Pub-Sub Log Aggregator Terdistribusi dengan Idempotent Consumer, Deduplikasi, dan Kontrol Transaksi

Faiz Ahnaf Samudra Azis♠

♠Institut Teknologi Kalimantan
Program Studi Informatika
Balikpapan, Kalimantan Timur, Indonesia

Abstrak

Laporan ini menyajikan implementasi sistem Pub-Sub Log Aggregator terdistribusi yang berfokus pada jaminan konsistensi data melalui mekanisme idempotent consumer dan kontrol transaksi. Sistem dibangun menggunakan arsitektur microservices dengan empat komponen: Aggregator (FastAPI), Publisher, Broker (Redis), dan Storage (PostgreSQL). Implementasi menerapkan pola idempotent consumer untuk mencapai semantik exactly-once processing, mekanisme deduplikasi persisten menggunakan unique constraint database, serta transaksi ACID dengan isolation level READ COMMITTED. Hasil pengujian menunjukkan sistem mampu memproses lebih dari 20.000 event dengan tingkat duplikasi 35% secara konsisten tanpa terjadi race condition.

Kata Kunci: Sistem Terdistribusi, Publish-Subscribe, Idempotent Consumer, Deduplikasi, Transaksi ACID, Docker Compose

1 Pendahuluan

Sistem terdistribusi modern menghadapi tantangan dalam menjaga konsistensi data ketika komponen-komponen berjalan secara paralel dan komunikasi jaringan tidak selalu reliable (Coulouris et al., 2012). Arsitektur publish-subscribe (Pub-Sub) menawarkan decoupling antara pengirim dan penerima pesan, namun menimbulkan kompleksitas dalam penanganan duplikasi dan ordering (Steen & Tanenbaum, 2023).

Laporan ini menyajikan implementasi Pub-Sub Log Aggregator yang mengatasi tantangan tersebut melalui: (1) pola idempotent consumer untuk mencegah pemrosesan ganda, (2) deduplikasi berbasis unique constraint PostgreSQL, dan (3) transaksi ACID untuk menjamin konsistensi data pada operasi konkuren.

2 Arsitektur Sistem

Sistem terdiri dari empat komponen yang berjalan dalam jaringan Docker Compose (Khannedy, 2023a):

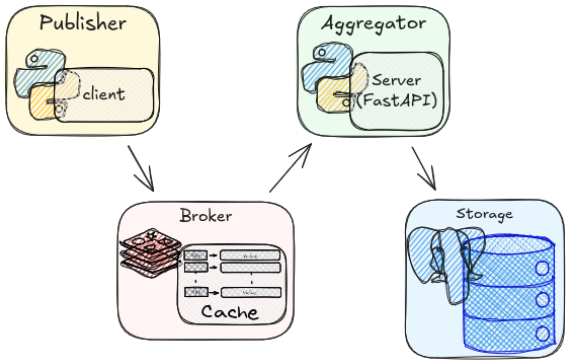

Figure 1: Arsitektur sistem Pub-Sub Log Aggregator

1. **Aggregator**: Layanan utama berbasis FastAPI (Sebastián Ramírez, 2024) yang menyediakan REST API untuk penerimaan event dan menjalankan consumer untuk memproses antrian pesan.
2. **Publisher**: Generator event yang mensimulasikan kondisi jaringan dengan menyuntikkan 35% pesan duplikat untuk menguji mekanisme deduplikasi.
3. **Broker**: Redis (Khannedy, 2023b) sebagai message queue yang memungkinkan time decoupling antara Publisher dan Aggregator.
4. **Storage**: PostgreSQL 18 sebagai penyimpanan persisten dengan dukungan transaksi ACID dan unique constraint untuk deduplikasi atomik.

2.1 Model Event

Setiap event menggunakan format JSON dengan struktur sebagai berikut:

Format JSON

```
{
  "topic": "auth.login",
  "event_id": "550e8400-e29b-41d4-a716-446655440000",
  "timestamp": "2025-12-15T10:30:00Z",
  "source": "user-service",
  "payload": { "user_id": 123, "action": "login_success" }
}
```

Table 1: Contoh format Event JSON

Field event_id menggunakan UUID v4 untuk menjamin keunikan global, sedangkan topic menggunakan format hierarkis (dot notation) untuk pengelolaan namespace.

