BAB I

1.1 Karakteristik Utama Sistem Terdistribusi

Sistem terdistribusi adalah kumpulan komputer independen yang tampil sebagai satu kesatuan bagi pengguna, bertujuan berbagi sumber daya perangkat keras dan perangkat lunak melalui model client-server. Tiga karakteristik utama sistem ini adalah:

- Konkurensi (Concurrency): Komponen berjalan paralel dan mengakses sumber daya bersamaan, meningkatkan kinerja namun menimbulkan tantangan sinkronisasi dan pengendalian konflik data.
- Tidak Adanya Jam Global: Setiap node memiliki waktu internal sendiri tanpa sinkronisasi sempurna, menyulitkan penentuan urutan kejadian global akibat clock skew.
- Kegagalan Independen: Node dapat gagal terpisah tanpa menghentikan sistem keseluruhan, sehingga toleransi kesalahan menjadi elemen penting.
- Prinsip desain lainnya mencakup berbagi sumber daya, transparansi (menyembunyikan kompleksitas distribusi), keterbukaan (interoperabilitas melalui standar API), keamanan (autentikasi, otorisasi, enkripsi), dan skalabilitas (efisiensi meski beban meningkat).

1.2 Trade-off Umum pada Desain Pub-Sub Log Aggregator

Sistem log aggregator berbasis publish-subscribe seperti Kafka mengatur komunikasi melalui broker, dengan berbagai trade-off:

- Durabilitas vs. Latensi: Durabilitas tinggi (penyimpanan ke disk) aman namun meningkatkan latensi; latensi rendah (ACK langsung) cepat tetapi berisiko kehilangan data.
- Jaminan Pengiriman vs. Kinerja: At-most-once cepat namun berisiko hilang; at-least-once menjamin pengiriman tetapi menimbulkan duplikasi; exactly-once akurat namun kompleks dan lambat.
- Ketersediaan vs. Konsistensi (CAP): Ketersediaan tinggi (AP) tetap menerima pesan saat partisi namun konsistensi tertunda; konsistensi kuat (CP) menjamin keseragaman data tetapi dapat berhenti saat partisi.
- Urutan Pesan vs. Throughput: Urutan global menjamin keteraturan namun throughput rendah; urutan per partisi memungkinkan pemrosesan paralel dengan throughput tinggi.

Pemilihan strategi bergantung pada kebutuhan aplikasi keandalan data, latensi rendah, atau skalabilitas tinggi—untuk merancang solusi yang efisien dan konsisten.

BAB II

2.1 Perbedaan

Aspek	Client-Server	Publish-Subscribe
Model komunikasi	Client mengirim permintaan langsung ke server dan menunggu respon (request-reply).	Publish dan subscribe dilakukan secara asinkron tanpa saling mengenal secara eksplisit.

Keterikatan temporal dan referensial	Terikat temporal dan referensial, client dan server harus aktif saat komunikasi.	Dekoupling temporal dan referensial, publisher dan subscriber tidak perlu saling aktif bersamaan.
Interaksi	Ada referensi eksplisit antar entitas, client harus tahu server.	Dekoupling dengan event bus atau shared data space, tidak ada referensi eksplisit.
Penyimpanan data	Server menyimpan data, melayani permintaan client.	Middleware bisa menyimpan notifikasi jika subscriber perlu mengambil data secara eksplisit.
Skalabilitas	Terbatas, server bisa menjadi bottleneck saat banyak client.	Lebih skalabel karena proses publish dan subscribe terpisah secara independen.
Kompleksitas implementasi	Lebih sederhana, model komunikasi langsung.	Lebih kompleks, membutuhkan matching subscription dan notifikasi, penanganan event composition.
Pola penggunaan	Cocok untuk operasi sinkron seperti query database, sistem login, transaksi langsung.	Cocok untuk sistem event-driven, notifikasi, monitoring, sistem dengan banyak subscriber.
Ketergantungan status	Server menyimpan state dari objek layanan (stateful).	Bersifat stateless pada middleware, state biasanya disimpan oleh publisher atau subscriber.

