Nama: Azhari Rambe
NIM: 11221019
Mata Kuliah: Sistem Paralel dan Terdistribusi
Kelas: B

A. Soal Teori:

1. Jelaskan karakteristik utama sistem terdistribusi dan *trade-off* yang umum pada desain *Pub-Sub log aggregator*!

   Jawaban:

   Karakteristik utama yang mendefinisikan sebuah sistem terdistribusi adalah konkurensi, yaitu komponen-komponen yang berjalan secara paralel, tidak adanya jam global yang mana pada tiap *node* memiliki jamnya sendiri yang mungkin tidak sinkron, dan kegagalan independen, yaitu kegagalan satu komponen tidak serta-merta menghentikan komponen lain (Coulouris et al., 2012, Bab 1). Kemudian desain sistem terdistribusi selalu dihadapkan pada tantangan fundamental seperti heterogenitas, yaitu perbedaan perangkat keras, OS, dan jaringan, kemudahan untuk diperluas atau diintegrasikan, keamanan, skalabilitas atau kemampuan menangani pertumbuhan, penanganan kegagalan secara parsial, dan transparansi yang diartikan dengan menyembunyikan kompleksitas distribusi dari pengguna (Coulouris et al., 2012, Bab 1; van Steen & Tanenbaum, 2023, Bab 1).

   Pada desain *Pub-Sub log aggregator*, *trade-off* /pertukaran yang biasa terjadi adalah sebagai berikut:

   1. Kinerja (Latensi) vs Throughput/Skalabilitas:

      Untuk *log aggregator*, latensi rendah atau log yang cepat terlihat oleh *subscriber* adalah hal yang mungkin diinginkan. Namun usaha untuk mencapai latensi yang sangat rendah dapat membatasi *throughput*/jumlah log per detiknya karena sistem tidak dapat melakukan yang namanya *batching*. Sebaliknya, mengoptimalkan *throughput* dengan *batching* dapat meningkatkan latensi (van Steen & Tanenbaum, 2023, Bab 1).

   2. Durabilitas (Reliability) vs Kinerja:

      Jaminan bahwa setiap *log event* tidak akan pernah hilang atau yang kita maksud adalah durabilitas yang tinggi, contohnya dengan mewajibkan *broker* menulis log ke disk sebelum mengakui penerimaan akan dapat menurunkan kinerja/*throughput* dan latensinya karena tingginya *overhead* I/O (van Steen & Tanenbaum, 2023, Bab 1).

3. Konsistensi vs. Ketersediaan (CAP Theorem):

   Jika aggregator didistribusikan ke banyak *broker* yang dimanfaatkan untuk skalabilitas, maka kita harus memilih, apakah kita ingin *subscriber* yang berbeda dapat melihat urutan log yang sama persis dengan konsistensi yang kuat, atau kita ingin sistem selalu tersedia untuk menerima log baru dari *publisher* dengan ketersediaan yang tinggi, bahkan jika terjadi partisi jaringan (Coulouris et al., 2012, Bab 18).

4. Ekspresivitas Filter vs. Skalabilitas:

   Sistem *Pub-Sub* yang mengizinkan *filter* berbasis konten yang kompleks misalnya seperti mencari log error dari server X yang berisi kata 'SegFault'" jauh lebih fleksibel, tetapi secara signifikan lebih sulit untuk diskalakan daripada sistem berbasis topik sederhana misalnya, "log.error.serverX" karena *broker* harus memeriksa isi setiap pesan (Coulouris et al., 2012, Bab 6).

2. Bandingkan arsitektur *client-server* vs. *publish-subscribe* untuk aggregator. Kapan memilih *Pub-Sub*? Berikan alasan teknis!

   Jawaban:

   Arsitektur Client-Server (C/S) adalah sebuah model interaksi sinkron berbasis *request-reply* (Coulouris et al., 2012, Bab 2). Dalam konteks *log aggregator*, *publisher* akan bertindak sebagai *client* yang secara aktif mengirimkan/*push request* yang berisi log ke *aggregator*/server. Di sisi lain, *subscriber*/konsumen juga akan bertindak sebagai *client* yang harus secara aktif melakukan *polling*/menarik data ke server secara berkala untuk menanyakan apakah ada log baru untuk saya?. Model ini menciptakan *coupling*/keterikatan yang erat yang mana *publisher* dan *subscriber* harus mengetahui alamat jaringan *aggregator*, dan *subscriber* harus terus-menerus membuang sumber daya jaringan untuk *polling*, yang dapat menyebabkan latensi.