ACM ISBN 979-8-0000-0000-0/00/00…$15.00
https://github.com/fzhnf/pub-sub-log-aggregator-v2

2.2 API Endpoints

Endpoint	Fungsi
GET /health	Health check untuk liveness probe
POST /publish	Menerima single event
POST /publish/batch	Menerima batch event dalam satu transaksi
GET /events	Mengambil daftar event berdasarkan topic
GET /stats	Statistik: received, processed, duplicates, uptime

Table 2: Daftar API Endpoints

2.3 Skema Database

Database memiliki dua tabel utama:

Skema SQL

```sql
-- Tabel untuk menyimpan event yang sudah diproses
CREATE TABLE processed_events (
    id SERIAL PRIMARY KEY,
    topic VARCHAR(255) NOT NULL,
    event_id VARCHAR(255) NOT NULL,
    timestamp TIMESTAMPTZ NOT NULL,
    source VARCHAR(255) NOT NULL,
    payload JSONB NOT NULL,
    processed_at TIMESTAMPTZ DEFAULT CURRENT_TIMESTAMP,
    CONSTRAINT unique_topic_event UNIQUE (topic, event_id)
);

-- Tabel singleton untuk statistik sistem
CREATE TABLE stats (
    id INTEGER PRIMARY KEY DEFAULT 1 CHECK (id = 1),
    received INTEGER DEFAULT 0,
    unique_processed INTEGER DEFAULT 0,
    duplicate_dropped INTEGER DEFAULT 0,
    started_at TIMESTAMPTZ DEFAULT CURRENT_TIMESTAMP
);
```

Table 3: Skema database dengan unique constraint untuk deduplikasi

3 Keputusan Desain

3.1 Strategi Idempotency dan Deduplikasi

Dalam sistem terdistribusi, jaringan yang tidak reliable menyebabkan pesan dapat dikirim lebih dari sekali (at-least-once delivery). Untuk mencegah pemrosesan ganda, sistem menerapkan pola Idempotent Consumer (Ahmed, 2025; Kumar, 2023).

Pendekatan yang digunakan:

1. Setiap event memiliki identifier unik berupa kombinasi (topic, event_id)
2. Database menyimpan record setiap event yang telah diproses
3. Operasi insert menggunakan klausa ON CONFLICT DO NOTHING untuk menolak duplikat secara atomik

Query SQL

```sql
INSERT INTO processed_events (topic, event_id, timestamp, source, payload)
VALUES ($1, $2, $3, $4, $5)
ON CONFLICT (topic, event_id) DO NOTHING
RETURNING id
```

Table 4: Query idempotent insert dengan conflict handling

Keunggulan pendekatan ini adalah deduplikasi dilakukan secara atomik di level database, sehingga tidak memerlukan locking eksplisit di level aplikasi.

3.2 Transaksi dan Kontrol Konkurensi

Untuk mencegah race condition pada operasi konkuren, sistem menerapkan transaksi ACID (Coulouris et al., 2012):

- **Atomicity**: Seluruh batch event berhasil atau gagal secara keseluruhan
- **Consistency**: Unique constraint menjaga invariant tidak ada event duplikat
- **Isolation**: Menggunakan level READ COMMITTED
- **Durability**: Data tersimpan di PostgreSQL dengan Docker named volume

Kode Python

```python
async with db.transaction() as conn:
    for event in events:
        is_new = await db.insert_event(conn, event)
    await db.update_stats(conn, received, processed, duplicates)
```

Table 5: Transaksi atomik untuk pemrosesan batch

3.2.1 Pemilihan Isolation Level

Sistem menggunakan READ COMMITTED dengan pertimbangan:

Isolation Level	Keuntungan	Kerugian
READ COMMITTED	Throughput tinggi, tidak ada deadlock	Mungkin phantom read
REPEATABLE READ	Mencegah non-repeatable read	Overhead lebih tinggi
SERIALIZABLE	Konsistensi maksimal	Throughput rendah, risiko deadlock

Table 6: Perbandingan isolation level

READ COMMITTED dipilih karena unique constraint sudah menjamin tidak ada duplicate insert, sehingga tidak memerlukan isolation level yang lebih ketat. Risiko phantom read tidak relevan karena operasi utama adalah insert, bukan select-for-update.

3.3 Ordering dan Reliability

Sistem tidak menjamin total ordering karena biaya koordinasi yang tinggi. Sebagai gantinya, sistem menggunakan partial ordering berbasis timestamp event. Untuk use case log aggregation, kelengkapan data lebih diprioritaskan daripada urutan absolut (Steen & Tanenbaum, 2023).

