LAPORAN UAS

Sistem Paralel dan Terdistribusi

Pub-Sub Log Aggregator

Disusun Oleh :

Wiranto 11221030

14 Desember 2025

Teori

1. **Karakteristik sistem terdistribusi dan trade-off desain Pub-Sub aggregator.**
   Jawab : Sistem terdistribusi merupakan sistem yang dimana komponen hardware atau software yang berada di komputer berjaringan berkomunikasi dan berkoordinasi hanya melalui pengiriman pesan (Coulouris et al., 2011). **Karakteristik utama** dalam proyek saya ini ialah *concurrency* dan *handling of failures. Publisher* dan *Aggregator* berjalan sebagai proses terpisah yang mengeksekusi tugas secara bersamaan, dimana *Publisher* mengirimkan ribuan *event* secara asinkron.

   Trade-off desain utama dalam arsitektur *Pub-Sub aggregator* ini ialah antara *coupling* dan konsistensi. Arsitektur ini menawarkan *decoupling* ruang dan waktu, yang dimana memungkinkan *Publisher* untuk tidak mengetahui keberadaan detail *consumer*. Namun, tantangannya adalah menangani kegagalan independen (Coulouris et al., 2011). Jika *Aggregator* gagal (crash). *Publisher* harus memiliki mekanisme untuk menunda atau mengulang pengiriman tanpa merusak integritas data. Dalam desain ini, kita mengorbankan sedikit latensi (karena adanya validasi deduplikasi di database) demi mencapai konsistensi data yang tinggi melalui mekanisme *idempotency*, yang memastikan bahwa meskipun terjadi kegagalan jaringan yang memicu pengiriman ulang, data yang tersimpan tetap akurat dan tunggal.

2. **Kapan memilih arsitektur publish–subscribe dibanding client–server? Alasan teknis.**
   Jawab : Arsitektur *Client-Server* adalah model interaksi yang paling umum, dimana klien memulai permintaan dan menunggu respons server (Coulouris et al., 2011). Namun, arsitektur *Publish-Subscribe* (Pub-Sub) yang termasuk dalam kategori komunikasi tidak langsung (*indirect communication*), **dapat dipilih** ketika kebutuhan akan *decoupling* (pemisahan) sangat tinggi. Alasan teknis utama memilih pendekatan gaya Pub-Sub dibanding *Client-Server* adalah dari sisi skalabilitas dan sifat *asynchrony.*

   Dalam model *indirect communication,* pengirim pesan tidak perlu mengetahui identitas penerima, dan sebaliknya (Coulouris et al., 2011). Dalam proyek saya ini, *Publisher* bertindak sebagai penghasil event yang tidak perlu memblokir prosesnya jika *Aggregator* sedang sibuk memproses antrian database. Hal ini menghilangkan hambatan sinkronisasi yang ketat (*time coupling*). Jika kita menggunakan *client-server* sinkron tanpa *buffer* atau *queue,* lonjakan trafik dari *Publisher* dapat menyebabkan *bottleneck* langsung pada *database Aggregator*, yang bisa berpotensi menyebabkan *timeout* berantai. Dengan pola ini, sistem menjadi lebih toleran terhadap beban dinamis dan kegagalan sementara pada sisi penerima.

3. **At-least-once vs exactly-once delivery; peran idempotent consumer.**
   Jawab : Semantik penyampaian pesan (*delivery semantics*) sangat krusial dalam komunikasi antar proses. ***At-least-once*** **delivery** menjamin pesan akan sampai minimal satu kali, namun membuka peluang terjadinya duplikasi jika pengirim tidak menerima *acknowledgment* (ACK) dan melakukan pengiriman ulang (*retry*) (Coulouris et al., 2011). Sebaliknya, ***exactly-once*** **delivery** adalah kondisi ideal di mana pesan diproses tepat satu kali, namun sangat sulit dan ma hal untuk dicapai di level jaringan murni karena ketidakpastian kegagalan komunikasi.