2.2 Kapan Memilih Arsitektur Publish-Subscribe

Pub-Sub ideal untuk sistem log aggregator modern karena menawarkan skalabilitas, fleksibilitas, dan decoupling tinggi. Pilih Pub-Sub ketika:

- Diperlukan keterlepasan referensial dan temporal antar komponen
- Sistem melibatkan banyak proses yang bergabung/berhenti dinamis
- Dibutuhkan penanganan event asinkron (monitoring, notifikasi, log real-time)
- Sistem memerlukan notifikasi selektif berbasis topik/konten
- Subscriber tidak selalu aktif, butuh buffering
- Diharapkan reduksi kompleksitas koordinasi dan peningkatan modularitas

Alasan Teknis:

- Keterlepasan: Menghilangkan ketergantungan langsung, meningkatkan toleransi kesalahan dan fleksibilitas.
- Komunikasi Asinkron: Pengiriman non-blocking meningkatkan throughput dan menurunkan latensi.
- Matching Ekspresif: Topic-based atau content-based menyalurkan pesan selektif sesuai kebutuhan.

- Skalabilitas: Pemisahan processing dan coordination memungkinkan pertumbuhan tanpa mengganggu komponen lain.
- Ketahanan: Middleware (Kafka, RabbitMQ) mendukung buffering, persistence, dan notifikasi otomatis.

Bentuk Decoupling:

- Space: Publisher tidak perlu tahu lokasi subscriber
- Time: Tidak harus aktif bersamaan, broker menyimpan log sementara
- Asynchronous: Publisher mengirim tanpa menunggu tanggapan

Pub-Sub unggul untuk sistem event-driven kompleks berskala besar, menawarkan fleksibilitas dan ketahanan melalui buffering, asynchrony, dan decoupling yang mendukung sistem tangguh dan mudah diadaptasi.

BAB III

3.1 Uraian At-Least-Once vs Exactly-Once Delivery Semantics

Semantik pengiriman pesan menjelaskan jaminan sistem terdistribusi mengenai frekuensi pengiriman atau eksekusi pesan. Dua model utama adalah:

1. **At-Least-Once Delivery**

   Sistem menjamin setiap pesan dikirim minimal satu kali. Pengiriman ulang (retries) memastikan pesan tidak hilang—pengirim terus mengirim ulang hingga menerima konfirmasi. Namun, jika konfirmasi hilang di jaringan, penerima dapat mengeksekusi operasi yang sama lebih dari sekali. *Contoh*: Permintaan "tambah saldo $10" dikirim dua kali akibat timeout, menyebabkan saldo bertambah ganda. Model ini menjamin keandalan pengiriman tetapi tidak mencegah duplikasi eksekusi.

2. **Exactly-Once Delivery**

   Menjamin setiap pesan dikirim dan dieksekusi tepat satu kali, tanpa kehilangan atau duplikasi. Meskipun ideal, penerapannya sangat kompleks karena sulit membedakan kegagalan jaringan, keterlambatan, atau crash server. Untuk mendekati jaminan ini, digunakan mekanisme seperti pelacakan nomor urut (sequence ID), deduplikasi di sisi server, atau protokol two-phase commit yang memastikan hanya satu eksekusi sah.

3.2 Pentingnya Idempotent Consumer dalam Sistem Pengiriman Ulang

Dalam sistem at-least-once, pengiriman ulang menyebabkan consumer berpotensi menerima pesan sama berkali-kali. Untuk mencegah efek samping salah, diperlukan idempotency sifat operasi yang memberikan hasil akhir sama meskipun dijalankan berulang kali.

Contoh operasi:

- **Idempoten**: Membaca saldo akun, menetapkan nilai tetap (setBalance(100)), menambah elemen ke set

- **Non-idempoten**: Menambah saldo (+10), menambah elemen ke list

Dengan menerapkan idempotent consumer, sistem mempertahankan konsistensi data meskipun terjadi retry atau duplikasi. Jika operasi tidak idempoten, diperlukan mekanisme deduplikasi berbasis ID unik atau message sequence untuk mendeteksi eksekusi ganda.

Kesimpulan: At-least-once memberikan keandalan tinggi namun berisiko duplikasi, sedangkan exactly-once menawarkan jaminan sempurna namun sulit dicapai. Strategi paling realistis adalah mengkombinasikan at-least-once delivery dengan idempotent consumer untuk menjamin integritas data, keandalan proses, dan ketahanan sistem terhadap gangguan komunikasi.

BAB IV

4.1 Skema Penamaan Topic

Dalam arsitektur publish-subscribe, topic berfungsi sebagai saluran komunikasi untuk pengelompokan dan penyaringan event. Skema penamaan yang baik harus hierarkis untuk mendukung fleksibilitas subscription dan skalabilitas sistem.

Rancangan: Format [domain].[context].[resource].[event_type]

Contoh: logs.production.auth-service.error, metrics.production.database.cpu-load

Justifikasi Teknis: Struktur hierarkis memudahkan filtering granular dengan wildcard:

- logs.production.*.error → seluruh log kesalahan di produksi
- logs.*.auth-service.* → semua log dari layanan autentikasi
- logs.# → seluruh pesan dalam domain logs

Pendekatan ini meningkatkan modularitas dan efisiensi routing tanpa menambah kompleksitas koordinasi antar komponen.

