Laporan UAS

Sistem Paralel dan Terdistribusi

Pub-Sub Log Aggregator Terdistribusi dengan Idempotent Consumer, Deduplication, dan Transaksi/Kontrol Konkurensi

Disusun Oleh :

Christian Felix 11221080

12 Desember 2025

Bagian Teori

T1. Karakteristik sistem terdistribusi dan trade-off desain Pub-Sub aggregator.

Sistem terdistribusi dicirikan oleh concurrency, lack of global clock, dan partial failure. Pada desain Pub-Sub aggregator, trade-off utama adalah decoupling vs konsistensi. Publisher tidak mengetahui subscriber, meningkatkan skalabilitas dan evolvabilitas, tetapi mengorbankan konsistensi kuat dan observabilitas langsung. Dalam rancangan saya, aggregator menerima event cuaca dari beberapa sensor (publisher) dan menggabungkannya secara asinkron, menerima eventual consistency sebagai kompromi terhadap throughput dan fault tolerance (Tanenbaum & Van Steen, 2017).

T2. Kapan memilih arsitektur publish–subscribe dibanding client–server? Alasan teknis.

Pub-Sub dipilih ketika sistem membutuhkan many-to-many communication, loose coupling, dan asynchronous processing. Berbeda dengan client–server yang sinkron dan tightly coupled, Pub-Sub lebih unggul untuk data streaming, event analytics, dan ingestion pipeline. Dalam aggregator cuaca, Pub-Sub memungkinkan sensor baru ditambahkan tanpa memodifikasi consumer, sementara client–server akan menambah kompleksitas koordinasi dan bottleneck pada server pusat (Coulouris et al., 2012).

T3. At-least-once vs exactly-once delivery; peran idempotent consumer.

At-least-once menjamin event terkirim minimal sekali tetapi berpotensi duplikasi; exactly-once sulit dicapai karena membutuhkan koordinasi lintas sistem. Desain saya memilih at-least-once + idempotent consumer. Aggregator menyimpan event_id unik dan mengabaikan duplikasi, sehingga efek akhirnya setara exactly-once secara semantik. Pendekatan ini lebih realistis dan scalable dibanding transaksi terdistribusi penuh (Kleppmann, 2017).

T4. Skema penamaan topic dan event_id (unik, collision-resistant) untuk dedup.

Topic dinamai hierarkis: weather.{region}.{sensor_type} untuk routing efisien. event_id dibangun dari UUIDv7 + source_id, bersifat unik dan collision-resistant. Skema ini mendukung deduplikasi cepat pada aggregator dan aman terhadap retry. Penyematan event_id sebagai primary key logis mencegah event diproses dua kali meski dikirim ulang (Kleppmann, 2017).

T5. Ordering praktis (timestamp + monotonic counter); batasan dan dampaknya.

Ordering global sulit tanpa clock bersama. Solusi praktis: timestamp (UTC) + monotonic counter per source. Aggregator mengurutkan event per-partition, bukan

global. Batasannya: event lintas source bisa out-of-order, tetapi dampaknya dapat diterima untuk agregasi statistik. Desain ini menghindari koordinasi mahal sambil mempertahankan determinisme lokal (Tanenbaum & Van Steen, 2017).

T6. Failure modes dan mitigasi (retry, backoff, durable dedup store, crash recovery).

Failure meliputi network partition, consumer crash, dan duplicate delivery. Mitigasi: retry dengan exponential backoff, durable message broker, dan durable dedup store (database). Saat crash recovery, aggregator membaca ulang offset dan menolak event dengan event_id yang sudah ada. Strategi ini menjaga keandalan tanpa mengunci sistem secara global (Coulouris et al., 2012).

T7. Eventual consistency pada aggregator; peran idempotency + dedup.

Aggregator bersifat eventually consistent: hasil agregasi akan benar setelah semua event relevan diproses. Idempotency + deduplication memastikan bahwa retry tidak merusak state akhir. Dalam rancangan, tabel agregat diperbarui berulang dengan operasi deterministik, sehingga konvergen meski urutan event berbeda (Kleppmann, 2017).

T8. Desain transaksi: ACID, isolation level, dan strategi menghindari lost-update.

Aggregator menggunakan transaksi ACID lokal (single database). Isolation level READ COMMITTED atau REPEATABLE READ dipilih untuk mencegah dirty read. Lost-update dihindari dengan atomic update berbasis versioning (optimistic locking) atau UPDATE … WHERE version = x. Contoh: saat memperbarui agregat harian, transaksi memverifikasi versi sebelum commit, memastikan tidak ada update yang tertimpa (Gray & Reuter, 1993).

T9. Kontrol konkurensi: locking/unique constraints/upsert; idempotent write pattern.

