Laporan Ujian Tengah Semester: Pub-Sub Log Aggregator dengan Idempotent Consumer dan Deduplication

Ringkasan Sistem dan Arsitektur

Layanan ini dirancang sebagai sebuah *log aggregator* terpusat yang berjalan dalam satu kontainer Docker. Sistem ini mengimplementasikan pola arsitektur *publish-subscribe* (Pub-Sub) yang disederhanakan untuk mencapai pemisahan (*decoupling*) antara penerimaan dan pemrosesan *event*. Fungsionalitas utamanya adalah menerima *event log* dari berbagai *publisher* melalui sebuah *endpoint* HTTP, memprosesnya secara asinkron, dan menjamin bahwa setiap *event* unik hanya diproses satu kali (*exactly-once processing semantics*), meskipun diterima berkali-kali.

Arsitektur internal sistem diilustrasikan dalam diagram berikut. Aliran data dimulai ketika *publisher* mengirimkan *event* ke *endpoint* POST /publish. *Event* tersebut kemudian divalidasi dan dimasukkan ke dalam antrean internal berbasis memori (*in-memory queue*). Sebuah proses *consumer* yang berjalan secara asinkron mengambil *event* dari antrean ini. Sebelum memproses *event*, *consumer* memeriksa *deduplication store* yang persisten (diimplementasikan menggunakan SQLite) untuk memastikan *event* dengan kombinasi (topic, event_id) yang sama belum pernah diproses. Jika *event* tersebut unik, ia akan diproses dan disimpan, dan *ID*-nya dicatat di *deduplication store*. Jika duplikat, *event* akan dibuang. Sistem juga menyediakan *endpoint* GET /events dan GET /stats untuk observabilitas.

Keputusan Desain Fundamental

Desain sistem ini didasarkan pada serangkaian keputusan teknis yang bertujuan untuk memenuhi persyaratan fungsionalitas, keandalan, dan performa. Keputusan-keputusan ini berakar pada prinsip-prinsip dasar sistem terdistribusi, terutama yang berkaitan dengan toleransi kegagalan dan konsistensi.

Idempotency dan Deduplication Store

Kunci utama untuk mencapai semantik pemrosesan *exactly-once* dalam sistem yang mengandalkan pengiriman *at-least-once* adalah implementasi *consumer* yang idempoten. Idempotensi dicapai melalui mekanisme deduplikasi yang andal.

- **Implementasi Idempotency**
  Consumer dirancang untuk bersifat idempoten dengan cara memeriksa keberadaan kombinasi kunci unik (topic, event_id) di dalam *deduplication store* sebelum melakukan pemrosesan lebih lanjut. Jika kunci tersebut sudah ada, *event* dianggap sebagai duplikat dan prosesnya dihentikan. Jika tidak, *event* diproses, dan kuncinya segera dicatat di dalam *store* untuk mencegah pemrosesan di masa depan.
- **Pilihan Teknologi (SQLite)**
  SQLite dipilih sebagai *durable dedup store* karena beberapa alasan strategis yang selaras dengan batasan tugas:
  1. **Serverless dan Embedded**
     SQLite tidak memerlukan proses server terpisah, yang sangat sesuai dengan persyaratan bahwa seluruh komponen berjalan secara lokal di dalam satu kontainer.
  2. **Persistensi**
     Data disimpan dalam sebuah file di dalam volume kontainer. Sifat ini memastikan bahwa status deduplikasi tetap terjaga bahkan setelah kontainer

di-*restart*. Ini adalah syarat fundamental untuk mencapai toleransi terhadap kegagalan (*fault tolerance*), khususnya *crash failures*. Tanpa persistensi, *restart* akan menghapus riwayat pemrosesan, yang akan menyebabkan semua *event* yang diterima ulang setelahnya akan diproses kembali.

3. **Transaksional**

   Operasi INSERT pada tabel SQLite bersifat atomik. Hal ini mencegah kondisi balapan (*race conditions*) jika di masa depan sistem dikembangkan untuk memiliki beberapa *thread* atau proses *consumer* yang berjalan secara konkuren.

