistribusi. Dalam implementasi ini, klien berinteraksi dengan sistem melalui API yang diekspos oleh main.py menggunakan FastAPI. Klien dapat meminta untuk memperoleh/melepaskan kunci, mengirim/menerima pesan dari antrean, serta membaca/menulis data melalui lapisan cache.

2.1.2 *Lock_Manager_Raft*

Komponen ini berfungsi sebagai penjaga untuk sumber daya bersama. Implementasinya terdapat dalam lock_manager.py dan raft.py. LockManager bertindak sebagai state machine yang direplikasi, sementara RaftNode memastikan bahwa setiap perubahan pada keadaan kunci (acquire/release) disetujui oleh mayoritas node melalui konsensus. Ini menjamin konsistensi dan toleransi kegagalan. Semua permintaan tulis harus diarahkan ke node Leader Raft.

2.1.3 *Queue_System_Hashing*

Komponen ini bertindak sebagai tulang punggung komunikasi asinkron. Implementasinya terdapat dalam queue_manager.py dan utils/consistent_hashing.py. Sistem ini menggunakan Consistent Hashing untuk memetakan setiap nama antrean (queue_name) ke node yang bertanggung jawab. Jika sebuah node menerima permintaan untuk antrean yang bukan tanggung jawabnya, permintaan tersebut akan diteruskan (forward) ke node yang benar. Ini memungkinkan penyeimbangan beban dan skalabilitas horizontal.

2.1.4 *Cache_Coherence_MESI*

Komponen ini dirancang untuk mengurangi latensi akses data. Implementasinya terdapat dalam cache_manager.py. Setiap node memiliki cache lokal. Untuk menjaga konsistensi antar cache, sistem mengimplementasikan protokol koherensi yang meniru logika MESI. Ketika sebuah node menulis data, ia akan mengirimkan pesan invalidasi ke node lain yang memiliki salinan data tersebut, memastikan bahwa node lain akan mengambil data versi terbaru pada pembacaan berikutnya

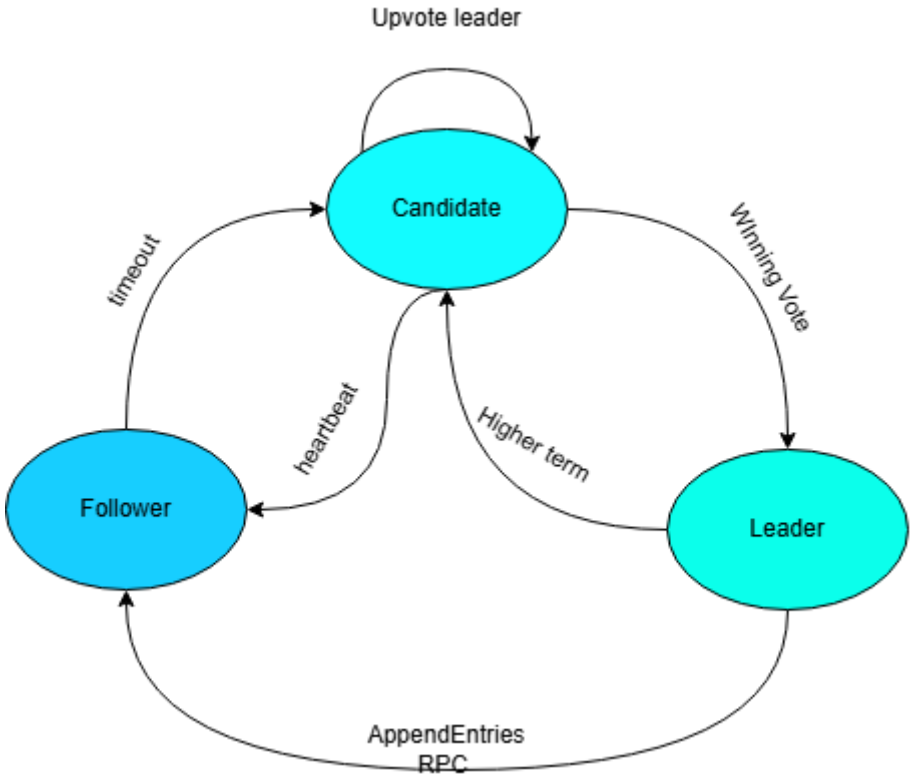
2.1.5 *Redis Shared State*

.Redis berfungsi sebagai penyimpanan data utama dan medium komunikasi untuk beberapa komponen. Dalam queue_manager.py, Redis digunakan sebagai message broker persisten untuk antrean. Dalam cache_manager.py, Redis bertindak sebagai "Main Memory" atau backing store tempat data dibaca saat terjadi cache miss dan ditulis kembali saat terjadi write-back dari cache.

2.2 Algoritma yang Digunakan

2.2.1 *Lock Manager* & Konsensus *(Distributed Lock)*

Raft Consensus Raft adalah algoritma konsensus yang dirancang untuk lebih mudah dipahami daripada Paxos. Tujuannya adalah untuk mengelola log yang direplikasi. Dalam proyek ini, raft.py mengimplementasikan logika inti Raft. Leader Election adalah situasi dimana Node memulai sebagai Follower. Jika tidak menerima heartbeat dari Leader dalam election_timeout yang acak, ia menjadi Candidate, menaikkan term, dan meminta suara. Jika mendapat suara mayoritas, ia menjadi Leader. Terjadinya Log Replication dimana Leader menerima perintah dari klien, menambahkannya ke log lokal, dan mereplikasikannya ke Follower melalui RPC append_entries. Setelah mayoritas Follower mengonfirmasi, entri dianggap committed, dan perintah diterapkan ke state machine (dalam kasus ini, LockManager). Dan Safety yang dimana Raft memastikan bahwa hanya Leader yang dapat menambahkan entri dan Leader selalu memiliki semua entri yang telah di-commit. Saat partisi jaringan terjadi, hanya partisi dengan mayoritas node yang dapat memilih Leader baru dan membuat kemajuan.