Reliability dicapai melalui:

- Persistensi data menggunakan Docker named volumes
- Redis sebagai buffer saat Aggregator tidak tersedia
- Mekanisme deduplikasi yang mencegah reprocessing setelah crash recovery
- Retry dengan exponential backoff dari sisi Publisher

4 Analisis Performa

4.1 Hasil Stress Test

Pengujian dilakukan dengan mengirimkan 20.000 event dengan tingkat duplikasi 35%:

Metrik	Nilai
Total Event Dikirim	20.000
Tingkat Duplikasi	35%
Event Unik Tersimpan	13.000
Duplikat Dibuang	7.000
Throughput	5.864 event/detik
Waktu Eksekusi	3,41 detik

Table 7: Hasil pengujian stress test

4.2 Hasil Uji Konkurensi

Pengujian dengan 10 worker paralel yang mengirimkan event dengan event_id yang sama secara bersamaan:

Metrik	Hasil
Jumlah Worker	10
Event per Worker	500
Total Event	5.000
Total Diproses	5.000
Waktu Eksekusi	0,41 detik
Throughput	12.115 event/detik

Table 8: Hasil pengujian konkurensi dengan 10 worker paralel

Hasil ini membuktikan bahwa mekanisme INSERT ON CONFLICT dan unique constraint efektif mencegah race condition tanpa memerlukan pessimistic locking.

5 Keterkaitan dengan Teori Sistem Terdistribusi

5.1 T1: Karakteristik Sistem Terdistribusi (Bab 1)

Menurut (Coulouris et al., 2012), sistem terdistribusi memiliki tiga karakteristik utama: konkurensi, ketiadaan jam global, dan kegagalan independen. Sistem Pub-Sub Log Aggregator ini mendemonstrasikan ketiga karakteristik tersebut.

Konkurensi ditunjukkan melalui pemrosesan paralel oleh multiple worker yang mengambil pesan dari Redis queue secara bersamaan. Setiap worker berjalan sebagai proses independen yang dapat memproses event tanpa menunggu worker lain.

Ketiadaan jam global ditangani dengan tidak bergantung pada sinkronisasi waktu antar komponen. Setiap event membawa timestamp sendiri dari sumber asalnya, dan sistem tidak mengasumsikan ordering berdasarkan waktu sistem.

Kegagalan independen terlihat dari isolasi antar komponen: Publisher dapat terus mengirim pesan ke Redis meskipun Aggregator sedang down. Pesan akan diproses ketika Aggregator kembali online. Hal ini sesuai dengan prinsip loose coupling dalam sistem terdistribusi.

5.2 T2: Arsitektur Publish-Subscribe (Bab 2)

Arsitektur Pub-Sub dipilih karena menyediakan tiga bentuk decoupling (Coulouris et al., 2012; Saafan, 2024):

Space decoupling: Publisher tidak perlu mengetahui alamat IP atau hostname Aggregator secara langsung. Komunikasi dilakukan melalui Redis broker dengan nama topic sebagai identifier. Ini memungkinkan penambahan atau penggantian consumer tanpa mengubah publisher.

Time decoupling: Publisher dapat mengirim pesan kapan saja, tidak perlu menunggu Aggregator online. Pesan tersimpan di Redis queue dan akan diproses ketika con-

sumer tersedia. Fitur ini penting untuk toleransi kegagalan sementara.

Synchronization decoupling: Publisher tidak memblokir menunggu respons pemrosesan dari Aggregator. Setelah pesan berhasil dikirim ke broker, publisher dapat melanjutkan pekerjaan lain. Model asinkron ini meningkatkan throughput sistem secara keseluruhan.

Dibandingkan arsitektur client-server tradisional yang memerlukan koneksi langsung dan sinkron, Pub-Sub lebih scalable untuk skenario dengan banyak producer dan consumer.

5.3 T3: Delivery Semantics (Bab 3)

Sistem mengimplementasikan semantik **at-least-once delivery** yang dikombinasikan dengan pola **idempotent consumer** untuk mencapai efek **exactly-once processing** (Steen & Tanenbaum, 2023).

At-least-once dipilih karena implementasinya lebih sederhana dibandingkan exactly-once delivery yang memerlukan protokol two-phase commit dengan overhead signifikan. Dengan at-least-once, sistem menjamin setiap pesan pasti terkirim minimal sekali, meski mungkin lebih karena retry pada network failure.

Potensi duplikasi diatasi dengan idempotent consumer: setiap event memiliki identifier unik (topic, event_id), dan database menolak insert duplikat melalui unique constraint. Dengan demikian, meski pesan diterima berkali-kali, efek akhirnya sama seolah diproses sekali saja.

Pendekatan ini memberikan trade-off yang baik antara reliability (pesan tidak hilang) dan simplicity (tidak perlu distributed transaction).

