LAPORAN TEORI - UAS SISTEM TERDISTRIBUSI

Nama: Zaky Dio Akbar Pangestu

NIM: 11221050

Mata Kuliah: Sistem Terdistribusi

Tema: Pub-Sub Log Aggregator dengan Idempotent Consumer dan Kontrol Konkurensi

---

Daftar Isi

T1: Karakteristik Sistem Terdistribusi dan Trade-offs {#t1}

Karakteristik Utama Sistem Terdistribusi

Sistem terdistribusi didefinisikan sebagai kumpulan komputer jaringan di mana proses dan sumber daya tersebar pada banyak mesin; oleh karena itu perancang harus menolak asumsi-asumsi yang sering keliru. jaringan selalu andal, laten nol, atau bandwidth tak terbatas karena asumsi tersebut memicu desain yang rapuh (lihat kutipan, p.53–54). Untuk log aggregator berbasis publish-subscribe, sifat utama Pub-Sub adalah referential dan temporal decoupling: publisher tidak perlu mengetahui subscriber dan komunikasi bisa asinkron (p.72). Trade-off yang muncul konkret: jika sistem memperbolehkan expressive subscriptions (content-based matching), biaya matching meningkat sehingga skalabilitas menurun; solusinya sering kali membatasi ekspresivitas ke model topic-based atau menempatkan brokers/overlay routing untuk membagi beban (p.309–310). Selain itu ada kompromi antara

reliability vs. throughput dan ordering vs. latency: menjamin total order atau strong consistency menaikkan latensi dan menurunkan throughput, sedangkan memilih eventual/loose guarantees meningkatkan skalabilitas tetapi menuntut mekanisme seperti deduplication, idempotency, dan penyimpanan sementara di subscriber/broker. Desain aggregator harus menimbang kebutuhan SLA (consistency, latency, throughput) terhadap biaya operasi dan kompleksitas routing. (Tanenbaum, A. S., & Van Steen, M. (2023). Distributed Systems (4th ed.). Pearson.)

T2: Perbandingan Arsitektur Client-Server vs Publish-Subscribe {#t2}

Arsitektur client-server ditandai oleh request–reply: klien tahu server yang menyediakan layanan, mengirim permintaan, lalu menunggu jawaban (p.79). Model ini cocok bila komunikasi bersifat sinkron, stateful, atau ketika operasi memerlukan kontrol dan konsistensi kuat (mis. transaksi database). Sebaliknya, publish-subscribe menganut referential decoupling: publisher tidak mengenal subscriber; ia hanya mem-publish notifikasi, dan subscriber mendaftar interest berdasarkan tipe/event (p.69). Keunggulan teknis Pub-Sub untuk log aggregator adalah loose coupling, dukungan untuk asynchronous delivery, dan kemudahan multicast ke banyak konsumen membuatnya lebih mudah di-scale out saat banyak sumber log. Namun, buku juga menekankan keterbatasan implementasi: jika matching (filtering) kompleks, skalabilitas menjadi masalah; solusi skala praktis seringkali mengandalkan topic-based filtering dan pembagian deterministik pekerjaan ke broker (p.308). Oleh karena itu pilih Pub-Sub ketika Anda butuh high fan-out, asynchrony, dan komponen yang dapat bergabung/keluar dinamis; tetap pertimbangkan beban pada broker/matching (batasi ekspresivitas filter atau gunakan hashing/rendezvous nodes) untuk menjaga throughput dan latency sesuai SLA. (Tanenbaum, A. S., & Van Steen, M. (2023). Distributed Systems (4th ed.). Pearson.)

T3: Delivery Semantics dan Idempotent Consumer {#t3}

Tanenbaum menjelaskan tiga semantik pengiriman: at-least-once (kirim ulang sampai ada jawaban — bisa diterima berkali-kali), at-most-once (tidak akan diterima lebih dari sekali