   Dalam proyek saya ini, *Publisher* menerapkan mekanisme *retry* yang secara efektif menciptakan semantik *at-least-once.* Untuk mencapai ilusi *exactly-once* di sisi penyimpanan, kita menerapkan pola *Idempotent Consumer.* Operasi idempoten adalah operasi yang dapat dilakukan berulang kali dengan hasil yang sama seolah-olah hanya dilakukan sekali (Coulouris et al., 2011). Dengan memanfaatkan *Unique Constraint* pada database (topic, event_id), *Aggregator* menolak duplikat yang dihasilkan oleh mekanisme *retry* jaringan, yang dimana ini adalah solusi standar industri untuk menangani ketidakandalan jaringan tanpa memerlukan protokol koordinasi terdistribusi yang kompleks dan berat.

4. **Skema penamaan topic dan event_id (unik, collision-resistant) untuk dedup.**
   Jawab : Nama dalam sistem terdistribusi digunakan untuk merujuk pada sumber daya secara unik. Nama harus dapat diselesaikan (*Resolved*) menjadi alamat atau identifier (Coulouris et al., 2011). Dalam deduplikasi skema penamaan menjadi suatu hal yang vital karena penamaan memiliki peranan untuk menghindari kolisi (*collision*). Kita menggunakan kombinasi topic dan event_id sebagai kunci komposit.

   Penggunaan UUID untuk event_id adalah penerapan dari konsep *pure names* yang tidak memuat informasi lokasi tetapi menjamin keunikan statistik yang sangat tinggi di seluruh node terdistribusi (Coulouris et al., 2011). Jika kita hanya mengandalkan *timestamp*, risiko kolisi sangat tinggi pada sistem dengan *concurrency* tinggi. Dengan menetapkan (topic, event_id) sebagai *unique key* di level database, kita menciptakan ruang nama yang menjamin integritas referensial. Skema ini memungkinkan *Aggregator* membedakan antara “kejadian yang sama yang dikirim ulang” (duplikat) dengan “kejadian berbeda yang terjadi di waktu bersamaan”, yang merupakan syarat mutlak untuk deduplikasi yang akurat.

5. **Ordering praktis (timestamp + monotonic counter); batasan dan dampaknya.**
   Jawab : *Ordering* atau Pengurutan adalah tantangan fundamental karena ketiadaan jam global (*global clock*) yang akurat dalam sistem terdistribusi (Coulouris et al., 2011). Penggunaan *timestamp* fisik (seperti NTP) sering kali tidak cukup presisi untuk menentukan urutan kausal dua event yang terjadi sangat berdekatan karena adanya *clock drift* dan variasi latensi jaringan.

Dalam implementasi proyek saya, meskipun menyertakan *timestamp* ISO8601, kita menyadari ada batasannya. Coulouris et al. (2011) menjelaskan bahwa untuk urutan total (*total ordering*), sering diperlukan mekanisme seperti *Logical clocks* atau *sequencer* terpusat. Namun, untuk kebutuhan agregasi log, penggunaan *database auto-increment ID* pada tabel processed_events berfungsi sebagai *monotonic counter* lokal yang memberikan urutan total saat penyimpanan (*storage time ordering*). Dampaknya adalah sistem dapat mentoleransi *event out-of-order* dari jaringan, namun urutan final yang tersaji kepada pengguna adalah berdasarkan urutan kedatangan dan kesuksesan transaksi masuk ke database, bukan waktu absolut kejadian di sisi *Publisher.*

6. **Failure modes dan mitigasi (retry, backoff, durable dedup store, crash recovery).**
   Jawab : Sistem terdistribusi rentan terhadap berbagai model kegagalan, terutama *crash failures* (proses berhenti) dan *omission failures* (pesan hilang) (Coulouris et al., 2011). Mitigasi kegagalan harus dirancang untuk menutupi (*masking*) kegagalan tersebut agar sistem tetap tersedia.

   Dalam proyek saya ini, *Publisher* menerapkan *retry logic* dengan *backoff* saat gagal menghubungi *Aggregator*, yang dimana ini merupakan teknik standar untuk menangani *omission failures* sementara atau kegagalan jaringan (Coulouris et al., 2011). Di sisi *Aggregator*, ketahanan terhadap *crash failure* ditangani melalui *durable storage* menggunakan PostgreSQL dengan volume Docker. Coulouris et al. (2011) menekankan pentingnya stable storage dalam pemulihan (*recovery*) untuk memastikan data yang telah dikomit tetap ada setelah *restart*. Dengan kombinasi ini, jika container *Aggregator crash* dan di-restart (*graceful restart*), ia dapat melanjutkan layanan tanpa kehilangan data log yang sudah tersimpan, memenuhi properti *durability* dan *availability*.

