SISTEM PARALEL DAN TERDISTRIBUSI

Nama : Naufal Andrian
NIM : 11221075
KELAS : A

1. Karakteristik Sistem Terdistribusi dan Trade-off Desain Pub-Sub Aggregator

Sistem terdistribusi Pub-Sub Log Aggregator menunjukkan lima karakteristik utama.

1. *autonomy and heterogeneity* ditunjukkan melalui arsitektur multi-layanan yang terdiri atas komponen aggregator, publisher, dan basis data yang beroperasi secara independen namun tetap terkoordinasi.
2. karakteristik *concurrency* terlihat dari kemampuan sistem dalam menangani banyak publisher yang mengirimkan event secara bersamaan, sementara consumer workers memproses data tersebut secara paralel.
3. *failure independence* tercermin dari kemampuan sistem untuk tetap beroperasi meskipun terjadi kegagalan pada salah satu container, karena mekanisme pemulihan didukung oleh penyimpanan deduplikasi yang bersifat persisten.
4. *partial failures* dapat terjadi akibat gangguan jaringan, seperti timeout antara publisher dan aggregator, namun kondisi ini ditangani melalui penerapan mekanisme *retry logic*.
5. *scale transparency* diwujudkan dengan perancangan sistem yang mendukung *horizontal scaling* melalui penambahan jumlah publisher dan instance aggregator tanpa memengaruhi fungsi utama sistem.

Arsitektur Pub-Sub menawarkan tingkat *loose coupling* yang tinggi, di mana publisher tidak memiliki ketergantungan langsung terhadap subscriber, serta mendukung skalabilitas horizontal dan mekanisme pengiriman pesan *at-least-once*. Namun, pendekatan ini memiliki konsekuensi berupa latensi yang relatif lebih tinggi. Arsitektur Pub-Sub dipilih karena sistem log aggregation membutuhkan *loose coupling* untuk menangani berbagai sumber event, toleransi terhadap mekanisme *at-least-once delivery* yang dapat diatasi melalui penerapan idempotensi, serta kemampuan skalabilitas untuk mengelola banyak publisher dan aggregator secara simultan.

2. Kapan Memilih Arsitektur Publish-Subscribe dibanding Client-Server?

Arsitektur Publish–Subscribe dipilih untuk beberapa skenario penggunaan tertentu.

1. arsitektur ini sesuai untuk pola *multiple producers to single consumer*, seperti pada sistem log aggregation dan event streaming, di mana banyak sumber data mengirimkan event ke satu atau lebih konsumen.
2. Publish–Subscribe mendukung *loose coupling*, sehingga producer tidak perlu mengetahui detail atau spesifikasi consumer yang memproses data.
3. arsitektur ini memungkinkan *horizontal scaling* dengan menambahkan publisher baru tanpa memerlukan perubahan pada sisi aggregator.
4. mekanisme pemrosesan bersifat asinkron sehingga tidak memerlukan respons langsung (*immediate response*), yang cocok untuk sistem non-real-time.
5. dukungan *content-based routing* memungkinkan event difilter berdasarkan topik atau pola tertentu, sehingga hanya data relevan yang diteruskan ke consumer.

Sebaliknya, arsitektur client–server lebih sesuai untuk skenario yang membutuhkan semantik *request–response*, seperti pemrosesan pemesanan atau eksekusi kueri. Model ini efektif untuk komunikasi *point-to-point* dengan tingkat *tight coupling* yang masih dapat diterima, serta mendukung konsistensi kuat melalui transaksi ACID. Selain itu, client–server lebih unggul dalam hal latensi rendah untuk operasi sinkron yang membutuhkan respons secara real-time.

Secara teknis, pemilihan arsitektur Publish–Subscribe untuk sistem log aggregator didasarkan pada beberapa pertimbangan utama.

1. sistem harus menangani banyak sumber data heterogen, seperti delapan layanan berbeda yang mencakup modul autentikasi, pembayaran, dan inventori, dengan platform dan karakteristik yang beragam.
2. prinsip *decoupling* memungkinkan aggregator beroperasi tanpa ketergantungan terhadap detail implementasi setiap sumber log.
3. dukungan *replay* dan *retention* pada antrian event memungkinkan data log diproses ulang untuk keperluan analisis dan audit.
4. mekanisme *content filtering* memungkinkan aggregator berlangganan pada topik tertentu, seperti logs.payment atau logs auth, sehingga pemrosesan menjadi lebih efisien.
5. kemampuan *backpressure handling* memungkinkan aggregator mengatur laju pemrosesan event sesuai kapasitasnya sendiri tanpa membebani publisher.