tetapi bisa hilang), dan idealnya exactly-once meskipun secara umum sulit diwujudkan (p.512). Untuk menghadapi retry/retransmission, buku menegaskan pentingnya menjadikan operasi bersifat idempotent yang artinya menjalankan operasi berulang tetap menghasilkan efek yang sama seperti sekali saja (p.513). Praktik umum untuk mendukung deteksi duplikat (sehingga mendekati exactly-once) adalah memasukkan metadata pada header pesan, mis. sequence number atau identifier lain sehingga penerima bisa mengenali dan membuang pesan yang sudah diproses (Note 8.12, p.542–543). Dalam konteks log aggregator Pub-Sub, kombinasi dua hal ini mendesain consumer agar idempotent dan memberi pesan identifier/sequence adalah strategi praktis: idempotency mencegah efek ganda saat retry; header/sequence memungkinkan deduplikasi stateful pada consumer atau dedup store. Secara operasional, ini berarti: (1) pesan diberi ID/seq di producer, (2) consumer menyimpan history (atau menggunakan bounded dedup store) untuk menolak duplikat, dan (3) operasi agregasi ditulis supaya aman di-replay tanpa menggandakan hasil. (Tanenbaum, A. S., & Van Steen, M. (2023). Distributed Systems (4th ed.). Pearson.)

T4: Skema Penamaan untuk Topic dan Event ID {#t4}

Berdasar kutipan Tanenbaum, A. S. (2023) (Bab 6, hlmn. 326–329), menegaskan perbedaan antara name, address, dan identifier. Untuk desain skema penamaan topic dan event_id pada Pub-Sub log aggregator, prinsip-prinsip itu penting: topic dapat berfungsi sebagai human-readable, structured name (mis. logs/serviceA/error), sedangkan event_id harus berupa true identifier memenuhi ketiga properti: menunjuk pada paling banyak satu entitas, setiap entitas punya paling banyak satu identifier, dan identifier tidak dipakai ulang. Praktisnya, event_id idealnya berupa UUID v4 atau nama yang disertai content hash (mis. SHA-256(payload) atau SHA-256(payload || producer_id || timestamp)) sehingga bersifat collision-resistant dan stabil. Memisahkan name (topic) dan identifier (event_id) memungkinkan routing/filtrasi berdasarkan topic tanpa mengorbankan kemampuan deduplikasi berdasarkan identifier unik. Karena identifier “never reused”, consumer atau dedup store dapat menyimpan history event_id untuk deteksi duplikat dengan aman; ini memperkecil false positives pada deduplication dan memudahkan implementasi idempotent processing (lihat juga mekanisme header/sequence untuk retransmission di buku). Jika ingin membatasi ukuran history, gunakan TTL atau bounded cache dengan strategi persisten (mis. durable dedup store + bloom filter + occasional compaction) agar dedup tetap efektif tanpa

pertumbuhan state tak terkendali. (Tanenbaum, A. S., & Van Steen, M. (2023). Distributed Systems (4th ed.). Pearson.)

T5: Event Ordering dan Timestamp {#t5}

Dari kutipan Tanenbaum, A. S. (2023) bahwa total ordering hanya mutlak diperlukan ketika replikasi state-machine menuntut semua replika menjalankan operasi persis dalam urutan yang sama (p.265). Namun untuk banyak aplikasi (termasuk log aggregator), peristiwa yang bersifat concurrent atau tidak kausal tidak perlu diurutkan secara global boleh berbeda urutan antar-node (p.261, p.271). Pendekatan praktis yang sering dipakai adalah kombinasi Lamport logical clock (lokal monotonic counter + timestamp pesan) untuk memproduksi urutan total yang respect causality relatif terhadap pengirim, atau vector clocks bila kita perlu mendeteksi kausalitas vs concurrency secara eksplisit (pp.262, 269–270). Implementasi sederhana yang Anda usulkan — event timestamp + monotonic counter per-producer — cocok sebagai solusi ringan: setiap producer menambahkan timestamp (bisa Lamport counter atau physical time) dan counter monoton lokal untuk ordering lokal; aggregator dapat mengurutkan per-producer lalu menerapkan merge heuristik antar-producer. Batasannya: (1) clock skew / lack of global time membuat per-producer timestamps tidak mewakili urutan global nyata; (2) Lamport clocks menjamin order yang respect causality tetapi tidak membedakan concurrency (membutuhkan tie-breaker seperti producer id); (3) vector clocks menangani kausalitas tapi mahal pada memori/metadata untuk banyak producer. Jadi trade-off praktis: gunakan monotonic counter/Lamport untuk throughput rendah-latency dengan akurasi ordering lokal, dan pakai vector clocks atau TrueTime-like layanan hanya bila analisis lintas-sumber memerlukan jaminan kausal penuh. (Tanenbaum, A. S., & Van Steen, M. (2023). Distributed Systems (4th ed.). Pearson.)