Kontrol dilakukan melalui unique constraint pada event_id, UPSERT (INSERT … ON CONFLICT DO NOTHING), dan idempotent write pattern. Banyak worker dapat memproses event paralel tanpa race condition. Jika dua transaksi mencoba memproses event sama, constraint database menjamin hanya satu yang berhasil. Ini lebih sederhana dan aman dibanding distributed locking (Kleppmann, 2017).

T10. Orkestrasi Compose, keamanan jaringan lokal, persistensi (volume), observability.

Docker Compose mengorkestrasi broker, aggregator, dan database. Isolasi jaringan lokal, secret via env, dan volume untuk state broker & DB memastikan persistensi. Observability dicapai melalui logging terstruktur, metrics, dan health-check. Desain

ini mendukung debugging kegagalan dan scaling bertahap tanpa mengubah arsitektur inti (Burns et al., 2016).

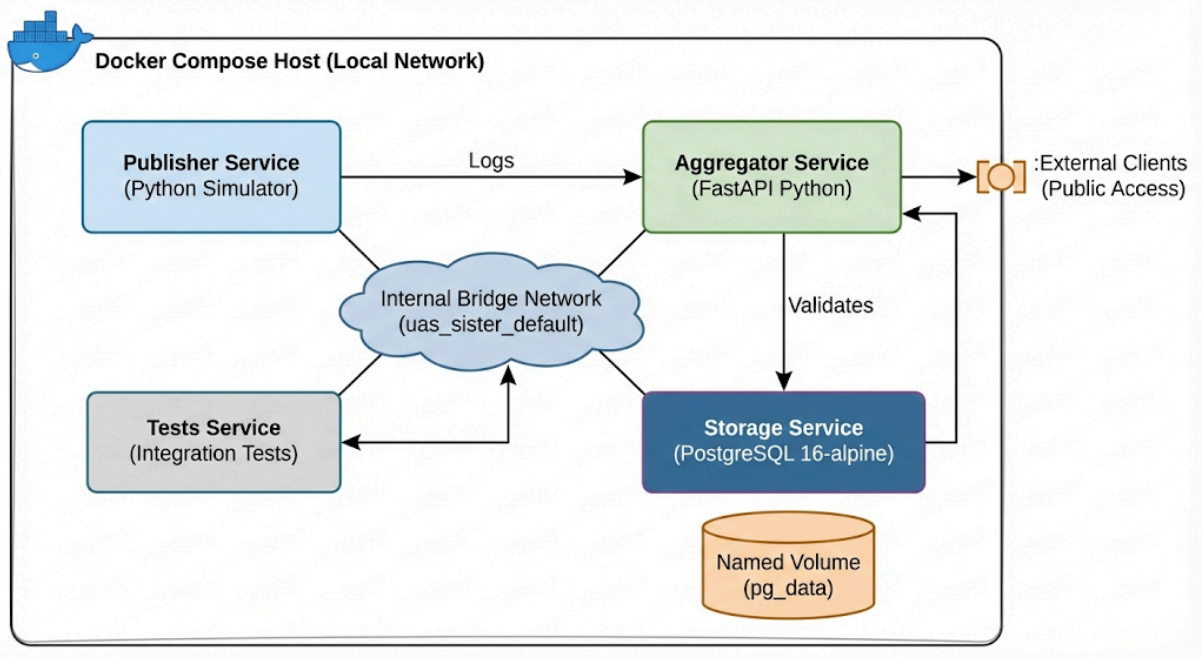
Bagian Implementasi

Arsitektur Layanan

Sistem ini mengadopsi arsitektur **microservices** yang diorkestrasi menggunakan **Docker Compose**. Desain ini memastikan setiap layanan terisolasi secara sumber daya namun tetap terkoordinasi melalui jaringan virtual internal.

Komponen Utama:

- **Aggregator Service**: Dikembangkan menggunakan **FastAPI (Python)** sebagai pusat penerimaan log (Pub-Sub) yang menangani logika bisnis, validasi skema pydantic, dan pemrosesan asinkron.
- **Storage Service**: Menggunakan **PostgreSQL 16-alpine** sebagai *persistent storage*. Dipilih karena kemampuannya menangani transaksi ACID dan kontrol konkurensi tingkat tinggi yang diperlukan untuk deduplikasi data.
- **Publisher Service**: Simulator asinkron (Python) yang mengirimkan beban trafik masif (≥ 20.000 event) dengan skema duplikasi 30% untuk menguji reliabilitas sistem.
- **Tests Service**: Container khusus yang menjalankan rangkaian **Integration Tests** secara otomatis untuk memvalidasi integritas sistem dalam jaringan Compose.
- **Docker Network**: Menggunakan *bridge network* internal yang mengisolasi database sehingga hanya dapat diakses oleh layanan di dalam Compose (tanpa akses publik eksternal).