3. At-Least-Once vs Exactly-Once Delivery; Peran Idempotent Consumer

Mekanisme *at-least-once delivery* memungkinkan sebuah pesan diproses lebih dari satu kali, namun memiliki keunggulan berupa implementasi yang relatif sederhana. Pendekatan ini dapat diterima untuk operasi yang bersifat idempoten, di mana pemrosesan berulang tidak mengubah hasil akhir sistem. Sebaliknya,

exactly-once delivery menjamin bahwa setiap pesan diproses tepat satu kali, tetapi membutuhkan mekanisme tambahan seperti deduplikasi atau koordinasi transaksi terdistribusi yang kompleks, sehingga meningkatkan overhead sistem.

Dalam praktiknya, penerapan *idempotent consumer* memungkinkan semantik *at-least-once* ditransformasikan menjadi *effectively exactly-once*. Mekanisme ini dicapai dengan memastikan bahwa pesan yang sama tidak diproses lebih dari sekali meskipun dikirim secara duplikat, sebagaimana ilustrasi berikut:

```
# At-least-once delivery dengan potensi duplikasi
POST /publish { event_id: "evt-001", ... }
POST /publish { event_id: "evt-001", ... }  # Duplikasi
```

```
# Idempotent consumer: pesan hanya diproses satu kali
await mark_processed(event)  # berhasil
await mark_processed(event)  # ditolak (UNIQUE constraint)
```

Pola implementasi yang umum digunakan adalah pemanfaatan *deduplication store* dengan operasi *atomic check-and-insert*. Contohnya, penyimpanan deduplikasi dapat direalisasikan melalui perintah SQL INSERT INTO dedup_store (topic, event_id) VALUES ($1, $2) ON CONFLICT DO NOTHING. Pada sisi konsumen, logika pemrosesan memeriksa status is_processed(topic, event_id) sebelum menjalankan fungsi save_event(), seluruhnya dieksekusi dalam satu transaksi atomik untuk menjaga konsistensi data.

Pendekatan ini menghasilkan trade-off yang menguntungkan, di mana kombinasi *at-least-once delivery* dan *idempotent consumer* mampu memberikan semantik *effectively exactly-once* dengan tingkat kompleksitas yang lebih rendah. Strategi ini menghindari kebutuhan koordinasi transaksi terdistribusi yang rumit, sekaligus tetap menjamin konsistensi data pada sistem terdistribusi berskala besar.

4. Skema Penamaan Topic dan Event_ID (Unik, Collision-Resistant) untuk Dedup

Penamaan topik (*topic naming convention*) pada sistem Publish–Subscribe menggunakan struktur hierarkis dengan format logs.<service>.<operation>.<severity>. Contoh penerapan konvensi ini antara lain logs.authentication.login.ERROR, logs.payment.transaction.INFO, dan logs.inventory.stock_update.WARNING. Struktur hierarkis tersebut memberikan beberapa keuntungan utama, yaitu memungkinkan penggunaan *wildcard subscription* seperti logs.payment.*, memberikan makna semantik yang jelas terkait sumber dan jenis event, serta mempermudah proses penyaringan (*filtering*) dan pengelompokan log.

Perancangan *event identifier* dilakukan dengan mempertimbangkan ketahanan terhadap tabrakan (*collision resistance*) serta kemudahan integrasi dalam sistem terdistribusi. Pendekatan yang direkomendasikan adalah penggunaan UUID,

di mana event_id = str(uuid.uuid4()) menghasilkan entropi 128-bit dengan probabilitas tabrakan yang sangat kecil, yakni kurang dari 10^{-15} untuk satu miliar event, serta tidak memerlukan koordinasi antar layanan. Alternatif lain adalah skema *timestamp plus counter*, dengan format event_id = f"{timestamp_ms}-{service_id}-{local_counter}", yang bersifat bebas tabrakan untuk satu layanan dan mendukung pengurutan (*sortable*). Namun, pendekatan ini memerlukan koordinasi tambahan apabila diterapkan pada lingkungan multi-layanan (Wijaya & Santosa, 2022). Pendekatan ketiga adalah penggunaan *cryptographic hash*, misalnya hashlib.sha256(f"{timestamp}{source}{unique_data}").hexdigest()[:16], yang bersifat deterministik sehingga pengiriman ulang event akan menghasilkan *event_id* yang sama, dan oleh karena itu sesuai untuk mendukung mekanisme *idempotent retries*.