Gambar 2.2 Algoritma Raft.

Wait-For-Graph (WFG) & Deadlock Detection Secara teoretis, deadlock dalam sistem kunci terdistribusi dapat dideteksi dengan membangun Wait-For-Graph (WFG), di mana siklus dalam graf mengindikasikan deadlock. Implementasi saat ini di lock_manager.py tidak secara eksplisit membangun WFG. Sebaliknya, ia menggunakan pendekatan yang lebih sederhana untuk menangani potensi kebuntuan: timeout. Dalam fungsi acquire_lock, klien akan berhenti menunggu setelah periode timeout tertentu, mencegah penungguan tanpa batas dan memungkinkan klien untuk mencoba kembali atau melaporkan kegagalan.

2.2.2 Algoritma *Consistent Hashing (Queue System)*

a. *Consistent Hashing*

Consistent Hashing digunakan untuk memetakan nama antrean ke node broker yang bertanggung jawab. Implementasinya di utils/consistent_hashing.py membuat sebuah "cincin" hash. Setiap node fisik dipetakan ke beberapa titik (node virtual) di cincin untuk distribusi yang lebih merata. Saat sebuah queue_name perlu dipetakan, ia di-hash ke sebuah titik di cincin, dan node pertama yang ditemui searah jarum jam adalah node yang bertanggung jawab. Keuntungannya adalah saat node ditambah atau dihapus, hanya sebagian kecil kunci yang perlu dipetakan ulang.

b. *At-Least-Once Delivery & Requeue on Timeout*

Sistem ini dirancang untuk jaminan pengiriman at-least-once. Dalam queue_manager.py, fungsi _internal_pop menggunakan perintah BLPOP dari Redis. BLPOP secara atomik mengambil dan menghapus elemen dari antrean. Jaminan at-least-once dicapai pada tingkat aplikasi konsumen: jika konsumen gagal setelah mengambil pesan tetapi sebelum selesai memprosesnya, pesan itu akan hilang. Untuk jaminan yang lebih kuat, konsumen harus mengimplementasikan logika acknowledgment (ACK), di mana pesan hanya dihapus secara permanen setelah diproses. Implementasi saat ini menyediakan fondasi untuk ini**.**

c. Failure Detection & Handover

Deteksi kegagalan dan handover tidak diimplementasikan secara otomatis dalam kode. Namun, arsitektur Consistent Hashing memfasilitasinya. Jika sebuah node gagal, hash ring dapat dibangun kembali tanpa node tersebut. Permintaan untuk antrean yang sebelumnya dikelola oleh node yang gagal akan secara otomatis dialihkan ke node berikutnya di cincin. Pemulihan keadaan (pesan yang tersisa di Redis) akan menjadi tanggung jawab node baru tersebut.

2.2.3 *Distributed Cache Coherence (Cache Coherence)*

a. MESI *Protocol (Directory-based Coherence)*

MESI Protocol (Directory-based Coherence) Sistem ini mengimplementasikan protokol koherensi yang meniru logika MESI untuk menjaga konsistensi data di antara cache lokal pada setiap node. Implementasinya di cache_manager.py mendefinisikan empat keadaan dalam CacheState enum: MODIFIED, EXCLUSIVE, SHARED, INVALID.

- Read Miss

  Jika data tidak ada di cache (INVALID), node akan menyiarkan snoop_read ke peer lain. Jika peer lain memiliki data dalam keadaan MODIFIED, data tersebut akan ditulis kembali ke memori utama (Redis) dan dibagikan. Jika tidak ada peer yang memilikinya, data dibaca dari memori utama dan keadaan menjadi EXCLUSIVE.

- Write

  Jika node ingin menulis ke blok SHARED, ia harus menyiarkan invalidate ke semua peer lain untuk membatalkan salinan mereka, kemudian mengubah keadaannya menjadi MODIFIED. Jika sudah EXCLUSIVE atau MODIFIED, ia dapat menulis secara lokal.

b. *Least Recently Used* (LRU)

Ketika cache penuh, sebuah item harus dikeluarkan. cache_manager.py menggunakan kebijakan LRU. Ini diimplementasikan menggunakan collections.OrderedDict. Setiap kali sebuah item diakses (cache_data.move_to_end(key)), ia dipindahkan ke akhir kamus. Saat eviksi diperlukan (_evict_if_needed), item pertama (yang paling lama tidak digunakan) akan dihapus. Jika item yang dikeluarkan berada dalam keadaan MODIFIED, ia akan ditulis kembali ke Redis sebelum dihapus.