   Sebaliknya, arsitektur Publish-Subscribe (Pub-Sub) adalah paradigma pengiriman pesan asinkron (van Steen & Tanenbaum, 2023, Bab 4). Sistem ini memisahkan *publisher* dari *subscriber* melalui perantara, yaitu *broker* yang mana dalam hal ini adalah *log aggregator*. *Publisher* hanya mengirimkan *event* log ke *topic* tertentu tanpa mengetahui siapa atau berapa banyak *subscriber* yang ada. *Subscriber* mendaftarkan minatnya pada *topic* tertentu, dan *broker* yang bertanggung jawab mendorong/*push* log tersebut ke semua *subscriber* yang tertarik (Coulouris et al., 2012, Bab 6).

Model *Pub-Sub* dipilih ketika:

1. Dibutuhkannya Pemrosesan Asinkron:

   Publisher log misalnya aplikasi web tidak boleh diblokir atau diperlambat. Mereka hanya perlu menembakkan dan melupakan (*fire-and-forget*) log tersebut dan yakin bahwa log itu akan diproses nanti.

2. Banyaknya Konsumen yang Beragam:

   Ketika log yang sama perlu dikonsumsi oleh beberapa sistem yang berbeda misalnya, satu *subscriber* untuk *real-time dashboard*, satu lagi untuk arsip jangka panjang, dan satu lagi untuk sistem deteksi anomali.

3. Skalabilitas dan Elastisitas Konsumen Penting:

   Subscriber dapat ditambahkan atau dihapus kapan saja tanpa mempengaruhi *publisher* atau konfigurasi sistem.

Alasan teknis utama untuk memilih *Pub-Sub* adalah *decoupling*/pemisahan. Coulouris et al. (2012, Bab 6) mengidentifikasi ada tiga jenis *decoupling* yang disediakan oleh *Pub-Sub*:

1. Decoupling Ruang (Space):

   Publisher dan *subscriber* tidak perlu saling mengetahui identitas atau lokasi jaringan satu sama lain. Mereka hanya perlu tahu alamat *broker*.

2. Decoupling Waktu (Time):

   Komponen tidak harus aktif pada saat yang bersamaan. *Publisher* dapat mengirim log meskipun *subscriber* sedang *offline*. *Broker* akan menyimpan log tersebut dan mengirimkannya saat *subscriber* kembali *online*.

3. Decoupling Sinkronisasi (Synchronization):

   Operasi *publish* bersifat asinkron dan tidak memblokir. *Publisher* tidak perlu menunggu *subscriber* menerima pesan. Yang mana ini sangat berlawanan dengan model C/S *pull* (polling) yang bersifat sinkron dan tidak efisien.

3. Uraikan *at-least-once* vs *exactly-once delivery semantics*. Mengapa *idempotent consumer* krusial di *presence of retries*?

   Jawaban:

   Delivery semantics/semantik pengiriman mendefinisikan jaminan yang diberikan oleh sistem pesan mengenai berapa kali sebuah pesan akan dikirimkan ke penerima (van Steen & Tanenbaum, 2023, Bab 8).

   1. At-Least-Once (Setidaknya Sekali):

      Ini adalah jaminan bahwa sistem akan berusaha mengirimkan pesan, dan pesan tersebut dijamin akan tiba pada *consumer* minimal satu kali. Sistem misalnya *broker* akan terus mencoba mengirim ulang /*retry* pesan sampai *consumer* mengirimkan konfirmasi penerimaan atau disebut *acknowledgment/ACK*. Tantangannya, jika *consumer* berhasil memproses pesan tetapi *crash* sebelum mengirim *ACK*, atau jika *ACK* tersebut hilang di jaringan, maka *broker* akan salah mengira bahwa pesan itu gagal dan mengirimkannya lagi. Hal ini menyebabkan pemrosesan duplikat (Coulouris et al., 2012, Bab 15).