Implementasi deduplikasi event direalisasikan pada tingkat basis data dengan mendefinisikan kendala unik, seperti UNIQUE (topic, event_id) pada tabel processed_events. Kendala ini memungkinkan deteksi tabrakan dilakukan secara otomatis oleh sistem basis data. Apabila terdapat *event_id* yang sama pada topik yang sama, maka akan terjadi *unique violation exception*, sehingga pemrosesan ganda dapat dicegah secara efektif. Pendekatan ini memberikan jaminan konsistensi data tanpa menambah kompleksitas logika deduplikasi pada sisi aplikasi.

5. Ordering Praktis (Timestamp + Monotonic Counter); Batasan dan Dampaknya

Penerapan *global total ordering* pada sistem terdistribusi sulit dicapai secara konsisten karena adanya *clock skew* antar node. Dua event yang berasal dari layanan berbeda dapat memiliki cap waktu yang identik, misalnya Event1 pada t = 10:00:00.001 dan Event2 pada t = 10:00:00.001, meskipun urutan kausalnya berbeda. Ketidaksinkronan waktu ini menyebabkan pengurutan berbasis *timestamp* menjadi tidak andal. Sebagai alternatif, *causal ordering* dapat diterapkan, di mana Event1 dianggap mendahului Event2 apabila Event1 menyebabkan terjadinya Event2. Pendekatan ini umumnya diimplementasikan menggunakan Lamport clock, Vector clock, atau pelacakan dependensi pada tingkat aplikasi.

Dalam konteks sistem log aggregation, pendekatan yang lebih praktis adalah penerapan *per-topic monotonic ordering*. Setiap event dalam satu topik diberikan nomor urut (*sequence number*) yang meningkat secara monoton, terlepas dari nilai *timestamp*-nya. Sebagai contoh, pada topik logs.payment, urutan pemrosesan ditentukan berdasarkan nilai seq, bukan berdasarkan waktu kejadian:

```
Topic: logs.payment
  Event 1: seq=1, timestamp=10:00:00.100, amount=100
  Event 2: seq=2, timestamp=10:00:00.050, amount=50
  Event 3: seq=3, timestamp=10:00:01.900, amount=75
```

```
# Urutan pemrosesan: seq 1 → 2 → 3, terlepas dari timestamp
```

Implementasi pengurutan ini dapat dilakukan dengan kueri basis data seperti SELECT * FROM processed_events WHERE topic = $1 ORDER BY seq ASC, yang memastikan konsistensi urutan pemrosesan secara kausal dalam satu topik. Pendekatan ini dinilai lebih andal dibandingkan pengurutan berbasis *timestamp* semata, khususnya dalam lingkungan terdistribusi dengan ketidakselarasan waktu antar node.

Meskipun demikian, pendekatan ini memiliki beberapa batasan dan implikasi. *Clock skew* dapat menyebabkan pencatatan waktu event menjadi tidak akurat, namun dampaknya dapat diminimalkan dengan menggunakan *monotonic counter* sebagai acuan utama urutan. Pada kondisi *network partition*, event dapat tiba dalam keadaan tidak berurutan, sehingga diperlukan mekanisme *reordering* di sisi aggregator. Keberadaan multiple publisher dapat menghasilkan event yang saling terinterleaving, namun pengurutan pada tingkat topik umumnya sudah memadai untuk kebutuhan analisis log. Selain itu, pada skenario *crash recovery*, terdapat risiko kehilangan nilai *sequence counter*, sehingga diperlukan penyimpanan nilai tersebut secara persisten di basis data.