T6: Failure Modes dan Strategi Mitigasi {#t6}

Failure Modes dalam Distributed Log Aggregator

Dari kutipan Tanenbaum, A. S. (2023) menjabarkan failure modes utama: crash (proses berhenti), message loss / omission (paket hilang) dan duplicate (mis. akibat retransmission atau reinjection) serta network partition (p.543). Untuk mitigasi, Tanenbaum menggambarkan praktik-praktik yang relevan: (1) retransmission on timeout—klien/penyampai mengirim ulang bila tidak menerima reply, tetapi ini dapat menyebabkan duplikasi sehingga perlu mekanisme deteksi duplikat di penerima (p.513); (2) idempotent operations—menulis konsumen agar operasi yang diulang tidak mengubah hasil (p.513); (3) menambahkan metadata/header pada pesan—mis. sender/receiver id, sequence number, delivery number—sehingga penerima dapat mengenali dan menolak duplikat (Note 8.12, pp.542–543); (4) stable messages / durable storage—menandai pesan sebagai stabil setelah ditulis ke penyimpanan andal sehingga dapat dipakai untuk recovery/replay (Note 8.12, p.542); (5) untuk komunikasi grup, gunakan sequence numbers + history buffer + negative ACKs agar penerima dapat meminta retransmisi jika terdeteksi missing (pp.518–519). Implementasi praktis pada Pub-Sub log aggregator berarti: producer menyertakan sequence/id pada header; broker/consumer menyimpan history (durable dedup store atau bounded persisted cache) untuk mengenali event_id duplikat; gunakan retry dengan backoff adaptif (buku menjelaskan retransmit dan feedback suppression untuk skala) serta persistenkan pesan sebelum dianggap “stable” agar recovery tidak menghasilkan orphan/inkonsistensi. Semua langkah ini bersama-sama mengurangi duplikasi, memperbaiki urutan relatif, dan memungkinkan recovery setelah crash tanpa kehilangan atau penggandaan data. (Tanenbaum, A. S., & Van Steen, M. (2023). Distributed Systems (4th ed.). Pearson.)

T7: Eventual Consistency pada Aggregator {#t7}

Definisi Eventual Consistency

Menurut kutipan Tanenbaum, A. S. (2023) (Bab 7), eventual consistency berarti semua replika akan bertahap konvergen ke nilai yang sama jika tidak ada pembaruan baru untuk jangka waktu cukup lama; inti model ini adalah propagasi update ke semua replika (p.407–408). Untuk sebuah log aggregator Pub-Sub yang bekerja secara asinkron dan toleran

latensi, eventual consistency sering dipilih karena memberikan high availability sambil menerima inkonsistensi sementara pada replika/konsumer berbeda.

Idempotency dan deduplikasi membantu mencapai konsistensi akhir dengan dua cara saling melengkapi. Pertama, apabila operasi bersifat idempotent (kutipan: definisi idempotent, p.513), maka pemrosesan ulang pesan akibat retry atau replay tidak mengubah keadaan akhir sehingga replay tidak merusak agregat. Kedua, mekanisme deduplication (contoh praktis: menyimpan event_id, sequence number, atau metadata) mencegah pemrosesan berulang terhadap pesan yang benar-benar sama; buku juga menyorot pendekatan CRDT/keep-all-updates yang “keep all updates while avoiding clear duplicates” untuk menghindari kehilangan update sambil mengeliminasi duplikat (p.410). Dengan kombinasi ini pesan diberi identifier unik + consumer idempotent + mekanisme rekonsiliasi (mis. merge based on vector timestamps/CRDT semantics bila perlu) aggregator dapat menerima pesan secara asinkron, memproses ulang bila perlu, dan pada akhirnya semua replika/konsumer akan menyatu pada keadaan yang konsisten sesuai model eventual consistency. (Tanenbaum, A. S., & Van Steen, M. (2023). Distributed Systems (4th ed.). Pearson.)

T8: Desain transaksi: ACID, isolation level, dan strategi menghindari lost-update.{#t8}

Dari kutipan

Dari kutipan Tanenbaum, A. S. (2023) , terlihat: transaksi didefinisikan sebagai rentetan operasi dengan sifat **all-or-nothing**; empat sifat ACID (Atomic, Consistent, Isolated, Durable) dijelaskan secara eksplisit dalam teks. _Isolation_ direlasikan ke konsep **serializability** — yaitu tujuan bahwa jadwal eksekusi konkuren harus menghasilkan efek yang setara dengan suatu eksekusi serial. _Lost update_ dijelaskan sebagai fenomena di mana dua transaksi membaca item yang sama lalu menulis kembali sehingga salah satu tulisan tertimpa. Buku menyebut bahwa **protokol penguncian**, khususnya **two-phase locking (2PL)**, adalah pendekatan standar untuk mencegah lost updates dan anomali lain, karena 2PL memastikan transaksi memperoleh dan menahan kunci sehingga serializability dapat dijaga.