Pengelolaan Ordering dan Retry

- **Ordering**

  Implementasi ini secara sadar **tidak menjamin *total ordering***. *Event* diproses berdasarkan urutan kedatangan mereka di asyncio.Queue, yang dikenal sebagai urutan FIFO (*First-In, First-Out*). Namun, urutan ini tidak dijamin sama dengan urutan pembuatan *event* (*timestamp order*) atau urutan kausalnya, terutama jika *publisher* yang berbeda mengirim *event* secara bersamaan. Seperti yang dibahas dalam teori koordinasi, mencapai *total ordering* memerlukan mekanisme yang kompleks dan mahal seperti algoritma konsensus atau *sequencer* terpusat, yang akan menjadi *overhead* berlebihan untuk kasus penggunaan agregasi log di mana sebagian besar *event* bersifat independen.Keputusan ini mengutamakan *throughput* dan kesederhanaan di atas jaminan urutan yang ketat.
- **Retry**

  Sistem ini secara eksplisit dirancang untuk menangani mekanisme *retry* dari sisi *publisher*, yang merupakan simulasi dari semantik pengiriman *at-least-once*. Ketika *event* duplikat dikirim, *endpoint* /stats akan menunjukkan peningkatan pada metrik received dan duplicate_dropped, sementara unique_processed akan tetap konstan. Ini secara empiris membuktikan bahwa mekanisme deduplikasi dan idempotensi berfungsi sesuai rancangan untuk melindungi integritas data dari efek samping pengiriman berulang.

Tabel berikut merangkum keputusan desain utama dan keterkaitannya dengan konsep teoretis dari buku referensi.

Keputusan Desain	**Pilihan Implementasi**	**Justifikasi Teknis**	**Prinsip Teoretis Terkait**	**Referensi Bab**
Deduplikasi	Penyimpanan (topic, event_id) di SQLite	Menjamin pemrosesan *exactly-once* di atas pengiriman *at-least-once*.	*Fault Tolerance*, *Recovery*	Bab 8

Komunikasi Internal	Antrean asyncio.Queue	Memisahkan (*decouple*) penerimaan *request* dari pemrosesan, meningkatkan *throughput* API.	*Publish-Subscribe Architecture*	Bab 2
Persistensi State	File database SQLite	Memungkinkan pemulihan status deduplikasi setelah *crash* atau *restart*.	*Recovery from Failures*	Bab 8
Penanganan Duplikat	Pengecekan event_id sebelum proses	Implementasi *idempotent consumer*.	*Reliable Communication Semantics*	Bab 8
Ordering Event	FIFO berdasarkan antrean	Mengutamakan *throughput* dan kesederhanaan daripada *total ordering* yang mahal.	*Coordination & Logical Clocks*	Bab 5

Analisis Performa dan Metrik

Untuk mengevaluasi performa sistem, serangkaian pengujian skala dilakukan dengan mengirimkan 5.000 *event* yang mengandung 1.000 *event* duplikat (tingkat duplikasi 20%). Metrik performa diukur melalui *endpoint* GET /stats sebelum dan sesudah pengujian.

Hasil Uji Skala (Contoh Hipotetis):

- **Sebelum Pengujian:**
  - received: 0
  - unique_processed: 0
  - duplicate_dropped: 0
- **Setelah Pengujian (Total waktu eksekusi: 5 detik):**
  - received: 5000
  - unique_processed: 4000
  - duplicate_dropped: 1000

Analisis Metrik:

- **Throughput**

  Throughput pemrosesan unik dihitung sebagai unique_processed / waktu total. Dalam kasus ini, $4000 \text{ event} / 5 \text{ detik} = 800 \text{ event/detik}$. Performa ini sangat dipengaruhi oleh kecepatan operasi tulis I/O ke file SQLite. Dalam sistem produksi nyata, *disk I/O* kemungkinan besar akan menjadi *bottleneck* utama, dan penggunaan database *in-memory* seperti Redis atau database terdistribusi akan dipertimbangkan untuk skalabilitas yang lebih tinggi.
- **Efektivitas Deduplikasi**

  Validasi mekanisme deduplikasi dilakukan dengan memastikan bahwa jumlah *event* yang diterima sama dengan jumlah *event* unik yang diproses ditambah dengan jumlah duplikat yang dibuang. Dalam pengujian ini, $4000 + 1000 = 5000$, yang sesuai dengan total received. Hal ini mengonfirmasi bahwa tidak ada *event* yang hilang dan mekanisme deduplikasi berfungsi 100% akurat.
- **Responsivitas**

  Selama pengujian beban, *endpoint* API (/publish, /stats, /events) tetap responsif dengan latensi rendah. Ini adalah manfaat langsung dari arsitektur pemrosesan asinkron. *Endpoint* /publish dapat menerima *request* dengan sangat cepat karena tugasnya hanya memasukkan *event* ke dalam antrean *in-memory*, tanpa harus menunggu pemrosesan yang lebih lambat (yang melibatkan I/O disk) selesai.