   2. Exactly-Once (Tepatnya Sekali):

      Ini adalah jaminan ideal di mana sistem memastikan setiap pesan dikirimkan dan diproses oleh *consumer* tepat hanya satu kali, tidak pernah hilang, dan tidak pernah terduplikasi. Dalam praktiknya, *exactly-once delivery* di tingkat infrastruktur jaringan hampir tidak mungkin dicapai secara efisien dalam sistem terdistribusi yang rentan terhadap kegagalan (van Steen & Tanenbaum, 2023, Bab 8). Seringkali, yang dicapai adalah *exactly-once processing* melalui kombinasi *at-least-once delivery* dengan mekanisme di sisi *consumer*.

   Mengapa Idempotent Consumer Krusial?

   Dalam *log aggregator*, kita tidak boleh kehilangan log (*at-most-once* bukan pilihan). Oleh karena itu, kita harus menggunakan *at-least-once delivery* dan menerima konsekuensinya: retries. *Retries* adalah mekanisme fundamental untuk mencapai *fault tolerance* (toleransi kegagalan) terhadap kegagalan jaringan sementara atau *consumer crash* (Coulouris et al., 2012, Bab 15).

   Di sinilah idempotency/idempotensi menjadi krusial. Operasi yang *idempotent* adalah operasi yang jika dieksekusi berkali-kali maka hasilnya akan sama, seolah-olah hanya dieksekusi satu kali.

   Ketika *broker* melakukan *retry* karena *ACK* hilang atau *consumer crash*, ia akan mengirimkan *event* yang sama atau duplikat, sehingga untuk menangani ini di rancanglah *consumer* yang *idempotent*. Caranya adalah dengan menggunakan *event_id* unik dari pesan tersebut. Sebelum

memproses *event*, *consumer* memeriksa *event_id* tersebut ke sebuah *durable dedup store* atau penyimpanan dedikasi yang persisten.

- Jika *event_id* belum pernah terlihat, *consumer* akan memproses *event* misalnya, menulis log ke database dan mencatat *event_id* tersebut di *store*.
- Jika *event_id* sudah ada di *store*, *consumer* tahu bahwa ini adalah duplikat dari *retry*, *consumer* akan melewatkan pemrosesan dan langsung mengirim *ACK* ke *broker*.

Dengan demikian, *idempotent consumer* memungkinkan kita menggunakan *at-least-once delivery* atau praktis dan tangguh terhadap kegagalan, yang secara efektif mencapai exactly-once processing yang merupakan tujuan bisnis kita.

4. Rancang skema penamaan untuk topic dan event_id (unik, *collision-resistant*). Jelaskan dampaknya terhadap dedup!

   Jawaban:

   Dalam sistem *Pub-Sub*, penamaan/*naming* adalah adalah hal krusial untuk mengidentifikasi dan mengelompokkan *event* (van Steen & Tanenbaum, 2023, Bab 4). Kita perlu merancang dua jenis nama yaitu topic untuk pengelompokan/kategori, dan event_id untuk identifikasi unik (individual).

   1. Skema Penamaan topic:

      Untuk topic, skema penamaan terstruktur/structured naming adalah yang paling cocok (van Steen & Tanenbaum, 2023, Bab 4). Tujuannya adalah untuk mengelompokkan log secara logis dan hierarkis.

      - Rancangan: Menggunakan format hierarkis yang dibatasi oleh titik (.).
      - Struktur: layanan.lingkungan.jenis_log
      - Contoh:
        - api.users.production.error (Log error dari layanan user di produksi)
        - api.payments.staging.info (Log info dari layanan payment di staging)
        - batch.invoice.production.audit (Log audit dari batch invoice di produksi)

      Desain ini sangat efisien karena memungkinkan *subscriber* menggunakan *wildcards* untuk berlangganan pola tertentu misalnya, api.*.production.error untuk semua *error* produksi di semua layanan API, sebuah fitur inti dari *topic-based publish-subscribe* (Coulouris et al., 2012, Bab 6).