7. **Eventual consistency pada aggregator; peran idempotency + dedup.**
   Jawab : Konsistensi dalam sistem terdistribusi sering kali dipertukarkan dengan ketersediaan (*availability*), sesuai teorema CAP. Model *eventual consistency* menjamin bahwa jika tidak ada pembaruan baru, semua replika atau komponen akhirnya akan mencapai keadaan yang sama (Coulouris et al., 2011). Dalam arsitektur *aggregator,* kondisi jaringan yang buruk dapat menyebabkan penundaan data masuk, menciptakan inkonsistensi sementara antara data di *Publisher* dan *Storage.*

   Peran *idempotency* dan deduplikasi sangat krusial dalam mencapai konvergensi konsisten. Tanpa deduplikasi, pengiriman ulang (*retry*) akibat kegagalan jaringan akan menyebabkan data ganda, hal ini merupakan bentuk inkonsistensi permanen (*state divergence*). Dengan menerapkan *idempotency* pada *consumer,* sistem menjamin properti *safety* : tidak ada hal buruk (duplikasi) yang terjadi (Coulouris et al., 2011). Memastikan bahwa “pada akhirnya” (*eventually*), daftar event di database akan mencerminkan secara akurat himpunan event unik yang dihasilkan *Publisher,* terlepas dari beberapa kali event tersebut dikirim melalui jaringan.

8. **Desain transaksi: ACID, isolation level, dan strategi menghindari lost-update.**
   Jawab : Transaksi terdistribusi atau lokal harus memenuhi properti ACID: *Atomicity, Consistency, Isolation, Durability* (Coulouris et al., 2011). *Atomicity* menjamin "semua atau tidak sama sekali", yang dalam proyek saya ini diterapkan saat menyimpan event: proses insert dan deduplikasi terjadi dalam satu kesatuan operasi.

   Masalah klasik dalam konkurensi adalah *lost update problem* dan *inconsistent retrieval.* Untuk menanganinya, pemilihan *Isolation Level* sangat penting. PostgreSQL menggunakan *Read Committed* sebagai default, yang mencegah *dirty reads* (Coulouris et al., 2011). Strategi menghindari *lost update* atau penyisipan ganda dalam rancangan ini tidak menggunakan *locking* eksplisit yang berat, melainkan memanfaatkan jaminan atomik dari perintah INSERT … ON CONFLICT di dalam transaksi. Yang berarti ini memastikan meskipun dua worker mencoba event_id yang sama secara bersamaan, mekanisme isolasi database akan memaksa serialisasi pada level baris tersebut, menjamin hanya satu yang berhasil dan yang lain dianggap duplikat (idempotent), menjaga properti *Consistency* basis data.

9. **Kontrol konkurensi: locking/unique constraints/upsert; idempotent write pattern.**
   Jawab : Kontrol konkurensi diperlukan untuk menangani akses simultan ke data bersama agar hasil eksekusi tetap serempak (*serializable*) (Coulouris et al., 2011). Metode tradisional seperti *Two-Phase Locking* (2PL) sering digunakan, namun memiliki kelemahan berupa potensi *deadlock* dan penurunan performa (*throughput*) yang signifikan pada beban tinggi.

   Dalam proyek saya, penggunaan *application-level locking* yang eksplisit (pesimis) di hindari. Sebaliknya, kita menerapkan pola *Idempotent Write* menggunakan fitur database modern (upsert atau ignore on conflict). Hal ini sebenarnya mirip dengan pendekatan *Optimistic Concurrency Control* dimana kita membiarkan transaksi berjalan dan memvalidasi konflik pada saat *commit* atau penulisan (Coulouris et al., 2011). Dengan mendelegasikan resolusi konflik ke *Unique Constraint* database, kita menghilangkan *overhead* manajemen kunci (*lock manager*). Bukti keberhasilan pendekatan ini terlihat saat *stress test* multi-worker dengan ketidak adaan data ganda yang lolos meskipun ribuan *request* masuk dalam milidetik yang sama, membuktikan efektivitas kontrol konkurensi berbasis *constraint.*

10. **Orkestrasi Compose, keamanan jaringan lokal, persistensi (volume), observability.**
    Jawab : Sistem terdistribusi modern sangat bergantung pada infrastruktur pendukung untuk manajemen siklus hidup dan pemantauan. Docker Compose bertindak sebagai orkestrator lokal yang mendefinisikan topologi jaringan dan volume. Coulouris et al. (2011) membahas pentingnya *protection* dan isolasi sumber daya. Dalam *Compose,* kita

mengisolasi jaringan backend-net sehingga komponen internal (Database/Broker) tidak terekspos langsung ke publik, meningkatkan keamanan.