Keterkaitan ke Teori Sistem Terdistribusi (Bab 1–7)

Bagian ini membahas secara mendalam keterkaitan antara desain dan implementasi sistem dengan konsep-konsep teoritis yang diuraikan dalam buku *Distributed Systems* (van Steen & Tanenbaum, 2023).

T1: Karakteristik Utama dan Trade-off Sistem Terdistribusi (Bab 1)

Sistem *log aggregator* ini mencerminkan beberapa karakteristik fundamental dari sistem terdistribusi seperti yang dijelaskan dalam Bab 1. **Berbagi sumber daya** (*resource sharing*) adalah inti dari sistem ini, di mana data log dibagikan antara berbagai *publisher* dan *consumer*. Arsitektur Pub-Sub yang diadopsi secara inheren menyediakan **transparansi distribusi** (*distribution transparency*), khususnya transparansi lokasi, karena *publisher* tidak perlu mengetahui di mana atau berapa banyak *consumer* yang ada. Sistem ini juga dirancang untuk **keterbukaan** (*openness*), memungkinkan penambahan sumber log atau sistem analisis baru tanpa modifikasi pada komponen yang ada.

Namun, desain ini juga merupakan manifestasi dari serangkaian *trade-off* yang tak terhindarkan. Tujuan utama sistem ini adalah **skalabilitas** (*scalability*) dan **keandalan** (*dependability*), yaitu kemampuan untuk menangani volume log yang besar tanpa kehilangan data. Untuk mencapai ini, pilihan arsitektur yang *loosely-coupled* (Pub-Sub) dan mekanisme pengiriman yang tangguh terhadap kegagalan jaringan menjadi prioritas. Mekanisme pengiriman yang paling sederhana dan andal adalah dengan melakukan *retry*, yang secara alami mengarah pada semantik pengiriman *at-least-once*. Keputusan ini mengutamakan keandalan pengiriman di atas segalanya. Namun, ini menciptakan masalah turunan: **duplikasi data**. Duplikasi mengancam integritas data dan konsistensi status akhir.

Oleh karena itu, sistem *harus* memperkenalkan mekanisme kompensasi di tingkat aplikasi: **idempotency** dan **deduplication**. Dengan demikian, keseluruhan desain sistem ini adalah hasil dari rantai *trade-off*: untuk mendapatkan skalabilitas dan keandalan, jaminan

exactly-once di tingkat transport dikorbankan, yang kemudian memaksa jaminan tersebut untuk dibangun kembali di tingkat aplikasi.

T2: Perbandingan Arsitektur Client-Server vs. Publish-Subscribe (Bab 2)

Bab 2 membedakan beberapa gaya arsitektur, termasuk *client-server* dan *publish-subscribe*.Arsitektur *client-server* tradisional ditandai oleh interaksi *request-reply* yang erat, di mana *client* harus mengetahui alamat *server* secara eksplisit. Jika diterapkan pada *log aggregator*, setiap sumber log (*publisher*) akan bertindak sebagai *client* yang secara langsung mengirimkan log ke *server aggregator*. Pola ini menciptakan kopling yang ketat (*tight coupling*).

Sebaliknya, arsitektur *publish-subscribe* yang dipilih untuk proyek ini memisahkan *publisher* dan *subscriber* melalui perantara (dalam kasus ini, antrean internal). Alasan teknis pemilihan ini adalah:

1. **Decoupling**
   Publisher tidak perlu mengetahui alamat, jumlah, atau bahkan keberadaan *consumer*. Hal ini memungkinkan penambahan sistem analisis atau pengarsipan baru di kemudian hari tanpa mengubah satu baris kode pun di sisi *publisher*, sesuai dengan prinsip *openness*.
2. **Skalabilitas**
   Dalam arsitektur *client-server*, *aggregator* akan menjadi *bottleneck* karena harus menangani koneksi dari semua *publisher*. Dengan Pub-Sub, beban dapat didistribusikan dengan lebih mudah, dan sistem dapat menangani lonjakan (*burst*) *traffic* log dengan menyerapnya di dalam antrean (*buffer*).
3. **Ketahanan (*Resilience*)**
   Jika salah satu *consumer* (misalnya, sistem analisis *real-time*) gagal, *publisher* dan *consumer* lainnya (misalnya, sistem pengarsipan) tidak terpengaruh dan dapat terus beroperasi.