2. Skema Penamaan event_id:

   Untuk event_id, tujuannya adalah keunikan global dan resistensi terhadap kolisi/collision-resistant. Setiap *event* log, di seluruh sistem harus memiliki ID yang tidak mungkin sama dengan *event* lain. Di sini, kita menggunakan penamaan datar/flat naming (van Steen & Tanenbaum, 2023, Bab 4).

   - Rancangan: UUID (Universally Unique Identifier), idealnya UUID versi 4 yang dihasilkan secara acak.
   - Contoh: 123e4567-e89b-12d3-a456-426614174000
   - Alasan: UUID 128-bit memiliki probabilitas kolisi yang sangat rendah secara astronomis. Ini memungkinkan setiap *publisher* misalnya, ratusan *node* aplikasi menghasilkan event_id secara mandiri dan konkurens tanpa perlu berkoordinasi dengan otoritas pusat yang akan menjadi *bottleneck*.

Dampak terhadap Deduplikasi (Dedup):

Dampak rancangan event_id terhadap deduplikasi sangatlah fundamental/mendasar. Seperti yang dibahas di soal ke 3, untuk mencapai *exactly-once processing* di atas *at-least-once delivery*, kita memerlukan *idempotent consumer*.

Mekanisme idempotensi ini bergantung sepenuhnya pada event_id yang unik. event_id (UUID) bertindak sebagai kunci deduplikasi (deduplication key).

Berikut, ketika *consumer* menerima sebuah *event* log:

1. Ia mengambil event_id dari *event* tersebut.
2. Ia memeriksa apakah event_id ini sudah ada di dalam *durable dedup store* atau penyimpanan data dedikasi, misal Redis Set.
3. Jika ID sudah ada, maka *event* itu adalah duplikat atau hasil dari *retry* dan dibuang atau dilewati.
4. Jika ID belum ada, maka *event* itu baru. *Event* tersebut diproses, dan event_id-nya segera dicatat di *durable dedup store* sebelum mengirim *ACK*.

Tanpa event_id yang unik dan *collision-resistant*, *consumer* tidak akan memiliki cara yang baik untuk membedakan *retry* yang sah dari *event* baru yang kebetulan datanya mirip.

5. Bahas *ordering*: kapan *total ordering* tidak diperlukan? Usulkan pendekatan praktis (mis. *event timestamp* + *monotonic counter*) dan batasannya!

   Jawaban:

   Ordering/pengurutan adalah jaminan mengenai urutan *event* yang akan dikirimkan ke *subscriber*. Dalam sistem *Pub-Sub*, jaminan *ordering* yang paling ketat adalah total ordering, di mana semua *event* dari semua *publisher* dikirimkan ke semua *subscriber* dalam urutan yang sama persis. Untuk mencapainya, sistem memerlukan mekanisme koordinasi global seperti *atomic broadcast* atau *sequencer* terpusat yang sangat mahal dan menjadi *bottleneck* kinerja (Coulouris et al., 2012, Bab 14).

   Kapan Total Ordering Tidak Diperlukan? *Total ordering* tidak diperlukan untuk *log aggregator* jika:

   1. Event Bersifat Komutatif:

      Jika operasi pada log bersifat komutatif yang mana hasilnya sama, terlepas dari urutannya, dan *total ordering* tidak relevan. Contohnya seperti menghitung jumlah total *request* HTTP 500. Tidak masalah *request* mana yang dihitung terlebih dahulu.

   2. Analisis Tidak Memerlukan Urutan Global:

      Banyak kasus penggunaan *log* hanya memerlukan urutan yang lebih lemah, seperti FIFO ordering per-publisher, misal semua log dari *publisher* A tiba berurutan atau causal ordering, jika log A menyebabkan log B, maka A harus datang sebelum B. Misalnya, untuk men-debug sesi pengguna, kita hanya peduli pada urutan *event* dari pengguna tersebut, bukan urutan global relatif terhadap pengguna lain (Coulouris et al., 2012, Bab 6, 14).

   3. Ketersediaan dan Throughput Lebih Penting:

      Menegakkan *total ordering* sangat membatasi skalabilitas dan ketersediaan sistem. Untuk *log aggregator* yang harus menangani volume *event* yang sangat tinggi/high *throughput*, mengorbankan *total ordering* adalah *trade-off* yang umum.