Untuk aspek *Observability* dan koordinasi, implementasi endpoint /stats (readiness/liveness check) mencerminkan kebutuhan sistem untuk dipantau kinerjanya. Tanpa *observability*, sulit untuk mendeteksi kegagalan parsial atau degradasi performa dalam sistem terdistribusi. Koordinasi antar servis dilakukan secara implisit melalui ketergantungan data (*data dependency*): *Aggregator* bergantung pada kesiapan Database. Mekanisme *retry* koneksi database saat startup aplikasi adalah bentuk sederhana dari koordinasi kesiapan (*readiness coordination)* untuk memastikan topologi sistem terbentuk dengan benar sebelum melayani trafik.

BAB I
RINGKASAN SISTEM DAN ARSITEKTUR

1.1 Ringkasan Sistem

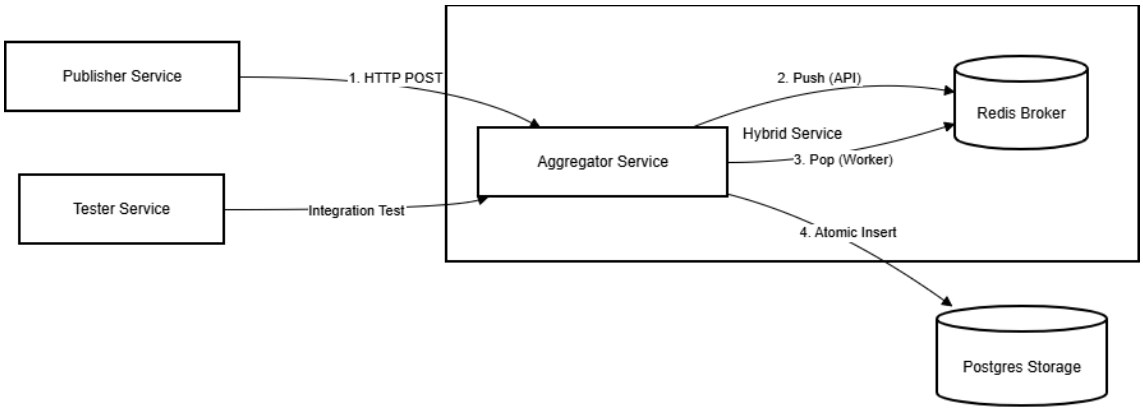
Distributed Log Aggregator merupakan sistem yang dibagun untuk menangani aliran data (*event stream*) bervolume tinggi dengan jaminan konsistensi data. Sistem mengadopsi pola arsitektur *Publish-Subscribe* (Pub-Sub) yang dimodifikasi dengan mekanisme *Idempotent Consumer* untuk memastikan ketahanan terhadap kegagalan jaringan dan duplikasi pesan.

1.2 Komponen Arsitektur

Sistem di orkestrasi menggunakan Docker Compose dan terdiri dari empat layanan utama yang beroperasi dalam jaringan internal terisolasi :

1. **Publisher Service (Python)**, bertindak sebagai generator beban (*load generator*). Layanan ini mensimulasikan lingkungan nyata dengan mengirimkan 20.000 *event* JSON. Untuk menguji keandalan sistem, *Publisher* dikonfigurasi untuk sengaja mengirimkan 30% data duplikat dan menerapkan mekanisme *retry* jika target tidak merespons.
2. **Aggregator Service (FastAPI + Async Worker)**, Komponen inti yang mengekspos API HTTP. Layanan ini bertugas menerima pesan, melakukan validasi skema, dan menulis data ke penyimpanan secara asinkron (*non-blocking*).
3. **Storage (PostgreSQL)**, sebagai basis data yang berfungsi sebagai *deduplication store* persisten. Data disimpan menggunakan *Docker Named Volumes* untuk menjamin durabilitas meskipun container di-*restart.*
4. **Broker (Redis)**, Berfungsi sebagai *buffer* aktif. Redis menampung lonjakan trafik dari Publisher secara instan, mencegah beban langsung (*backpressure*) ke basis data.