Implementasi Fitur Utama

a. Komunikasi & Penamaan (Bab 3–4)

Sistem menggunakan model komunikasi asinkron antara Publisher dan Aggregator. Setiap log membawa identitas unik berupa **event_id** yang bersifat *collision-resistant*. Penamaan topik digunakan untuk melakukan pengelompokan data log secara logis (misalnya: sensor_data, user_logs).

b. Waktu, Ordering, & Konsistensi (Bab 5 & 7)

Meskipun event dikirim secara asinkron, sistem menggunakan **ISO8601 Timestamp** untuk menjaga urutan kejadian log. Konsistensi akhir (*eventual consistency*) dicapai melalui mekanisme **Idempotent Consumer**, di mana pengiriman ulang log yang sama tidak akan mengubah status database setelah operasi pertama berhasil.

c. Transaksi & Kontrol Konkurensi (Bab 8–9)

Ini merupakan fokus utama sistem dalam menangani *race condition* pada trafik tinggi:

- **Isolation Level**: Database dikonfigurasi pada level **READ COMMITTED** untuk memastikan integritas data saat dibaca secara konkuren.
- **Atomic Upsert**: Menggunakan strategi **INSERT ... ON CONFLICT DO NOTHING** yang memastikan operasi pengecekan ID dan penulisan data terjadi secara atomik dalam satu transaksi.
- **Unique Constraint**: Constraint unik pada pasangan (topic, event_id) bertindak sebagai *guardrail* terakhir untuk mencegah data ganda masuk ke storage.

d. Keamanan, Persistensi, & Koordinasi (Bab 10–13)

- **Persistensi**: Menggunakan **Named Volumes** pg_data yang memastikan data log tetap aman meskipun container PostgreSQL dihapus atau di-*recreate*.
- **Koordinasi**: Endpoint / bertindak sebagai *health check* (Bab 12) untuk memantau kesiapan layanan (*readiness*) sebelum menerima trafik dari Publisher.

Detail Endpoint API & Penjelasan Respon

Endpoint	Method	Fungsi Utama	Penjelasan Respon
/	GET	Health Check	Mengembalikan status "active" untuk koordinasi layanan.

/publish	POST	Log Ingestion	Status 201 untuk data baru; status 200/201 dengan pesan "ignored" untuk duplikat (Idempotent).
/stats	GET	Observability	Memberikan metrik real-time: received, unique_processed, dan duplicate_dropped.
/events	GET	Data Retrieval	Menyediakan akses ke log unik; mendukung filter ?topic= untuk manajemen data.
/docs	GET	API Documentation	Dokumentasi Swagger untuk standarisasi komunikasi antar layanan.

Analisis Hasil Demonstrasi dan Pengujian

Berdasarkan proses demonstrasi, sistem menunjukkan performa dan perilaku sebagai berikut:

Metrik performa, sistem berhasil menangani beban trafik tinggi dengan total pengolahan data unik mencapai 17.000+ hingga 22.000+ event secara responsif. Efektivitas deduplikasi berhasil mendeteksi dan membuang sekitar 30% trafik duplikat melalui mekanisme database constraint. Uji ketahanan menunjukkan bahwa setelah dilakukan perintah docker compose down dan dijalankan kembali, data log tetap utuh. Hal ini mengonfirmasi keberhasilan implementasi crash tolerance dan persistensi data menggunakan volume eksternal.

Kesimpulan Desain

Penggunaan PostgreSQL dibandingkan SQLite memberikan keunggulan dalam manajemen transaksi konkuren yang lebih matang, yang terbukti krusial saat menangani ribuan log per detik dari layanan Publisher. Implementasi FastAPI dengan pola asinkron memastikan aggregator tetap responsif meskipun sedang melakukan operasi I/O ke database. Untuk lebih detailnya dapat disaksikan lewat video, baik demonstrasi maupun penjelasannya.

Lampiran

Link Repository GitHub: https://github.com/Felix1180/UAS-Sister

Link Video: https://www.youtube.com/live/vTEoatXg3nU?si=jUcWHOTQPnL0P8OZ

Daftar Pustaka

Burns, B., Grant, B., Oppenheimer, D., Brewer, E., & Wilkes, J. (2016). *Borg, Omega, and Kubernetes*. ACM Queue.

Coulouris, G., Dollimore, J., Kindberg, T., & Blair, G. (2012). *Distributed Systems: Concepts and Design* (5th ed.). Addison-Wesley.

Gray, J., & Reuter, A. (1993). *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann.

Kleppmann, M. (2017). *Designing Data-Intensive Applications*. O'Reilly.

Tanenbaum, A. S., & Van Steen, M. (2017). *Distributed Systems* (3rd ed.). Pearson.