   Pendekatan Praktis: Kita dapat menggunakan physical clocks/jam fisik untuk Event Timestamp + Monotonic Counter yang menyediakan pengurutan *best-effort* yang cukup baik untuk menganalisis tanpa biaya *total ordering* (van Steen & Tanenbaum, 2023, Bab 6).

   - Pendekatan: Setiap *publisher* menyematkan *timestamp* UTC presisi tinggi misalnya, milidetik atau nanodetik pada setiap *event* log saat *event* itu dibuat.
   - Masalah: Dua *event* dapat terjadi pada milidetik yang sama, dan jam antar *publisher* tidak pernah sinkron secara sempurna (*clock skew*).

- Solusi yang diusulkan: Menggabungkan *timestamp* dengan monotonic counter/pencacah yang selalu naik di sisi *publisher*.
    - Setiap *publisher* mengelola pencacah internalnya sendiri.
    - Jika *timestamp event* baru sama dengan *timestamp event* sebelumnya, maka *counter* dinaikkan.
    - Jika *timestamp event* baru lebih besar, maka *counter* di-reset, misal ke 0.
    - *Event* kemudian dapat diurutkan oleh *consumer* menggunakan *timestamp* sebagai kunci utama dan *counter* serta ID *publisher* untuk mengatasi *tie-break* antar *node* sebagai kunci sekunder.

Batasan Pendekatan Ini:

1. Bergantung pada Sinkronisasi Jam:

   Pendekatan ini sangat bergantung pada asumsi bahwa jam di semua *node publisher* disinkronkan menggunakan protokol seperti NTP (Network Time Protocol). Jika terjadi *clock skew* yang signifikan, maka urutan *event* di *aggregator* tidak akan mencerminkan urutan kejadian di dunia nyata (van Steen & Tanenbaum, 2023, Bab 6).

2. Bukan Jaminan Kausalitas:

   Timestamp fisik tidak dapat menangkap kausalitas atau hubungan sebab-akibat antar *node*. *Event* A bisa menyebabkan *event* B di *node* lain, tetapi karena latensi jaringan dan *clock skew*, *event* B bisa saja mendapatkan *timestamp* yang lebih awal daripada A (Coulouris et al., 2012, Bab 14).

3. Latensi Jaringan:

   Event yang lebih lama atau timestamp lebih awal bisa saja mengalami keterlambatan jaringan dan tiba di *consumer* setelah *event* yang lebih baru. Sehingga *consumer* memerlukan *buffer* dan *windowing* untuk mengurutkan ulang *event* yang datang *out-of-order*, yang menambah latensi pemrosesan.

6. Identifikasi *failure modes* (duplikasi, *out-of-order*, *crash*). Jelaskan strategi mitigasi (*retry*, *backoff*, *durable dedup store*)!

   Jawaban:

   Dalam *log aggregator*, *failure modes*/mode kegagalan adalah cara-cara spesifik sistem dapat gagal. Mengacu pada terminologi standar, kegagalan yang paling relevan adalah:

   1. Crash Failures:

      Komponen seperti Publisher, Broker, atau Consumer berhenti bekerja sepenuhnya (Coulouris et al., 2012, Bab 15). Jika *consumer crash* setelah memproses *event* tetapi sebelum mengirim *ACK*, maka *broker* akan menganggap *event* gagal dan mengirimnya lagi.

   2. Omission Failures:

      Pesan seperti log atau *ACK* hilang saat transit di jaringan (Coulouris et al., 2012, Bab 15). Jika *ACK* dari *consumer* ke *broker* hilang, maka ini akan memicu pengiriman ulang, sama seperti *consumer crash*.

   3. Timing Failures (Out-of-Order):

      Pesan tiba dalam urutan yang berbeda dari yang diharapkan, hal ini seringkali terjadi karena latensi jaringan yang bervariasi atau *clock skew* antar *publisher* (van Steen & Tanenbaum, 2023, Bab 6).

   Kegagalan 1 dan 2 secara langsung menyebabkan *event* duplikasi, sementara kegagalan 3 menyebabkan *event* out-of-order.