5.4 T4: Skema Penamaan (Bab 4)

Identifikasi event menggunakan kombinasi (topic, event_id) dengan karakteristik (Steen & Tanenbaum, 2023):

Event ID menggunakan UUID v4 (128-bit random) yang menjamin keunikan global tanpa koordinasi pusat. Probabilitas collision sangat rendah (2^122 kemungkinan), sehingga aman untuk generate secara independen di setiap publisher.

Topic menggunakan format hierarkis dengan dot notation (contoh: auth.login, payment.success, order.created). Struktur hierarkis memungkinkan:

- Pengelolaan namespace yang terorganisir
- Filtering berdasarkan prefix (misal: semua topic auth.*)
- Pemisahan logical domain dalam satu sistem

Kombinasi (topic, event_id) sebagai identifier memungkinkan event_id yang sama digunakan di topic berbeda, memberikan fleksibilitas tanpa mengorbankan uniqueness.

5.5 T5: Ordering (Bab 5)

Sistem tidak menjamin **total ordering** (urutan global yang konsisten di semua consumer) karena trade-off dengan throughput (Coulouris et al., 2012). Implementasi total ordering memerlukan koordinasi antar node (contoh: Lamport logical clock, vector clock) yang menambah latency.

Sebagai gantinya, sistem menggunakan **partial ordering** berbasis timestamp yang dibawa setiap event. Ordering hanya dijamin dalam scope satu producer untuk satu topic.

Untuk use case log aggregation, pendekatan ini memadai karena:

- Kelengkapan data lebih penting dari urutan absolut
- Log biasanya dianalisis dalam window waktu, bukan urutan strict
- Consumer dapat melakukan sorting berdasarkan timestamp jika diperlukan

Batasan: jika dua event dari producer berbeda memiliki timestamp identik, urutan relatifnya tidak ditentukan.

5.6 T6: Fault Tolerance (Bab 6)

Toleransi kegagalan dicapai melalui beberapa mekanisme (Coulouris et al., 2012; Steen & Tanenbaum, 2023):

Message buffering: Redis menyimpan pesan sementara, sehingga jika Aggregator crash, pesan tidak hilang dan dapat diproses setelah recovery.

Persistent storage: PostgreSQL menyimpan data ke disk yang di-mount sebagai Docker named volume. Data aman meski container dihapus dan dibuat ulang.

Crash recovery: Setelah restart, Aggregator melanjutkan pemrosesan dari queue. Mekanisme deduplikasi mencegah reprocessing event yang sudah tersimpan sebelum crash.

Retry dengan backoff: Publisher mengimplementasikan exponential backoff saat gagal mengirim ke broker, mencegah thundering herd saat recovery.

Sistem tidak menangani Byzantine failure (komponen yang berperilaku tidak terduga/jahat), hanya crash failure (komponen berhenti total).

5.7 T7: Konsistensi (Bab 7)

Sistem mencapai model konsistensi berbeda untuk operasi berbeda (Steen & Tanenbaum, 2023):

Strong consistency untuk penyimpanan event: setiap write ke PostgreSQL langsung visible untuk read berikutnya. Unique constraint menjamin tidak ada duplikat.

Eventual consistency untuk statistik: update statistik dilakukan dalam transaksi yang sama dengan insert event, namun pembacaan statistik oleh endpoint /stats mungkin melihat nilai yang sedang di-update oleh transaksi lain (READ COMMITTED). Ini acceptable karena statistik bersifat informasional.

Penggunaan single PostgreSQL node sebagai source of truth menghindari kompleksitas replikasi dan distributed consensus (Paxos/Raft) yang diperlukan untuk strong consistency pada multiple nodes.

5.8 T8: Desain Transaksi (Bab 8)

Properti ACID diterapkan sepenuhnya (Coulouris et al., 2012):

Atomicity: Pemrosesan batch event dalam satu transaksi database. Jika insert salah satu event gagal (bukan karena duplicate), seluruh batch di-rollback. Tidak ada state parsial yang tersimpan.

Consistency: Unique constraint (topic, event_id) menjaga database invariant bahwa tidak boleh ada dua event dengan identifier sama. Constraint CHECK (id = 1) pada tabel stats menjamin hanya ada satu row statistik.

Isolation: Transaksi konkuren tidak saling melihat perubahan yang belum di-commit. Dengan READ COMMITTED, setiap statement dalam transaksi melihat snapshot data yang sudah committed sebelum statement tersebut dimulai.