Berdasarkan pertimbangan tersebut, dapat disimpulkan bahwa *per-topic monotonic ordering* sudah mencukupi untuk sistem log aggregation. Penerapan *global total ordering* tidak memberikan manfaat yang signifikan dalam konteks ini, sementara kompleksitas koordinasi dan overhead sinkronisasi yang ditimbulkannya relatif tinggi. Oleh karena itu, desain sistem lebih dioptimalkan pada konsistensi urutan dalam satu topik dibandingkan konsistensi global lintas topik.

6. Failure Modes dan Mitigasi (Retry, Backoff, Durable Dedup Store, Crash Recovery)

Sistem Publish–Subscribe pada log aggregation dirancang untuk menangani berbagai skenario kegagalan melalui mekanisme mitigasi yang terukur. Pada kasus *publisher crash*, event disimpan dalam antrian lokal persisten dan dikirim ulang saat layanan aktif kembali, sementara deduplikasi mencegah pemrosesan ganda. *Network timeout* ditangani dengan *exponential backoff retry* hingga batas percobaan tertentu, dan karena pengiriman bersifat idempoten, proses pengulangan tetap aman.

Apabila terjadi *aggregator crash*, event yang telah diproses tetap terlindungi oleh *deduplication store* yang persisten di basis data, sehingga saat sistem pulih, event yang sama tidak diproses ulang. Kehilangan koneksi basis data ditangani melalui mekanisme *auto-reconnect* pada *connection pool* serta *retry* transaksi, dengan dukungan *health check endpoint* untuk pemantauan layanan. Pada *partial*

batch failure, sistem mengembalikan respons parsial sehingga klien hanya mengirim ulang event yang gagal.

Strategi mitigasi ini direalisasikan melalui pola implementasi berikut:

```
# Durable deduplication store
CREATE TABLE dedup_store (
   topic TEXT,
   event_id TEXT,
   UNIQUE (topic, event_id)
);

# Crash recovery pattern
async def consumer_restart(event):
   if is_processed(event):
      return  # Skip processed event
   else:
      process_event()  # Re-process if needed

# Exponential backoff
delay = (2 ** attempt) * 100  # 100ms, 200ms, 400ms, ...
```

Sebagai pendukung, sistem menyediakan endpoint *health check* (GET /health), *structured logging* untuk debugging, serta log kontainer untuk analisis pascakegagalan. Pendekatan ini memastikan ketahanan sistem tanpa menambah kompleksitas berlebih.

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

Konsistensi pada sisi aggregator bersifat *eventual consistency*. Ketika sebuah event dipublikasikan, event tersebut akan melalui tahap antrian, pemrosesan, dan penyimpanan. Pada saat permintaan statistik dilakukan secara langsung setelah publikasi, data yang sedang diproses (*in-flight*) belum sepenuhnya tercermin. Namun, setelah selang waktu tertentu, statistik akan diperbarui dan mencerminkan seluruh event yang telah diproses secara konsisten.

```
T1: Event published → Queued → Processing → Saved
T2: GET /stats (immediate) → statistik belum mencerminkan T1
T3: GET /stats (after delay) → statistik telah mencerminkan T1
```

Idempotensi dan deduplikasi berperan penting dalam menjaga konsistensi sistem. Mekanisme idempotensi memungkinkan pengiriman ulang event secara aman, di mana beberapa permintaan POST /publish dengan event_id yang sama menghasilkan efek yang identik. Deduplikasi mencegah pemrosesan ganda dengan melacak pasangan (topic, event_id). Pada sisi konsumen, status deduplikasi diperiksa sebelum pemrosesan dilakukan, sehingga event yang telah diproses akan dilewati.

Jaminan *eventual consistency* dicapai melalui kombinasi idempotensi dan deduplikasi. Event duplikat tetap diproses secara aman tanpa kehilangan pembaruan, seluruh operasi penyimpanan dan pembaruan statistik dilakukan secara transaksional, serta urutan event dalam satu topik tetap terjaga. Dalam skenario terburuk, sebuah event dapat diterima dan diproses beberapa kali akibat *network retry*, namun hanya satu entri yang akan tersimpan di basis data karena adanya kendala unik, sementara statistik tetap konsisten melalui operasi *transactional update*.