4.2 Skema Penamaan event_id

Setiap event membutuhkan pengenal unik (event_id) yang collision-resistant dan dapat dihasilkan secara terdesentralisasi.

Rancangan yang Disarankan:

UUID v4 (Random): Menggunakan nilai 128-bit acak yang menjamin keunikan global tanpa koordinasi pusat.

UUID v7 (Time-based): Kombinasi timestamp dan bit acak, bersifat K-sortable, memudahkan pengurutan kronologis dan meningkatkan performa penyimpanan berbasis waktu.

Kedua pendekatan memberikan keunikan tinggi dan memungkinkan setiap publisher menghasilkan ID secara independen, menghindari bottleneck koordinasi.

4.3 Dampak terhadap Deduplikasi

Deduplikasi memastikan setiap pesan hanya diproses satu kali, meskipun dikirim ulang dalam sistem at-least-once.

Peran topic: Tidak berpengaruh langsung terhadap deduplikasi karena berfungsi sebagai routing channel, bukan identitas pesan.

Peran event_id: Menjadi elemen kunci dalam deduplikasi. Consumer menyimpan setiap event_id yang telah diproses dalam idempotency store (Redis, cache, atau basis data). Saat menerima pesan baru:

1. Jika event_id belum tercatat → pesan diproses dan ID disimpan
2. Jika event_id sudah ada → pesan diabaikan sebagai duplikat

Mekanisme ini mengubah semantik at-least-once menjadi effectively-once di tingkat aplikasi, tanpa memerlukan protokol koordinasi kompleks.

Kesimpulan: Desain hierarkis topic meningkatkan efisiensi distribusi pesan, sedangkan event_id berbasis UUID memungkinkan deduplikasi kuat dan independen. Kombinasi keduanya menciptakan sistem publish-subscribe yang reliabel, terdistribusi, dan idempoten.

BAB V

5.1 Konsep Pengurutan dalam Sistem Terdistribusi

Pengurutan (ordering) adalah jaminan terhadap urutan pemrosesan pesan (event ordering) di antara proses yang berkomunikasi. Tiga model utama:

1. **Total Ordering**: Menjamin seluruh proses melihat pesan dalam urutan global sama. Jika pesan m_1 dikirim sebelum m_2, semua node menerima m_1 sebelum m_2. Model ini menjamin konsistensi sempurna namun memiliki latensi tinggi karena memerlukan mekanisme konsensus global.
2. **Causal (Partial) Ordering**: Menjamin urutan hanya untuk pesan yang memiliki hubungan sebab-akibat (happened-before). Pesan konkuren (tidak berkaitan) dapat diproses dalam urutan berbeda antar node. Pendekatan ini lebih efisien dan umum digunakan dalam sistem berskala besar.
3. **FIFO Ordering**: Menjamin pesan dari satu pengirim diproses sesuai urutan pengirimannya, tanpa menjamin urutan antar pengirim. Model ini ringan dan cocok untuk sistem dengan banyak sumber pesan independen.

5.2 Kebutuhan dan Trade-off Pengurutan

Pengurutan total tidak selalu diperlukan, khususnya pada aplikasi yang tidak bergantung pada urutan global (sistem log, notifikasi, pemantauan sensor). Causal ordering atau FIFO ordering mencukupi bila:

- Urutan hanya penting untuk pesan dari sumber yang sama
- Aplikasi hanya membutuhkan konsistensi kausal antar pesan yang saling bergantung

Total ordering digunakan hanya pada sistem yang menuntut urutan absolut, seperti replikasi data finansial atau transaksi bank.

5.3 Pendekatan Praktis: Event Timestamp dan Monotonic Counter

Pendekatan efisien menggunakan kombinasi timestamp peristiwa dan monotonic counter lokal, atau **Lamport Logical Clock**:

1. Setiap node memiliki penghitung waktu logis yang meningkat setiap kali terjadi peristiwa
2. Setiap pesan dikirim bersama nilai timestamp tersebut
3. Penerima memperbarui jam logisnya berdasarkan nilai maksimum antara jam lokal dan timestamp yang diterima

Pendekatan ini menjamin urutan parsial (causal order) tanpa memerlukan sinkronisasi waktu global.