T3: Semantik Pengiriman dan Idempotency (Konteks Bab 8)

Sistem terdistribusi harus menangani kegagalan, dan semantik pengiriman adalah kontrak tentang bagaimana kegagalan tersebut ditangani.

At-Least-Once Delivery

Ini adalah jaminan bahwa pesan akan terkirim minimal satu kali. Ini dicapai melalui mekanisme *acknowledgment* (ACK) dan *retry*. Jika *publisher* tidak menerima ACK, ia akan mengirim ulang pesan. Skenario ini, seperti yang diimplikasikan dalam diskusi tentang semantik RPC saat terjadi kegagalan , dapat menyebabkan pengiriman duplikat.

Exactly-Once Delivery

Ini adalah jaminan bahwa pesan akan diproses tepat satu kali. Dalam praktiknya, ini bukanlah jaminan di tingkat jaringan, melainkan sebuah "ilusi" yang dibangun di atas *at-least-once delivery* ditambah dengan deduplikasi di sisi *consumer*. Di sinilah peran krusial dari **idempotent consumer** muncul. Idempotensi adalah properti di mana sebuah operasi, jika dieksekusi berulang kali dengan input yang sama, menghasilkan efek yang sama seperti dieksekusi sekali. Dalam sistem dengan *retry*, *consumer* yang idempoten memastikan bahwa pemrosesan duplikat tidak berbahaya. Ini adalah sebuah "kontrak" arsitektural untuk toleransi kegagalan: infrastruktur pesan berjanji untuk mengirimkan pesan (*at-least-once*), dan aplikasi berjanji untuk dapat menangani duplikat tanpa efek samping. Dengan demikian, idempotensi memungkinkan sistem yang andal dibangun di atas jaringan yang tidak andal, sebuah prinsip

inti dari *fault tolerance*.

T4: Desain Skema Penamaan (Bab 6)

Penamaan adalah aspek fundamental dalam sistem terdistribusi untuk mengidentifikasi dan menemukan entitas. Bab 6 membedakan antara *flat naming* dan *structured naming*.

Penamaan topic

Untuk topic, skema **structured naming** digunakan, dengan format hierarkis seperti layanan.sumber.jenis_event. Contoh: pembayaran.api.transaksi_berhasil. Struktur ini memungkinkan *consumer* untuk berlangganan secara fleksibel (misalnya, ke pembayaran.api.*) dan memudahkan pengelolaan ruang nama.

Penamaan event_id

Untuk event_id, skema **flat naming** digunakan. Pengenal unik global seperti UUID (Universally Unique Identifier) sangat ideal. UUID dapat dihasilkan secara terdesentralisasi oleh setiap *publisher* tanpa memerlukan koordinasi, yang sangat penting untuk skalabilitas. Sifatnya yang hampir dijamin unik secara statistik membuatnya menjadi dasar yang kokoh untuk mekanisme deduplikasi. Tanpa event_id yang unik, mustahil bagi *consumer* untuk membedakan antara *event* baru yang sah dan pengiriman ulang dari *event* lama.

T5: Analisis Ordering Event (Bab 5)

Bab 5 membahas tentang koordinasi, termasuk sinkronisasi waktu dan urutan *event*.

Total Ordering

Menjamin bahwa semua komponen dalam sistem melihat semua *event* dalam urutan global yang sama persis. Ini sangat sulit dan mahal untuk dicapai, seringkali memerlukan algoritma konsensus.Kapan Total Ordering Tidak Diperlukan? Untuk aplikasi *log aggregator*, di mana setiap *event* log seringkali merupakan fakta yang independen, *total ordering* menjadi *over-engineering*. Misalnya, tidak masalah apakah log "User A login" diproses sebelum atau sesudah log "User B login". Yang mungkin lebih penting adalah *causal ordering* (jika *event* A menyebabkan *event* B, maka A harus diproses sebelum B), tetapi bahkan ini tidak diimplementasikan dalam sistem ini demi kesederhanaan.