Durability: PostgreSQL menulis ke WAL (Write-Ahead Log) sebelum mengkonfirmasi commit, menjamin data tidak hilang meski terjadi crash setelah commit.

Trade-off: overhead transaksi menurunkan throughput dibanding operasi non-transactional, namun memberikan jaminan konsistensi yang diperlukan untuk sistem produksi.

5.9 T9: Kontrol Konkurensi (Bab 9)

Sistem menggunakan **optimistic concurrency control** melalui unique constraint dan INSERT ON CONFLICT (Coulouris et al., 2012):

```
INSERT INTO processed_events (...)
VALUES (...)
ON CONFLICT (topic, event_id) DO NOTHING
```

Pendekatan ini berbeda dari pessimistic locking yang melakukan SELECT FOR UPDATE terlebih dahulu:

- **Tidak ada lock eksplisit**: operasi insert langsung mencoba, dan ditolak jika constraint violated
- **Menghindari deadlock**: tidak ada lock untuk ditunggu
- **Throughput tinggi**: tidak ada blocking antar worker

Untuk update statistik, digunakan atomic increment:

```
UPDATE stats SET received = received + $1, unique_processed =
unique_processed + $2
```

Pattern ini mencegah lost-update tanpa memerlukan row-level locking, karena PostgreSQL menjamin atomicity operasi aritmatika dalam satu statement.

5.10 T10: Orkestrasi dan Keamanan (Bab 10-13)

Orkestrasi menggunakan Docker Compose (Khannedy, 2023a) dengan fitur:

Dependency management: depends_on memastikan PostgreSQL dan Redis running sebelum Aggregator start.

Internal network: Semua service berjalan di network internal Compose. Hanya port 8080 Aggregator yang diexpose untuk akses eksternal. PostgreSQL dan Redis tidak memiliki port mapping ke host, mengurangi attack surface.

Named volumes: pg_data dan broker_data menyimpan data persisten. Volume tidak dihapus saat docker compose down, hanya saat eksplisit docker compose down -v.

Health check: Endpoint /health digunakan untuk liveness probe, memungkinkan orchestrator mendeteksi service yang unhealthy dan melakukan restart otomatis.

Observability: Endpoint /stats menyediakan metrik runtime (event received, processed, duplicates, uptime). Logging ke stdout memungkinkan agregasi log dengan docker compose logs.

6 Kesimpulan

Sistem Pub-Sub Log Aggregator berhasil diimplementasikan dengan memenuhi seluruh kriteria:

- Idempotency dan deduplikasi berfungsi dengan benar (7.000 duplikat terdeteksi dari 20.000 event)
- Kontrol konkurensi mencegah race condition pada 10 worker paralel
- Persistensi data terjamin melalui Docker named volumes
- Throughput memenuhi target performa (≥20.000 event dengan ≥30% duplikasi)

Implementasi ini mendemonstrasikan penerapan konsep sistem terdistribusi dari Bab 1 hingga 13, dengan penekanan pada transaksi dan kontrol konkurensi (Bab 8-9) melalui penggunaan ACID properties dan optimistic concurrency control.

7 Lampiran

7.1 Link Video Demo

YouTube - Video Demo Pub-Sub Log Aggregator

(Link akan diperbarui setelah video di-upload)

Ucapan Terima Kasih

Terima kasih kepada dosen pengampu mata kuliah Sistem Terdistribusi di Program Studi Informatika, Institut Teknologi Kalimantan.

Daftar Pustaka

Ahmed, S. (2025,). *Idempotency in Distributed System*. https://sameerahmed56.medium.com/idempotency-in-a-distributed-system-df67fbd93b49

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

Khannedy, E. K. (2023a,). *Docker Compose*. https://docs.google.com/presentation/d/1xBHSvVuOA4boC5OBlzIl5VACpahpUzZ9cKhbXqLQKdE

Khannedy, E. K. (2023b,). *Redis Dasar*. https://docs.google.com/presentation/d/1kDwmRom2R7JioqkUh6mT1ohjy0t1kRQQHR1VwWgT-b0

Kumar, P. (2023). Message Deduplication Strategies in Event-Driven Architecture. *Medium - Software Engineering*. https://medium.com/

Saafan, A. (2024). Design Pattern: Publisher-Subscriber. *Nilebits Blog*. https://www.nilebits.com/blog/2024/07/design-pattern-publisher-subscriber/

Sebastián Ramírez. (2024,). *FastAPI Documentation*. https://fastapi.tiangolo.com/

Steen, M. van, & Tanenbaum, A. S. (2023). *Distributed Systems* (4th ed.). Maarten van Steen.