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

Implementasi properti ACID pada sistem dilakukan untuk menjamin integritas dan konsistensi data. *Atomicity* dicapai melalui mekanisme transaksi *all-or-nothing* menggunakan BEGIN dan COMMIT. *Consistency* dijaga dengan penerapan kendala basis data seperti UNIQUE dan *foreign key* untuk mencegah kondisi data yang tidak valid. *Isolation* diterapkan pada tingkat SERIALIZABLE, sementara *Durability* dijamin melalui mekanisme *Write-Ahead Logging* (WAL) pada PostgreSQL.

Pemilihan tingkat isolasi melibatkan trade-off antara performa dan konsistensi. Meskipun SERIALIZABLE memiliki overhead tertinggi dibandingkan tingkat isolasi lain, tingkat ini dipilih karena proses deduplikasi bersifat kritikal dan tidak dapat mentoleransi *phantom reads*. Selain itu, pemrosesan event harus bersifat atomik, sehingga penurunan throughput masih dapat diterima demi konsistensi yang kuat.

Pencegahan *lost update* menjadi aspek penting dalam pemrosesan event secara paralel. Masalah ini terjadi ketika beberapa worker melakukan pembaruan data secara bersamaan tanpa mekanisme sinkronisasi, sehingga sebagian pembaruan hilang.

```
Worker 1: read stats (count=10) → write stats (count=11)
Worker 2: read stats (count=10) → write stats (count=11) ← LOST!
Expected: 12, Actual: 11
```

Solusi yang direkomendasikan adalah menggunakan *atomic increment* langsung pada basis data, yang aman terhadap akses konkuren:

```
UPDATE event_stats
SET received = received + 1,
   unique_processed = unique_processed + 1
WHERE id = 1;
```

Selain itu, deduplikasi event dilakukan secara atomik melalui kendala unik pada basis data, sehingga hanya satu event dengan kombinasi (topic, event_id) yang dapat disimpan:

```
INSERT INTO dedup_store (topic, event_id)
VALUES ($1, $2)
ON CONFLICT DO NOTHING;
```

Kombinasi strategi ini memastikan integritas data dengan mencegah *race condition*, menjaga konsistensi melalui jaminan ACID, serta meningkatkan keandalan sistem melalui deteksi konflik secara otomatis.

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

Pengendalian konkurensi pada sistem pemrosesan event dapat diterapkan melalui beberapa teknik utama. *Pessimistic locking* menggunakan penguncian eksplisit untuk mencegah konflik saat akses bersamaan, namun pendekatan ini menimbulkan *lock contention* dan berisiko menurunkan performa pada skala besar. *Optimistic locking* memanfaatkan kolom versi untuk mendeteksi konflik tanpa penguncian, sehingga lebih skalabel, tetapi memerlukan logika *retry* tambahan pada tingkat aplikasi.

Pendekatan yang direkomendasikan adalah penggunaan kendala unik (*UNIQUE constraint*), karena mekanisme ini diimplementasikan langsung oleh basis data secara atomik dan idempoten, tanpa memerlukan penguncian eksplisit maupun pengelolaan versi. Teknik ini bersifat *fail-fast*, aman untuk *retry*, dan sangat sesuai untuk lingkungan dengan tingkat konkurensi tinggi.

Implementasi inti dari pola penulisan idempoten direalisasikan sebagai berikut:

```
INSERT INTO processed_events (topic, event_id, ...)
VALUES ($1, $2, ...)
ON CONFLICT DO NOTHING;
```

Dengan pendekatan ini, event yang sama tidak akan diproses lebih dari satu kali meskipun terjadi pengiriman ulang atau eksekusi paralel. Dibandingkan teknik lain, *UNIQUE constraint* tidak menimbulkan *lock contention*, tidak berisiko *deadlock*, mudah diimplementasikan, serta memberikan skalabilitas tinggi. Oleh karena itu, mekanisme ini dipilih sebagai solusi optimal untuk menjaga konsistensi dan keandalan pemrosesan event pada sistem log aggregation terdistribusi.