Pendekatan Praktis dan Batasannya

Sistem ini mengandalkan timestamp ISO8601 yang dihasilkan oleh *publisher*. Pendekatan ini praktis tetapi memiliki batasan signifikan: **clock skew**. Jam antar mesin tidak pernah sinkron sempurna, bahkan dengan NTP.Akibatnya, *event* yang terjadi lebih dulu secara kausal bisa saja memiliki *timestamp* yang lebih akhir. Untuk menjamin urutan kausal, diperlukan jam logis seperti Lamport Clocks atau Vector Clocks.

T6: Mode Kegagalan dan Strategi Mitigasi (Bab 8)

Bab 8 membahas toleransi kegagalan dan berbagai mode kegagalan.

Mode Kegagalan yang Relevan:

Crash Failure: *Publisher* atau *consumer* berhenti bekerja.
Omission Failure: Pesan hilang di jaringan atau ACK dari *consumer* hilang.

Konsekuensi: Kegagalan ini, ketika digabungkan dengan mekanisme *retry*, mengarah pada dua masalah utama: **duplikasi *event*** dan pengiriman ***out-of-order***.

Strategi Mitigasi:
Retry with Exponential Backoff: Untuk menangani kegagalan sementara, *publisher* harus mencoba mengirim ulang. Menggunakan *exponential backoff* (meningkatkan jeda waktu antar percobaan secara eksponensial) mencegah *publisher* membanjiri *consumer* yang sedang dalam proses pemulihan.
Durable Dedup Store: Ini adalah strategi mitigasi utama untuk duplikasi dan *crash failure*. Dengan menyimpan event_id yang telah diproses di media persisten (SQLite), sistem memastikan bahwa bahkan setelah *restart*, *consumer* tidak akan memproses ulang *event* yang sama. Ini adalah bentuk implementasi dari mekanisme *recovery*.

T7: Eventual Consistency dalam Aggregator (Bab 7)

Bab 7 memperkenalkan berbagai model konsistensi, termasuk *eventual consistency*.

Definisi Eventual Consistency, Jika tidak ada pembaruan baru yang dilakukan pada suatu data, pada akhirnya semua akses ke data tersebut akan mengembalikan nilai yang terakhir diperbarui. **Penerapan pada Aggregator,** Sistem ini menunjukkan perilaku *eventual consistency*. Mungkin ada jeda singkat (latensi pemrosesan) antara saat sebuah *event* diterima oleh /publish dan saat ia muncul dalam hasil GET /events atau terakumulasi di GET /stats. Namun, jika aliran *event* berhenti, semua *endpoint* pada akhirnya akan menyatu (*converge*) ke keadaan yang sama dan konsisten yang mencerminkan semua *event* unik yang telah diproses. **Peran Idempotency + Dedup**, Mekanisme ini adalah penegak *convergence*. Tanpa mereka, setiap *retry* akan dianggap sebagai "pembaruan baru" yang salah, menyebabkan status agregat (seperti unique_processed) terus meningkat secara keliru dan tidak pernah mencapai keadaan akhir yang benar. Dengan menolak duplikat, idempotensi dan deduplikasi memastikan bahwa efek dari setiap *event* unik hanya diterapkan sekali, yang merupakan jaminan bahwa sistem akan selalu bergerak menuju keadaan yang konsisten.

T8: Metrik Evaluasi Sistem (Konteks Bab 1–7)

Evaluasi sistem terdistribusi memerlukan metrik yang kuantitatif.

Throughput
Jumlah *event* unik yang dapat diproses per detik. Metrik ini terkait langsung dengan **size scalability**. Keputusan desain seperti penggunaan antrean *in-memory* dan pemrosesan asinkron bertujuan untuk memaksimalkan *throughput*.
Latency
Waktu dari pengiriman *event* hingga pemrosesan selesai. Ini adalah metrik kunci untuk **geographical scalability** dan pengalaman pengguna. Pemisahan antara penerimaan dan pemrosesan meningkatkan *throughput* API dengan mengorbankan sedikit peningkatan pada latensi *end-to-end*.
Duplicate Rate
Persentase duplicate_dropped dari received. Ini bukan metrik performa sistem, melainkan indikator kesehatan input dan efektivitas mekanisme deduplikasi. Tingkat duplikasi yang tinggi mungkin menandakan masalah jaringan atau konfigurasi *retry* yang terlalu agresif di sisi *publisher*.

Daftar Pustaka

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