5.4 Keterbatasan Pendekatan Timestamp

Kelemahan utama:

- Tidak menjamin kausalitas penuh karena dua peristiwa konkuren dapat memiliki urutan waktu berbeda di tiap node
- Ketergantungan pada sinkronisasi jam fisik dapat menyebabkan clock skew
- Monotonic counter hanya menjamin urutan lokal, bukan global

Untuk sistem yang membutuhkan urutan kausal penuh, dapat digunakan **Vector Clock**, meskipun lebih kompleks dan mahal secara komputasi.

Kesimpulan: Total ordering hanya dibutuhkan ketika urutan global seluruh peristiwa harus konsisten di setiap node. Untuk sebagian besar aplikasi Pub-Sub atau sistem log terdistribusi, causal ordering atau kombinasi timestamp + monotonic counter sudah memadai, memberikan keseimbangan antara efisiensi kinerja, latensi rendah, dan tingkat konsistensi yang cukup tanpa membebani sistem dengan mekanisme konsensus kompleks.

BAB VI

BAB VI: Failure Modes dan Strategi Mitigasi

6.1 Failure Modes dalam Sistem Pesan Terdistribusi

Dalam sistem terdistribusi, komunikasi antar komponen tidak andal akibat keterlambatan, crash, atau kehilangan paket. Tiga mode kegagalan utama:

1. **Crash (Kegagalan Proses)**: Terjadi ketika komponen (publisher, broker, atau consumer) berhenti beroperasi tiba-tiba. Dari sisi pengirim, kegagalan ini hanya tampak sebagai timeout, sehingga tidak dapat dibedakan apakah penerima benar-benar crash atau hanya terlambat merespons.
2. **Duplikasi Pesan**: Disebabkan oleh mekanisme retry untuk menjamin pengiriman. Jika pesan telah diproses tetapi acknowledgment (ACK) hilang di jaringan, publisher mengirim ulang pesan yang sama, sehingga consumer menerima dan memprosesnya lebih dari satu kali.

3. **Pesan Tidak Berurutan (Out-of-Order)**: Terjadi ketika pesan diterima dalam urutan berbeda dari pengiriman aslinya. Hal ini umum terjadi akibat perbedaan jalur jaringan atau buffering delay, dan dapat mengganggu logika bisnis bila urutan pemrosesan bersifat kritis.

6.2 Strategi Mitigasi

Untuk menjaga keandalan sistem dan konsistensi data, beberapa strategi mitigasi diterapkan:

1. **Retry (Pengiriman Ulang)**: Strategi dasar untuk menghadapi crash atau kehilangan pesan. Publisher mengirim ulang pesan jika tidak menerima ACK dalam jangka waktu tertentu. Mekanisme ini mengubah semantik at-most-once menjadi at-least-once, tetapi berpotensi menghasilkan duplikasi.
2. **Exponential Backoff (Penundaan Adaptif)**: Peningkatan interval pengiriman ulang secara eksponensial (misalnya: 1s, 2s, 4s, 8s) dengan tambahan jitter acak. Teknik ini mencegah lonjakan beban akibat retry storm saat terjadi kegagalan massal, serta membantu menstabilkan sistem selama pemulihan.
3. **Durable Deduplication Store**: Mekanisme di sisi consumer untuk mencegah duplikasi pesan akibat retry. Setiap pesan memiliki event_id unik yang disimpan dalam basis data tahan crash. Saat pesan baru diterima:
    - Jika event_id sudah ada, pesan diabaikan
    - Jika belum, pesan diproses dan event_id dicatat secara atomik sebelum ACK dikirim

Pendekatan ini menghasilkan perilaku idempotent consumer dan mendekati semantik exactly-once.

6.3 Prinsip Tambahan

Penerapan idempotent consumer menjadi krusial untuk memastikan hasil pemrosesan konsisten walaupun terjadi pengiriman ulang. Sistem modern seperti Kafka, AWS SQS, dan Google Pub/Sub mengombinasikan unique message ID dengan jendela deduplikasi (deduplication window) guna mencapai keseimbangan antara konsistensi, performa, dan durabilitas.

Kesimpulan: Sistem pesan terdistribusi tidak dapat sepenuhnya menghindari kegagalan, tetapi dapat mengontrol dampaknya. Kombinasi strategi retry, exponential backoff, dan durable deduplication memberikan keseimbangan optimal antara keandalan pengiriman, kinerja sistem, dan ketahanan terhadap crash. Dengan penerapan idempotent consumer, sistem dapat mencapai tingkat keandalan yang mendekati exactly-once delivery semantics meskipun berjalan di lingkungan yang tidak stabil.