Untuk implementasi praktis di _Pub-Sub log aggregator_: gunakan transactional/atomic batch append atau mekanisme version check (compare-and-swap / optimistic concurrency) saat menulis state/metadata; atau pakai locking/coordination (mis. lease/leader, ZooKeeper style) bila perlu serializability kuat. Jika ingin throughput tinggi dengan toleransi inkonsistensi sementara, gunakan level isolasi lemah + idempotency + deduplication, tapi untuk operasi kritikal gunakan 2PL atau distributed commit agar menghindari lost update.

T9

Dari kutipan Tanenbaum, A. S. (2023) , Untuk kontrol konkurensi pada aggregator, buku menekankan penggunaan **locking** (entry consistency) untuk serialisasi akses ke data bersama — setiap item dapat diasosiasikan ke satu lock sehingga operasi baca/tulis yang memerlukan konsistensi dilegalisasi lewat L(x)/U(x). Alternatif/pendukung praktis adalah **version numbers / optimistic check** (seperti contoh ZooKeeper): sebelum menulis, penulis menyertakan versi yang diharapkan; jika versi tak cocok, operasi gagal dan klien perlu retry, mencegah _lost-update_. Untuk menangani retransmission dan penggandaan akibat retry, buku menyarankan membuat operasi **idempotent** bila mungkin; bila tidak, gunakan **client-assigned sequence numbers** atau unique request IDs sehingga server dapat mengenali dan menolak retransmisi yang sama. Mekanisme ini juga dipakai antar-replica: setiap replika memberi identifier unik pada invocation agar hanya satu salinan yang diteruskan. Secara praktik: gabungkan locking untuk operasi kritis (strong correctness), version checks untuk optimis concurrency (minimal blocking), dan idempotent/sequence-ID untuk ketahanan terhadap retry. Trade-off utamanya adalah overhead koordinasi (latency/throughput) vs. tingkat konflik/keamanan integritas (consistency).

T10

Dari kutipan di atas terlihat arah teknis yang jelas untuk desain sistem terdistribusi: **orkestrasi** (mis. Docker Compose/Kubernetes) bukan sekadar menjalankan container, melainkan _resource allocation_—menjamin CPU, storage, memory, dan networking tersedia untuk tiap layanan agar SLA terpenuhi. Untuk **keamanan jaringan lokal**, buku

menyarankan pendekatan jaringan pribadi terenkripsi (VPN) dan enkripsi transport (TLS) untuk komunikasi antar-host; pada arsitektur Compose, ini berarti pastikan network overlay dan service mesh mendukung TLS/mutual-TLS atau VPN antar-node.

Untuk **persistensi**, konsep _stable message_ dan _checkpointing_ penting: data harus ditulis ke storage yang andal (volume persistent, replicated logs, atau stable storage) sebelum dianggap final; checkpointing terdistribusi membantu recovery namun mahal — jadi pilih trade-off antara frekuensi checkpoint dan waktu recovery.

Observability mencakup monitoring, logging, dan metrik (latency, bandwidth, spatial metrics). Implementasi praktis: kumpulkan latency/throughput/IO metrics, simpan log audit (untuk security) dan gunakan traces untuk debugging distribusi. Gabungkan semua ini di Compose dengan volume persistensi untuk log/metrics, network policy/TLS untuk keamanan, dan sidecar atau agent monitoring (prometheus/otel) untuk observability.