   Strategi Mitigasi:

   1. Retry (Percobaan Ulang):

      Ini adalah strategi fundamental untuk menangani *omission failures* atau *transient crash failures*. Ketika *broker* tidak menerima *ACK* dari *consumer* dalam batas waktu (*timeout*) tertentu, *broker* akan berasumsi *event* gagal terkirim dan mencoba mengirimkannya lagi (van Steen & Tanenbaum, 2023, Bab 8). Inilah yang menciptakan semantik *at-least-once*.

   2. Exponential Backoff (Penundaan Eksponensial):

      Retry yang naif atau yang mencoba ulang secepat mungkin dapat memperburuk masalah, seperti menciptakan badai *retry* yang membebani *consumer* yang mungkin sedang dalam proses pemulihan. Strategi *backoff* memerintahkan *broker* untuk menunggu dalam interval waktu yang meningkat secara eksponensial misalnya, 1s, lalu 2s, lalu 4s, 8s sebelum setiap *retry* baru. Ini memberi

consumer sebuah ruang bernapas untuk pulih (Coulouris et al., 2012, Bab 15).

7. Definisikan *eventual consistency* pada aggregator; jelaskan bagaimana *idempotency* + *dedup* membantu mencapai konsistensi!

   Jawaban:

   Eventual Consistency/Konsistensi Akhir adalah model konsistensi yang menjamin bahwa, jika tidak ada pembaruan log baru yang masuk ke sistem, semua replika atau dalam hal ini, semua *subscriber* yang berlangganan *topic* yang sama pada akhirnya akan konvergen ke keadaan yang sama (van Steen & Tanenbaum, 2023, Bab 7).

   Dalam konteks *log aggregator*, ini berarti:

   - Tidak ada jaminan bahwa semua *subscriber* akan melihat log yang sama pada detik yang sama, yang mana ada jeda propagasi.
   - Seorang *subscriber* mungkin sementara melihat data yang basi atau belum menerima log terbaru atau bahkan menerima log *out-of-order*.
   - Namun, sistem menjamin bahwa pada akhirnya, setiap *subscriber* akan menerima semua log dan status akhir mereka akan konsisten satu sama lain misalnya, jumlah total *error* yang dihitung oleh semua *subscriber* pada akhirnya akan sama.

   Bagaimana Idempotency + Deduplication Membantu:

   Untuk mencapai *eventual consistency* di *log aggregator*, kita harus menjamin dua hal: Pertama semua log akhirnya akan tiba, dan Kedua log tersebut diproses dengan benar.

   1. Menjamin Kedatangan (Liveness):

      Sistem menggunakan semantik *at-least-once delivery* seperti yang dibahas di soal nomor 3 dan 6. Melalui mekanisme *retry*, *broker* secara agresif memastikan bahwa *event* log akan terus dikirim ulang sampai *consumer*/subscriber berhasil mengirim *ACK*. Hal ini menjamin bahwa log tidak akan hilang karena kegagalan jaringan atau *consumer crash* sementara (Coulouris et al., 2012, Bab 15).

   2. Menjamin Kebenaran (Safety):

      Masalah dari *at-least-once delivery* adalah duplikasi. Jika *consumer* memproses duplikat, maka keadaan akhir mereka tidak akan konsisten misalnya, satu *subscriber* menghitung 100 log, maka *subscriber* lain yang menerima duplikat menghitung 102 log).

   Di sinilah Idempotency yang dicapai melalui Deduplication berperan. Dengan menggunakan event_id unik untuk memeriksa *durable dedup store*

seperti yang dibahas di soal nomor 4 dan 6), *consumer* yang *idempotent* dapat dengan aman melewatkan duplikat yang disebabkan oleh *retry*.

Kombinasi dari *retry* (*at-least-once*) dan *idempotency* (*dedup*) inilah yang memungkinkan *eventual consistency*:

- *At-least-once* menjamin bahwa data pada akhirnya akan tiba (properti *liveness* dari konsistensi).
- *Idempotency/Dedup* menjamin bahwa data yang tiba berkali-kali hanya akan diproses satu kali (properti *safety/correctness* dari konsistensi).

Hasilnya, semua *subscriber* pada akhirnya dijamin akan memproses kumpulan log yang identik, tepat satu kali.