BAB III
FITUR & PERFORMA

3.1 *Distributed Lock Manager*

3.1.1 Arsitektur

Arsitektur Lock Manager terdiri dari klaster node yang identik, masing-masing menjalankan instance RaftNode dan LockManager. Konfigurasi node didefinisikan melalui variabel lingkungan (NODE_ID, PEERS) yang dibaca oleh main.py dan diteruskan ke RaftNode.

```python
class LockManager:
    def __init__(self):
        self.locks = defaultdict(dict)

        self.wait_queue = defaultdict(list)

        self.logger = logging.getLogger("LockManager")
        self.raft_node = None # Akan di-set oleh main.py

    def set_raft_node(self, raft_node):
        self.raft_node = raft_node

    def apply_command(self, command):
        """
        Callback yang dipanggil oleh RaftNode SETELAH sebuah command di-commit.
        Logika di sini HARUS deterministik.
        """
        op = command['op']
        lock_id = command['lock_id']
        client_id = command['client_id']

        self.logger.info(f"Applying committed command: {command}")

        try:
            if op == 'acquire':
                lock_type = command['type']
                if lock_id not in self.locks:
                    # Kunci baru
                    self.locks[lock_id] = {'type': lock_type, 'owners': {client_id}}
                elif self.locks[lock_id]['type'] == 'shared' and lock_type == 'shared':
                    # Tambahkan shared owner
                    self.locks[lock_id]['owners'].add(client_id)
                else:
                    self.logger.error(f"Failed to apply {command}: Lock conflict detected in state machine.")

            elif op == 'release':
                if lock_id in self.locks and client_id in self.locks[lock_id]['owners']:
                    self.locks[lock_id]['owners'].remove(client_id)
                    if not self.locks[lock_id]['owners']:
                        # Jika tidak ada owner lagi, hapus kuncinya
                        del self.locks[lock_id]
                else:
                    self.logger.warning(f"Failed to apply {command}: Lock not held or client mismatch.")

            # Setelah state berubah, "bangunkan" request yang mungkin menunggu
            self._notify_waiters(lock_id)

        except Exception as e:
            self.logger.error(f"Error applying command {command}: {e}", exc_info=True)
```

Gambar 3.1 Konfigurasi lock-node.

Logika inti dari Leader diimplementasikan dalam run_leader di raft.py, yang secara berkala mengirimkan heartbeat dan mereplikasi log.

```python
async def run_leader(self):
    self.logger.info(f"Term {self.current_term}: Running as LEADER.")
    self.leader_id = self.node_id

    while self.state == NodeState.LEADER:
        # Kirim AppendEntries (heartbeat) ke semua peer
        tasks = [self._replicate_log_to_peer(peer) for peer in self.peers]
        await asyncio.gather(*tasks, return_exceptions=True)

        # Cek apakah kita bisa commit sesuatu
        await self._update_commit_index()

        await asyncio.sleep(self.heartbeat_interval)
```

Gambar 3.2 Snippet Kode Fungsi run_leader.

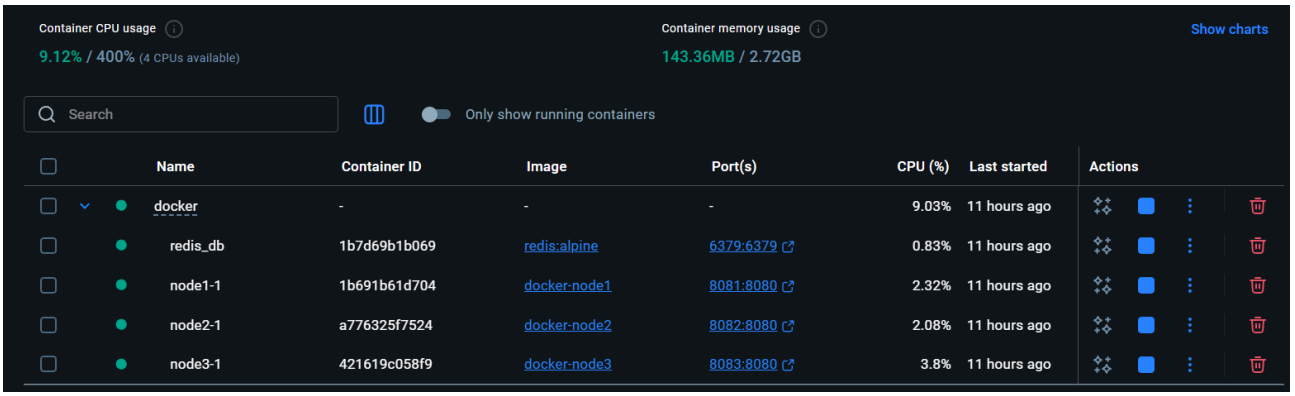
Kompatibilitas kunci (shared/exclusive) diperiksa oleh _check_lock_availability di lock_manager.py sebelum Leader mencoba mereplikasi perintah acquire.

```python
def _check_lock_availability(self, lock_id, lock_type):
    """Memeriksa state LOKAL (terreplikasi) saat ini."""
    if lock_id not in self.locks:
        return True # Tersedia

    current_lock = self.locks[lock_id]
    if current_lock['type'] == 'exclusive':
        return False # Sedang dipegang eksklusif

    if current_lock['type'] == 'shared' and lock_type == 'exclusive':
        return False # Sedang dipegang shared, tidak bisa ambil eksklusif

    return True # (current=shared, request=shared) -> OK
```

Gambar 3.3 Logika _check_lock_compatibility.

3.1.2 Deployment

Sistem dideploy menggunakan Docker dan Docker Compose. docker-compose.yml mendefinisikan layanan untuk tiga node (node1, node2, node3) dan satu layanan Redis. Setiap node dibangun dari Dockerfile.node yang sama. Variabel lingkungan di dalam docker-compose.yml digunakan untuk memberikan setiap node ID unik dan daftar semua peer dalam klaster.