I8. Desain Transaksi: ACID, Isolation Level, dan Strategi Menghindari Lost-Update (Bab 8)

ACID Properties Implementation (Tanenbaum & Van Steen, 2017):

1. Atomicity (All-or-Nothing):

```python
await self.db.execute("BEGIN IMMEDIATE")
```

try:

    cursor = await self.db.execute("INSERT OR IGNORE INTO processed_events ...")

    await self.db.commit() # All succeed

except:

    await self.db.rollback() # All fail

```

Implementasi: Explicit transaction boundary untuk operasi `mark_processed()`. Jika ada error, rollback memastikan tidak ada partial state.

**2. Consistency (Invariants Maintained)**:

- Invariant: `received = unique_processed + duplicate_dropped`

- Enforcement: Database unique constraint `UNIQUE(topic, event_id)` menjamin tidak ada duplikat

- Verification: Testing membuktikan formula ini selalu true (20,000 = 14,091 + 5,909)

**3. Isolation (Concurrent Transactions)**:

- **Level Chosen**: READ_COMMITTED

- **Alasan**:

  - Dirty reads tidak mungkin (tidak baca uncommitted data)

  - Phantom reads acceptable (tidak perlu SERIALIZABLE karena query tidak bergantung pada row count)

  - Better performance daripada SERIALIZABLE

- **Mechanism**: `BEGIN IMMEDIATE` untuk write transactions, blocking concurrent writes
```

4. Durability (Persist After Commit):

- SQLite WAL mode: `PRAGMA journal_mode=WAL`

- Sync mode: `PRAGMA synchronous=NORMAL` (balance performance vs crash safety)

- Named volume: Data survive container destroy

Strategi Menghindari Lost-Update:

1. Atomic Increment for Counters:

```sql

UPDATE stats SET unique_processed = unique_processed + 1 WHERE id = 1

```

Bukan: `read -> increment -> write` (vulnerable to lost update)

Tapi: Atomic `UPDATE ... SET column = column + 1` (database handles concurrency)

2. Unique Constraint for Idempotency:

```sql

CREATE UNIQUE INDEX idx_topic_event_id ON processed_events(topic, event_id)

```

First write wins automatically tanpa explicit locking

3. Optimistic Concurrency Control:

- `INSERT OR IGNORE` adalah optimistic approach

- Assume no conflict, let database detect and handle collision

- No explicit locking needed (lock-free for read operations)

4. Lock for Stats Update:

```python
async with self._lock:  # Python asyncio.Lock
    await self.db.execute("UPDATE stats ...")
```

Mencegah concurrent updates ke stats yang bisa cause lost update

Referensi: Tanenbaum, A. S., & Van Steen, M. (2017). *Distributed Systems: Principles and Paradigms* (3rd ed.). Pearson Education.

T9. Kontrol Konkurensi: Locking/Unique Constraints/Upsert dan Idempotent Write Pattern (Bab 9)

Concurrency Control Mechanisms (Tanenbaum & Van Steen, 2017):

1. Database-Level Unique Constraint (Primary Mechanism):

```sql
CREATE TABLE processed_events (
```

```
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    topic TEXT NOT NULL,
    event_id TEXT NOT NULL,
    ...,
    UNIQUE(topic, event_id)
)
```

Keuntungan:
- Enforcement di database level (tidak bisa di-bypass)
- Atomic check-and-insert (tidak ada race window)
- Works across multiple application instances
- No application-level coordination needed

2. Idempotent Upsert Pattern:

```python
cursor = await self.db.execute("""
    INSERT OR IGNORE INTO processed_events
    (topic, event_id, timestamp, source, payload, processed_at)
    VALUES (?, ?, ?, ?, ?, ?)
""", (topic, event_id, ...))

inserted = cursor.rowcount > 0  # True if new, False if duplicate
```

Karakteristik:

- **Idempotent**: Calling multiple times has same effect as calling once

- **Commutative**: Order doesn't matter (event A then B = event B then A)

- **Deterministic**: Same input always produces same output

- **Lock-free**: No explicit locks, database handles concurrency

3. Application-Level Lock for Stats:

```python
async with self._lock:  # asyncio.Lock
    await self.db.execute("UPDATE stats SET received = received + ? ...", (count,))
    await self.db.commit()
```

Alasan:

- Stats updates dari multiple sources (POST /publish, background worker)

- Need serialization untuk prevent lost updates

- Lightweight (in-process lock, not distributed lock)

4. Transaction Isolation:

- **READ_COMMITTED**: Mencegah dirty reads

- **Phantom reads OK**: Karena tidak ada range queries yang sensitive terhadap row count changes

- **Trade-off**: Performance vs strict serializability

Concurrent Processing Test Results:

```python
# Test: 10 workers trying to process same event simultaneously
async def test_concurrent_processing():
    tasks = [dedup_store.mark_processed(...same event_id...) for _ in range(10)]
    results = await asyncio.gather(*tasks)

    assert results.count(True) == 1   # Only 1 succeeds
    assert results.count(False) == 9  # 9 detect duplicate
```

Proof: Unique constraint ensures exactly one INSERT succeeds, others fail atomically.