8. Rumuskan metrik evaluasi sistem (*throughput*, *latency*, *duplicate rate*) dan kaitkan ke keputusan desain!

   Jawaban:

   Evaluasi kinerja/*performance* sebuah *log aggregator* yang berfokus pada seberapa cepat dan andal ia memindahkan data. Berikut tiga metrik utamanya:

   1. Throughput (Hasil):

      Diukur sebagai jumlah *event* atau total ukuran data yang dapat diterima dan diproses oleh sistem per satuan waktu misalnya, 100.000 *event*/detik. Ini adalah ukuran kapasitas sistem (van Steen & Tanenbaum, 2023, Bab 1).

   2. Latency (Latensi):

      Waktu tunda (*delay*) yang dialami sebuah *event* log. Paling sering diukur sebagai *end-to-end latency*, contohnya waktu sejak *publisher* mengirimkan *event* hingga *consumer* menerimanya, sering diukur dalam milidetik (ms) (van Steen & Tanenbaum, 2023, Bab 1).

   3. Duplicate Rate (Tingkat Duplikasi):

      Persentase *event* yang diproses oleh *consumer* lebih dari satu kali. Yang mana dalam sistem ideal dengan *exactly-once processing*, metrik ini harus 0%.

   Kaitan Metrik dengan Keputusan Desain:

   Metrik-metrik ini sering kali saling bertentangan atau *trade-off*, dan keputusan desain kita secara langsung mempengaruhinya:

- Trade-off Throughput vs. Latency: Ini adalah *trade-off* yang paling fundamental/mendasar.
    - Keputusan Desain (Batching): Untuk mencapai throughput tinggi, *publisher* dan *broker* biasanya mengumpulkan banyak *event* ke dalam satu *batch*/kelompok sebelum mengirimkannya melalui jaringan. *Consumer* juga memproses *event* secara *batch*. Desain ini sangat efisien, tetapi secara signifikan meningkatkan latensi karena *event* harus menunggu *batch* penuh terlebih dahulu.

- Kaitan Duplicate Rate, Latency, dan Reliability:
    - Keputusan Desain (At-Least-Once Delivery): Untuk mencapai *reliability* (seperti pada soal nomor 3 dan 6), kita memilih semantik *at-least-once* yang mengandalkan *retry*. Keputusan ini secara inheren meningkatkan *duplicate rate* mentah, yaitu jumlah duplikat yang *tiba* di *consumer*.

    - Keputusan Desain (Idempotency & Durable Dedup Store): Untuk melawan *duplicate rate* tersebut (seperti yang dibahas pada soal nomor 7), kita mendesain *idempotent consumer* menggunakan *durable dedup store,* misalnya Redis. Keputusan ini menurunkan *duplicate rate* yang diproses menjadi 0%. Namun, ini memiliki biaya, contohnya setiap *event* sekarang memerlukan pemeriksaan ke *store* eksternal atau pengecekan event_id, yang menambahkan latensi pada setiap pemrosesan *event*.

- Kaitan Throughput dan Arsitektur:
    - Keputusan Desain (Pub-Sub Decoupling): Pilihan arsitektur *Pub-Sub* (seperti yang dibahas pada soal nomor 2) yang memisahkan/*decouple publisher* dan *consumer* adalah kunci untuk skalabilitas (Coulouris et al., 2012, Bab 6). Hal ini memungkinkan kita menambah *consumer* secara independen tanpa membebani *publisher*, dan memungkinkan sistem mencapai throughput horizontal yang tinggi.

3. Durable Dedup Store (Penyimpanan Deduplikasi Persisten):

   Ini adalah mitigasi utama untuk duplikasi yang disebabkan oleh *retry*. Seperti dibahas di T3 dan T4, *consumer* menggunakan penyimpanan persisten (disebut *durable* agar data tidak hilang saat *consumer crash*) untuk mencatat event_id dari setiap *event* yang telah berhasil diproses. Sebelum memproses *event* baru, *consumer* memeriksa *store* ini. Jika event_id sudah ada, *event* tersebut adalah duplikat dan akan dilewati (menjadikan *consumer idempotent*).