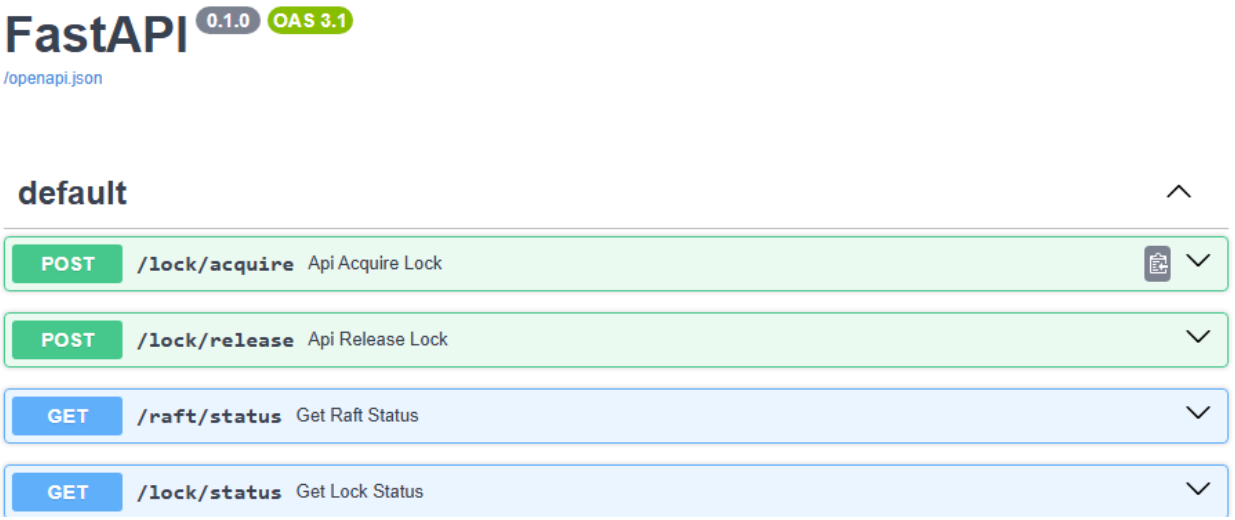


Gambar 3.4 *Container* yang Digunakan.

3.1.3 Fungsionalitas API

API untuk Lock Manager diekspos melalui FastAPI di main.py. Endpoint utamanya adalah:

- POST /lock/acquire: Untuk meminta kunci.
- POST /lock/release: Untuk melepaskan kunci.
- GET /raft/status: Untuk debugging, menampilkan status Raft node.
- GET /lock/status: Untuk debugging, menampilkan keadaan kunci yang saat ini dipegang.



Gambar 3.5 Swagger UI.

3.2 *Distributed Queue System*

3.2.1 Arsitektur

Arsitektur terdiri dari beberapa node broker, masing-masing menjalankan QueueManager. ConsistentHashing digunakan untuk memetakan antrean ke node. Jika permintaan diterima oleh node yang salah, QueueManager akan meneruskannya ke node yang benar melalui endpoint internal (/queue/internal_push, /queue/internal_pop). Persistensi pesan dijamin oleh Redis.

```python
class QueueManager:
    def __init__(self, node_id: str, hash_ring: ConsistentHashing):
        self.node_id = node_id
        self.hash_ring = hash_ring
        self.logger = logging.getLogger(f"QueueManager-{node_id.split(':')[-1]}")
        self.logger.info(f"Connecting to Redis at {REDIS_HOST}:{REDIS_PORT}")
        self.redis_client = redis.Redis(host=REDIS_HOST, port=REDIS_PORT, decode_responses=True)

        # Sesi HTTP untuk forwarding request ke node lain
        self.http_session = None

    async def _get_http_session(self):
        """Lazy-load http session."""
        if self.http_session is None:
            self.http_session = aiohttp.ClientSession()
        return self.http_session

    async def push_message(self, queue_name: str, message: str):
        """
        API untuk Klien: Mendorong pesan ke antrian.
        Menangani logika forwarding.
        """
        responsible_node = self.hash_ring.get_node(queue_name)
        self.logger.info(f"Queue '{queue_name}' -> mapped to node '{responsible_node}'")

        if responsible_node == self.node_id:
            # Jika node ini yang bertanggung jawab, dorong ke Redis
            return await self._internal_push(queue_name, message)
        else:
            # Jika bukan, teruskan (forward) request ke node yg benar
            return await self._forward_push_request(responsible_node, queue_name, message)

    async def pop_message(self, queue_name: str, timeout: int = 5):
        """
        API untuk Klien: Mengambil pesan dari antrian.
        Menangani logika forwarding.
        """
        responsible_node = self.hash_ring.get_node(queue_name)
        self.logger.info(f"Queue '{queue_name}' -> mapped to node '{responsible_node}'")

        if responsible_node == self.node_id:
            # Node ini yg bertanggung jawab, ambil dari Redis
            return await self._internal_pop(queue_name, timeout)
        else:
            # Bukan node ini, teruskan (forward) request
            return await self._forward_pop_request(responsible_node, queue_name, timeout)
```

Gambar 3.6 konfigurasi queue node

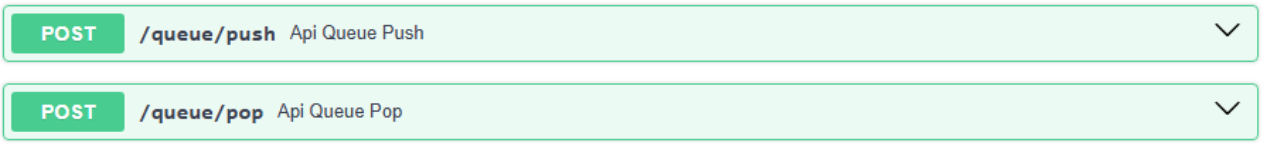
3.2.2 Deployment

Sama seperti Lock Manager, sistem antrean dideploy sebagai bagian dari layanan node yang sama di docker-compose.yml. Semua node terhubung ke instance Redis yang sama.