Write Pattern Philosophy:

- **Pessimistic Locking**: Acquire lock first, then write (traditional, used for stats)
- **Optimistic Concurrency**: Try write, handle conflict if occurs (INSERT OR IGNORE)
- **Choice**: Optimistic untuk event processing (high concurrency, low conflict rate on unique events)

Referensi: Tanenbaum, A. S., & Van Steen, M. (2017). *Distributed Systems: Principles and Paradigms* (3rd ed.). Pearson Education.

T10. Orkestrasi Compose, Keamanan Jaringan Lokal, Persistensi, dan Observability (Bab 10-13)

1. Orkestrasi Docker Compose (Bab 12-13):

Architecture:

```yaml
services:
  aggregator:
    depends_on:
      - condition: service_healthy  # Wait for health check
    healthcheck:
      test: ["CMD", "python", "-c", "import requests; ..."]
      interval: 10s
      timeout: 5s
      retries: 3
```

Coordination Patterns:

- **Dependency Management**: Publisher waits for aggregator to be healthy
- **Service Discovery**: Services use DNS names (`http://aggregator:8080`)
- **Graceful Startup**: Health checks prevent premature connections
- **Restart Policy**: `restart: unless-stopped` for resilience

2. Keamanan Jaringan Lokal (Bab 10):

Network Isolation:

```yaml
networks:
  uas-network:
    driver: bridge  # Isolated internal network
    internal: false  # Allow outbound for testing, set true for production
```

Security Measures:

- **No External Dependencies**: Semua komunikasi internal (aggregator ↔ publisher ↔ database)
- **No Public Exposure**: Hanya port 8080 exposed untuk testing lokal
- **Non-root User**: Containers run as `appuser` (UID 1000)
- **Minimal Attack Surface**: Alpine/slim base images

Production Enhancements (recommended):

- TLS/mTLS untuk inter-service communication
- Network policies untuk strict segmentation
- Secret management (Docker secrets atau external vault)

3. Persistensi dengan Volumes (Bab 11):

Named Volume Strategy:

```yaml
volumes:
  aggregator_data:
    driver: local
```

Data Persistence Guarantees:

- **Location**: `/var/lib/aggregator/dedup.db` di dalam container
- **Mapping**: Named volume `aggregator_data` mounted ke path tersebut
- **Durability**: Data survive `docker compose down` (tanpa `-v` flag)
- **Portability**: Volume bisa di-backup dan restore

Testing Persistence:

```bash
# Test 1: Send events
curl -X POST http://localhost:8080/publish ...

# Test 2: Destroy container
docker compose down
```

Test 3: Recreate container

docker compose up -d

Test 4: Verify data still exists

curl http://localhost:8080/stats # Stats unchanged ✓

```

**4. Observability** (Bab 13):

**Metrics Endpoint**:

```json

GET /stats

{

  "received": 20000,

  "unique_processed": 14091,

  "duplicate_dropped": 5909,

  "topics": ["alerts", "events", "logs", "metrics", "traces"],

  "uptime_seconds": 152,

  "queue_size": 0

}

```

Structured Logging:
```

```python
logger.info("EVENT PROCESSED - topic: {}, event_id: {}, source: {}")
logger.warning("DUPLICATE DROPPED - topic: {}, event_id: {}")
logger.error("Error in event consumer: {}", exc_info=True)
```

Log Levels:
- INFO: Normal operations (event processed, batch sent)
- WARNING: Duplicates detected, queue full
- ERROR: Exceptions, transaction failures

Observability Features:
- **Health Endpoint**: `/health` untuk readiness/liveness checks
- **Metrics Export**: Stats dalam JSON format untuk integration dengan monitoring tools
- **Audit Trail**: Setiap event punya `processed_at` timestamp
- **Queue Monitoring**: `queue_size` di stats untuk detect backpressure

Future Enhancements:
- Prometheus metrics export
- Distributed tracing (OpenTelemetry)
- Centralized logging (ELK stack)
- Alerting rules (on error rate, queue size)

Referensi: Tanenbaum, A. S., & Van Steen, M. (2017). *Distributed Systems: Principles and Paradigms* (3rd ed.). Pearson Education.

REFERENSI

Tanenbaum, A. S., & Van Steen, M. (2017). *Distributed Systems: Principles and Paradigms* (3rd ed.). Pearson Education.