BAB VII

7.1 Definisi *Eventual Consistency*

Eventual Consistency merupakan model konsistensi lemah (*weak consistency*) yang menekankan ketersediaan dan skalabilitas dalam sistem terdistribusi. Model ini menyatakan bahwa apabila tidak terjadi pembaruan baru, maka seluruh replika data pada akhirnya (*eventually*) akan konvergen pada keadaan yang sama.

Dalam konteks *log aggregator*, model ini menjamin bahwa:

1. Setiap log yang dipublikasikan akan sampai ke seluruh konsumen pada akhirnya.

2. Tidak terdapat batas waktu pasti kapan log tersebut akan diterima.
3. Urutan penerimaan log antar konsumen tidak dijamin identik (tidak ada jaminan *global order*).

Pendekatan ini banyak digunakan pada sistem seperti **DNS**, **Coda**, dan **Amazon Dynamo**, yang mengandalkan replikasi *lazy propagation* melalui mekanisme *gossip protocol* agar tetap beroperasi saat terjadi *partition* jaringan.

7.2. Peran Idempotensi dan Deduplikasi terhadap Konsistensi

Pada sistem dengan model pengiriman *at-least-once*, pengirim melakukan *retry* ketika gagal menerima konfirmasi (*ACK*). Meskipun meningkatkan jaminan keberhasilan pengiriman, strategi ini dapat menimbulkan duplikasi pesan, yang berpotensi mengakibatkan inkonsistensi pada *aggregator*.

Untuk mempertahankan *eventual consistency*, diperlukan dua mekanisme kunci:

1. **Deduplikasi (Deduplication)**
   Setiap log harus memiliki pengenal unik (event_id). *Aggregator* memelihara *durable deduplication store* yang mencatat seluruh event_id yang telah diproses.
   - Jika event_id belum ada → log diproses dan ID disimpan.
   - Jika event_id sudah tercatat → log diabaikan.

     Mekanisme ini memastikan bahwa setiap log diproses tepat satu kali (*effectively-once*), meskipun dikirim berulang.
2. **Idempotensi (Idempotency)**
   Idempotensi adalah sifat suatu operasi yang menghasilkan efek akhir yang sama meskipun dieksekusi berulang kali. Dengan adanya *deduplication*, operasi "tambah log jika belum ada" menjadi idempoten secara efektif: pemrosesan berulang atas event_id yang sama tidak mengubah keadaan sistem.

Kombinasi deduplikasi dan idempotensi memungkinkan sistem *log aggregator* dengan model pengiriman *at-least-once* mencapai eventual consistency. Walaupun pesan dapat dikirim lebih dari satu kali atau diterima tidak berurutan, keadaan akhir sistem akan tetap konsisten karena setiap log dipastikan diproses hanya sekali.

BAB VII

Metrik	Nilai	Deskripsi
received	11	Jumlah total event yang diterima sistem dari publisher
unique_processed	3	Jumlah event unik yang berhasil diproses setelah deduplikasi
duplicate_dropped	8	Jumlah event duplikat yang terdeteksi dan diabaikan
topics	hai, Hai, test	Daftar topik aktif yang dikirim oleh publisher
uptime (detik)	487.32	Lama waktu layanan aktif tanpa gangguan

Sistem *event log aggregator* ini menggunakan arsitektur Publish–Subscribe untuk menerima dan memproses event secara asinkron. Berdasarkan hasil uji, dari total 11 event yang diterima, hanya 3 event unik yang diproses, sementara 8 lainnya terdeteksi sebagai duplikasi. Hal ini menunjukkan bahwa mekanisme idempotensi dan deduplication berfungsi efektif melalui penyimpanan *persistent* berbasis SQLite.

Secara teoritis, desain ini menerapkan prinsip at-least-once delivery dan fault tolerance (Bab 4–5), di mana event duplikat dapat muncul akibat retransmisi, tetapi tidak diproses ulang. Dari sisi *reliability* dan *consistency*, sistem mempertahankan integritas data meskipun mengalami *restart*, sesuai konsep *durability* pada sistem terdistribusi.

Metrik *uptime* menegaskan stabilitas layanan, mendukung teori *availability* dan *scalability* (Bab 6–7). Dengan demikian, sistem berhasil mencapai keseimbangan antara konsistensi lokal dan ketersediaan sebagaimana dijelaskan dalam CAP Theorem (Bab 3), menjadikannya solusi efisien untuk agregasi log terdistribusi berskala kecil hingga menengah.

Daftar Pustaka

Tanenbaum, A. S., & Van Steen, M. (2007).

Distributed Systems: Principles and Paradigms **(2nd ed.). Pearson Prentice Hall.**