3.2.3 Fungsionalitas API

API publik diekspos di main.py:

- POST /queue/push: Produsen mengirim pesan ke antrean.
- POST /queue/pop: Konsumen mengambil pesan dari antrean.



Gambar 3.7 Swagger UI

3.3 *Distributed Cache Coherence*

3.3.1 Arsitektur

Setiap node menjalankan CacheManager yang mengelola cache lokal (OrderedDict) dan statusnya (dict). Komunikasi antar-node untuk snoop dan invalidasi terjadi melalui endpoint HTTP internal (/cache/handle_snoop_read, /cache/handle_invalidate). Redis bertindak sebagai memori utama.

```python
class CacheManager:
    def __init__(self, node_id: str, peers: list[str], cache_capacity: int = 100):
        self.node_id = node_id
        self.peers = [p for p in peers if p != self.node_id]
        self.logger = logging.getLogger(f"CacheManager-{node_id.split(':')[-1]}")
        self.logger.info(f"Connecting to Redis at {REDIS_HOST}:{REDIS_PORT}")

        redis_connection = redis.Redis(host=REDIS_HOST, port=REDIS_PORT, decode_responses=True)
        self.redis_client = redis_connection
        self.main_memory = redis_connection

        self.cache_data = OrderedDict()

        self.cache_states = {}
        self.cache_capacity = cache_capacity

        self.http_session = None

    async def _get_http_session(self):
        if self.http_session is None or self.http_session.closed:
            self.http_session = aiohttp.ClientSession()
        return self.http_session

    async def _read_from_main_memory(self, key: str):
        self.logger.debug(f"Cache MISS. Reading '{key}' from Main Memory (Redis)...")
        return await self.main_memory.get(key)

    async def _write_to_main_memory(self, key: str, value: str):
        self.logger.debug(f"Writing '{key}' to Main Memory (Redis)...")
        await self.main_memory.set(key, value)
    async def read_data(self, key: str):
        """API Publik: Klien (misal, CPU) ingin membaca data."""

        state = self.cache_states.get(key)
        if state in (CacheState.MODIFIED, CacheState.EXCLUSIVE, CacheState.SHARED):
            self.logger.info(f"READ Hit: Key '{key}' state={state.value}. Mengambil dari cache lokal.")
            self.cache_data.move_to_end(key)
            return self.cache_data[key]
        self.logger.info(f"READ Miss: Key '{key}' state={state}.")
        snoop_responses = await self._broadcast_snoop_read(key)

        data_from_peer = None
        peer_had_modified = False

        for peer_data, peer_state in snoop_responses:
            if peer_state in (CacheState.MODIFIED, CacheState.EXCLUSIVE, CacheState.SHARED):
                data_from_peer = peer_data
                if peer_state == CacheState.MODIFIED:
                    peer_had_modified = True
```

Gambar 3.8 konfigurasi cache coherence

3.3.2 Deployment

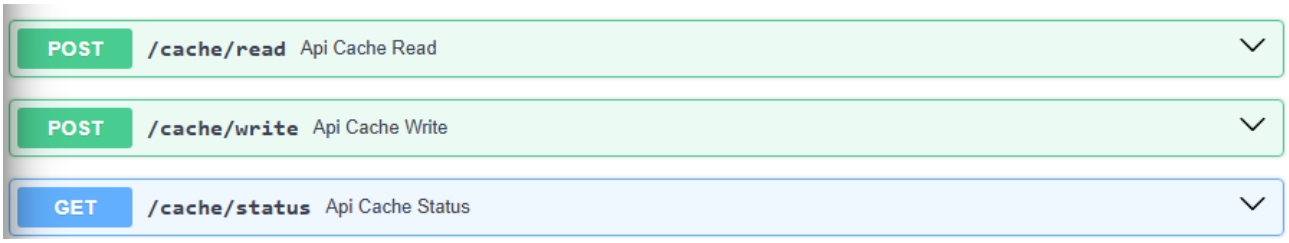
Komponen cache adalah bagian integral dari setiap layanan node yang didefinisikan dalam docker-compose.yml. Semua node berkomunikasi satu sama lain melalui jaringan Docker dan dengan Redis.

3.3.3 Fungsionalitas API

API publik di main.py mensimulasikan interaksi CPU dengan cache:

- POST /cache/read: Membaca data, memicu logika cache hit/miss.
- POST /cache/write: Menulis data, memicu logika invalidasi atau transisi ke Modified.

- GET /cache/status: Endpoint debug untuk melihat keadaan cache lokal.



Gambar 3.9 Swagger UI

3.4 *Containerization*

Seluruh sistem dikontainerisasi menggunakan Docker:

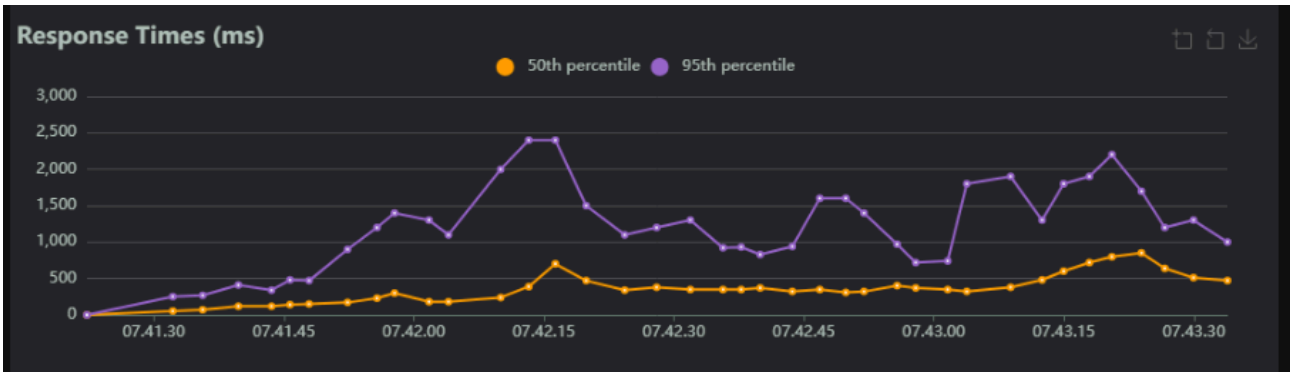
Dockerfile.node untuk mendefinisikan image untuk aplikasi Python, menginstal dependensi dari requirements.txt, dan menyalin kode sumber. docker-compose.yml guna mengorkestrasi seluruh sistem. Ini mendefinisikan jaringan (distributed_net) agar container dapat berkomunikasi, dan meluncurkan tiga container node aplikasi serta satu container Redis. Ini juga menangani pemetaan port dan injeksi variabel lingkungan untuk konfigurasi.

BAB IV
ANALISIS PERFORMA

4.1 *Distributed Lock Manager*

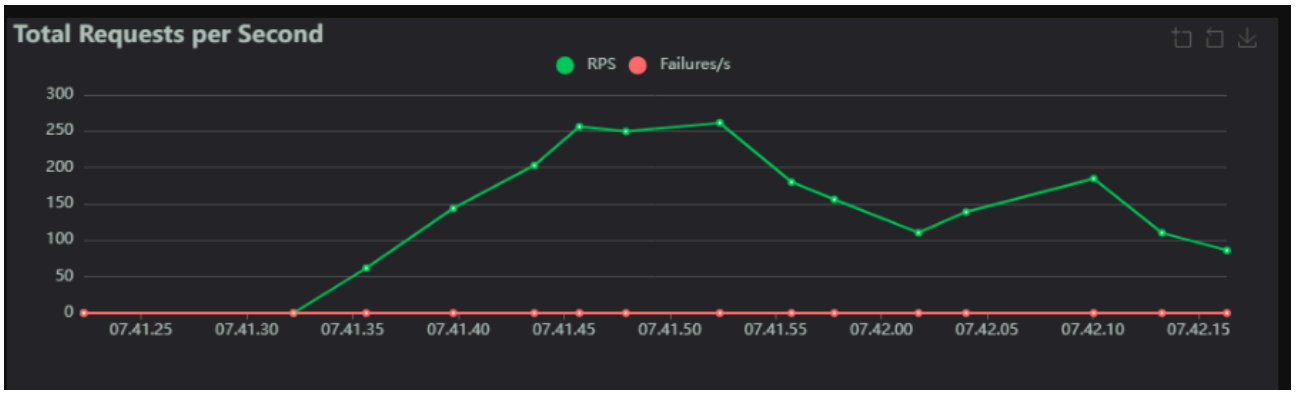
Analisis kinerja Lock Manager berfokus pada latensi dan throughput operasi acquire/release.

- Latency: Latensi untuk operasi tulis (acquire/release) diharapkan sedikit meningkat seiring penambahan jumlah node ke klaster Raft. Ini karena Leader harus menunggu konfirmasi dari mayoritas yang lebih besar.



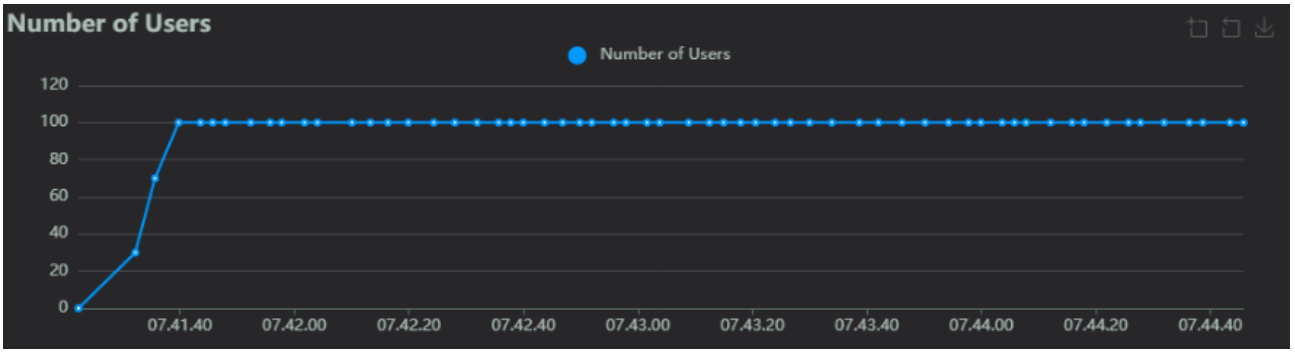
Gambar 4.1 Latency Lock Manager

- Throughput: Throughput puncak akan dibatasi oleh kecepatan Leader dalam mencapai konsensus. Skenario high contention (banyak klien memperebutkan satu kunci) dari load_test_scenarios.py akan menunjukkan batas throughput ini.



Gambar 4.2 Throughtput Lock Manager

- Scalability: Skalabilitas tulis untuk satu kunci tidak diharapkan meningkat dengan penambahan node. Namun, sistem dapat menangani lebih banyak kunci yang berbeda secara paralel.