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

Orkestrasi Docker Compose

Orkestrasi layanan dilakukan menggunakan Docker Compose untuk mengelola dependensi, ketersediaan layanan, dan *lifecycle* container. Layanan aggregator dikonfigurasi agar menunggu kesiapan basis data melalui *health check*, serta mendukung *automatic restart*, *service discovery*, dan persistensi data.

```
services:
```

```
  aggregator:
    depends_on:
      postgres:
        condition: service_healthy
    healthcheck:
      test: ["CMD", "curl", "-f", "http://localhost:8080/health"]
      interval: 30s
```

Keamanan Jaringan Lokal

Isolasi jaringan diterapkan menggunakan *bridge network* internal, sehingga komunikasi antarlayanan hanya terjadi di dalam jaringan privat. Basis data tidak diekspos ke luar container dan hanya dapat diakses oleh aggregator, yang meningkatkan keamanan sistem secara keseluruhan.

```
networks:
  aggregator_network:
    driver: bridge

services:
  aggregator:
    networks: [aggregator_network]
    ports: ["8080:8080"]
  postgres:
    networks: [aggregator_network]
```

Persistensi Data

Persistensi data direalisasikan melalui *named volume* untuk memastikan data tetap tersedia meskipun container dihentikan atau dihapus. Dengan pendekatan ini, proses *shutdown* dan *restart* tidak menyebabkan kehilangan data log.

```
volumes:
  postgres_data:

services:
  postgres:
    volumes:
      - postgres_data:/var/lib/postgresql/data
```

Observability

Observabilitas sistem dicapai melalui kombinasi *health check*, endpoint statistik, dan *structured logging*. Endpoint GET /health digunakan untuk pemantauan ketersediaan layanan, sedangkan GET /stats menyediakan metrik

pemrosesan event. *Structured logging* dan log kontainer mendukung proses debugging, audit, serta analisis performa secara proaktif.

Pendekatan ini memungkinkan pemantauan kondisi sistem secara menyeluruh, mempermudah deteksi kegagalan, dan meningkatkan keandalan operasional pada lingkungan terdistribusi.

REFERENSI

Ajismanto, F., Mahmud, Barovih, G., Rupilele, F. G. J., Guntoro, Octafian, D. T., Aprizal, Y., Siregar, M. T., Afnarius, S., Krisnanik, E., & Widiastiwi, Y. (2023). *Sistem terdistribusi*.

Richardo, D., & Kurniasih, T. (2024). *Implementasi Google Cloud Pub/Sub menggunakan metode subscription pull dalam pengiriman data promosi toko di PT XYZ*.

Siswanto, D., Tahitya, C. O., & Priyandoko, G. (2025). *Implementasi arsitektur publish/subscribe pada sistem hidroponik menggunakan NodeMCU V3*. Fortech, *4*(2). https://doi.org/10.56795/fortech.v4i2.4201.

Syahrul Alim, A. T., Kartikasari, D. P., & Bakhtiar, F. A. (2020). *Implementasi paradigma publish–subscribe untuk menjalankan event-based monitoring pada sistem pengamatan kandang ternak*.

Rizal, C., Supiyandi, Zen, M., & Eka, M. (2022). *Perancangan server kantor Desa Tomuan Holbung berbasis client–server*. Ekuitas: Jurnal Ekonomi dan Keuangan, *3*(2). https://doi.org/10.47065/ekuitas.v3i2.1167.

Kurniawan, A., & Wijaya, D. (2021). *Implementasi idempotent consumer pattern pada sistem message queue berbasis RabbitMQ*. Jurnal Teknologi Informatika dan Komputer, *7*(2), 89–97.

Prasetyo, H., Nugroho, E., & Santoso, P. (2022). *Analisis strategi delivery semantics pada distributed messaging system*. Jurnal Sistem Informasi Bisnis, *12*(1), 45–54.

Rahman, F., & Setiawan, B. (2023). *Penerapan deduplication strategy untuk exactly-once processing pada event-driven architecture*. Jurnal Informatika dan Komputer, *28*(2), 156–164.

Gunawan, W., & Wibowo, A. (2024). *Pemanfaatan algoritma K-means dalam klasterisasi gempa Sulawesi*. Faktor Exacta, *17*(3). https://doi.org/10.30998/faktorexacta.v17i3.23169.

Coulouris, G., Dollimore, J., Kindberg, T., & Blair, G. (2011). *Distributed systems: Concepts and design* (5th ed.). Cambridge University Press.

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