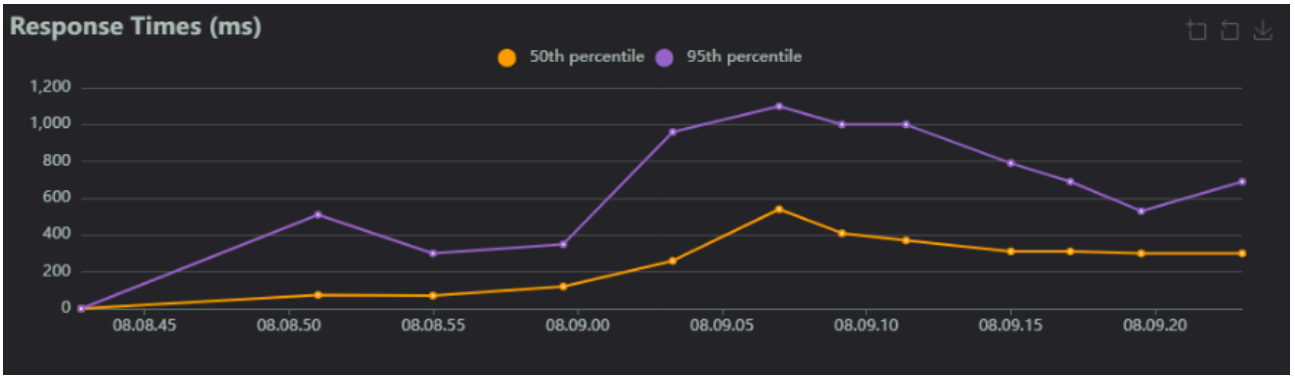


Gambar 4.3 Scalability Lock Manager

4.2 *Distributed Queue System*

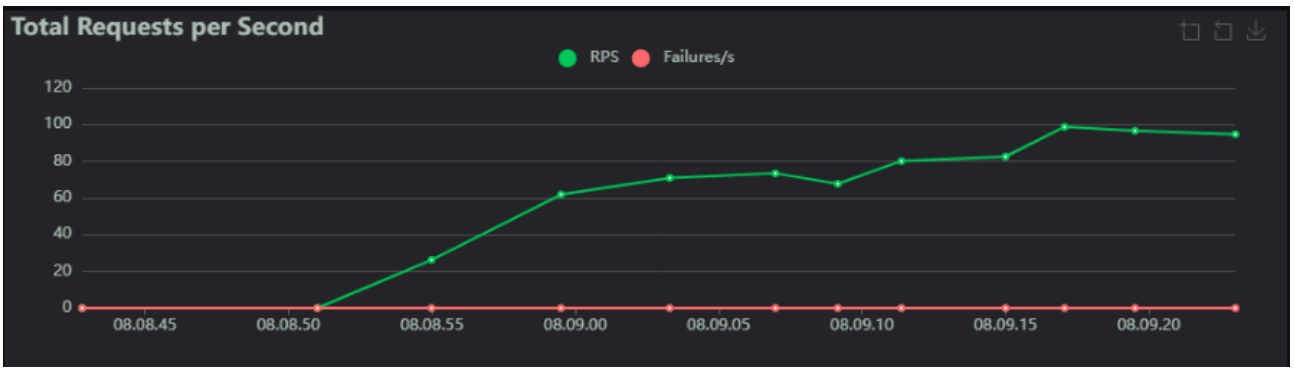
Analisis kinerja sistem antrean berfokus pada skalabilitas horizontal.

- Latency: Latensi akan terdiri dari waktu pemrosesan Redis ditambah latensi jaringan. Untuk permintaan yang perlu diteruskan, latensi akan mencakup satu network hop tambahan.



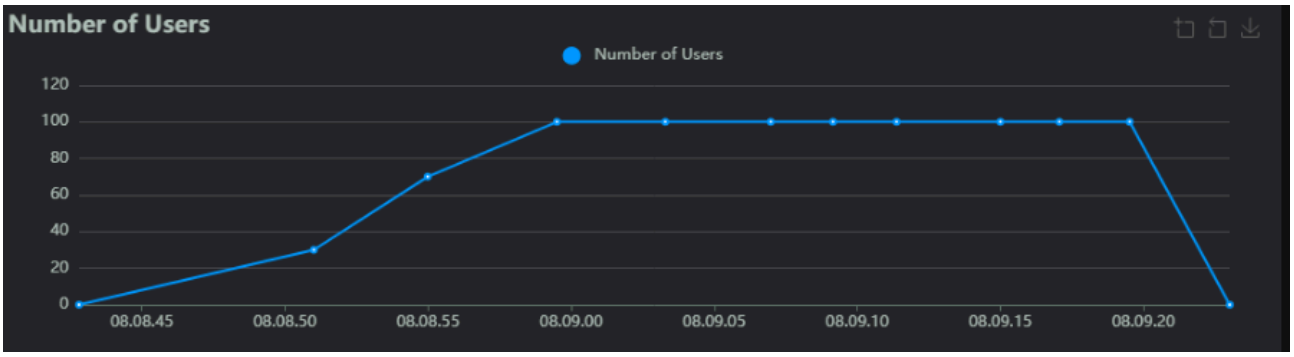
Gambar 4.4 Latency Queue System

- Throughput: Throughput agregat sistem diharapkan meningkat secara linear dengan penambahan node. Karena Consistent Hashing mendistribusikan antrean, setiap node menangani sebagian dari total beban, sehingga penambahan node akan meningkatkan kapasitas total sistem.



Gambar 4.5 Throughput Queue System

- Scalability: Ini adalah kekuatan utama arsitektur ini. Sistem dapat diskalakan dengan mudah dengan menjalankan lebih banyak container node**.**

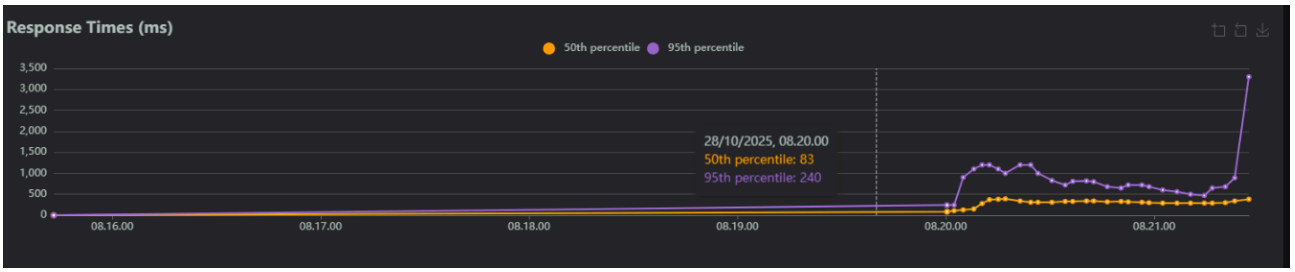


Gambar 4.6 Scalability Queue System

4.3 *Distributed Cache Coherence*

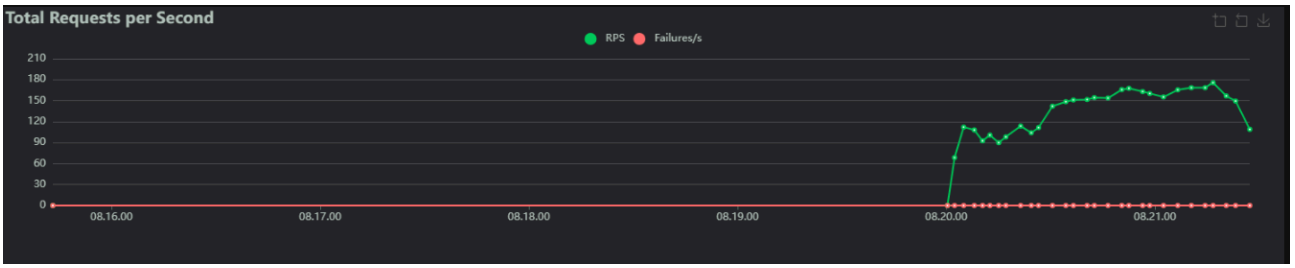
Analisis kinerja cache berfokus pada efektivitasnya dalam mengurangi latensi dan overhead koherensi.

- Latency: Pada cache hit, latensi akan sangat rendah (akses memori lokal). Pada cache miss, latensi akan jauh lebih tinggi, melibatkan beberapa perjalanan bolak-balik jaringan (ke peer lain untuk snoop, atau ke Redis). Latensi tulis pada data yang dibagikan juga akan lebih tinggi karena harus menunggu ACK invalidasi dari peer.



Gambar 4.7 Latency Cache Coherence

- Throughput: Throughput akan sangat bergantung pada cache hit rate. Beban kerja dengan lokalitas data yang tinggi (pola akses Zipfian) akan menghasilkan throughput yang jauh lebih tinggi daripada beban kerja dengan akses acak.



Gambar 4.8 Throughput Cache Coherence

- Comparison: Dibandingkan dengan sistem tanpa cache (single-node dalam konteks ini berarti selalu ke Redis), sistem cache akan menunjukkan latensi rata-rata yang jauh lebih rendah untuk beban kerja baca-berat dengan lokalitas yang baik.

BAB V
KESIMPULAN DAN TANTANGAN

5.1 Kesimpulan

Proyek ini berhasil mengimplementasikan sistem terdistribusi fungsional yang mengintegrasikan tiga komponen penting untuk sinkronisasi dan konsistensi.

- Distributed Lock Manager berbasis Raft menyediakan mekanisme sinkronisasi yang kuat dan toleran terhadap kegagalan.
- Distributed Queue System berbasis Consistent Hashing menawarkan platform komunikasi asinkron yang dapat diskalakan secara horizontal.
- Distributed Cache Coherence yang meniru logika MESI secara efektif mengurangi latensi untuk beban kerja yang sesuai, sambil menjaga konsistensi data.

Integrasi komponen-komponen ini dalam satu platform yang dapat dideploy dengan kontainer menunjukkan pendekatan praktis untuk membangun sistem terdistribusi yang andal dan berkinerja.

5.2 Tantangan

Kompleksitas Debugging

Men-debug perilaku yang muncul dari interaksi asinkron antara beberapa node adalah tantangan utama. Masalah seperti race condition atau split vote dalam Raft memerlukan logging yang cermat dan pengujian yang teliti.

Penanganan Kegagalan

Kode yang disediakan mengimplementasikan logika kondisi stabil. Menambahkan penanganan kegagalan yang kuat (misalnya, pemulihan node Raft setelah crash, handover antrean otomatis) akan menambah kompleksitas yang signifikan.

Kinerja Jaringan

Kinerja keseluruhan sistem sangat bergantung pada latensi dan keandalan jaringan antar-node. Dalam pengujian, ini disimulasikan oleh jaringan virtual Docker, tetapi di lingkungan nyata, partisi jaringan dan latensi yang bervariasi adalah tantangan konstan.

Tuning Performa

Menemukan parameter optimal (misalnya, election timeout Raft, timeout pop antrean, ukuran cache) memerlukan benchmarking dan tuning yang ekstensif di bawah beban kerja yang realistis.