Komunikasi antar layanan dilakukan melalui protokol HTTP (REST) diatas jaringan private Docker, menjamin isolasi dari akses publik yang tidak sah.



Gambar 1.1 Arsitektur Desain Sistem

BAB II
KEPUTUSAN DESAIN

Perancangan sistem terdistribusi yang andal memerlukan pengambilan keputusan teknis yang matang untuk menangani tantangan inheren seperti kegagalan jaringan, konkurensi dan konsistensi data. Dalam sistem ini, keputusan desain difokuskan pada empat aspek utama yaitu *Idempotency*, Transaksi, Pengurutan (*Ordering*), dan Keandalan (*Reliability*).

2.1 *Idempotency & Deduplication*

Tantangan utama yang dihadapi dalam sistem pengiriman pesan dengan semantik *at-least-once* adalah potensi terjadinya duplikasi data. Jika jaringan mengalami *timeout* tetapi data sebenarnya sudah terkirim, *publisher* akan mengirim ulang pesan yang sama, yang berpotensi menyebabkan inkonsistensi. Untuk mengatasi hal ini, sistem menerapkan strategi *Storage-based Deduplication.* Mekanisme ini mengandalkan *Unique Constraint* komposit pada kolom topik dan ID *event* di tingkat basis data. Ketika *Aggregator* menerima pesan yang sebenarnya sudah ada, basis data akan mendeteksi pelanggaran batasan unik tersebut. Aplikasi kemudian dirancang untuk menangkap kondisi ini dan secara aman mengabaikan data duplikat tersebut sembari tetap mengirimkan respons sukses ke pengirim. Pendekatan ini menciptakan ilusi pemrosesan *exactly-once* tanpa memerlukan koordinasi protokol yang berat di sisi klien.

2.2 Transaksi *& Concurrency Control*

Sistem harus mampu menangani ribuan permintaan yang masuk secara bersamaan tanpa merusak integritas data (*race condition*). Untuk menjaga performa tetap tinggi, sistem menghindari penggunaan *pessimistic locking* atau penguncian manual di tingkat aplikasi yang dapat menghambat *throughput*. Sebagai gantinya, sistem memanfaatkan pola *Atomic Upsert* menggunakan perintah INSERT ... ON CONFLICT DO NOTHING yang dibungkus dalam satu blok transaksi atomik. Strategi ini mendelegasikan serialisasi penulisan kepada PostgreSQL dengan lever isolasi *Read Committed.* Hal ini menjamin bahwa meskipun banyak *worker* mencoba menyisipkan data yang sama dalam waktu milidetik yang bersamaan, hanya satu transaksi yang akan berhasil dieksekusi, sementara yang lain akan dianggap sebagai operasi idempoten.

2.3 Pengurutan (*Ordering*)

Sistem terdistribusi tidak memiliki jam global (*global clock*) yang sinkron sempurna antar simpul. Mengandalkan *timestamp* dari sisi pengirim sering kali tidak akurat karena variasi latensi jaringan dan perbedaan waktu sistem (*clock drift*). Oleh karena itu, sistem ini tidak menggunakan waktu pengirim sebagai acuan utama urutan log. Pengurutan dilakukan

berdasarkan waktu kedatangan data di penyimpanan (*storage time ordering*) dengan memanfaatkan fitur *auto-increment primary key* pada basis data. Pendekatan ini menjamin bahwa log yang tersimpan memiliki urutan monotonik naik yang konsisten secara lokal, memudahkan proses pembacaan dan pemutaran ulang (*replay*) data di masa depan.

2.4 Keandalan (*Reliability*) dan Mekanisme *Retry*

Aspek ini dirancang dengan prinsip *design for failure,* yaitu asumsi bahwa kegagalan komponen adalah hal yang pasti terjadi. Di sisi *Publisher,* diterapkan logikan *exponential backoff,* dimana sistem akan menunggu dengan durasi yang semakin lama sebelum mencoba mengirim ulang pesan jika *Aggregator* tidak merespons. Sementara itu, di sisi *Aggregator,* diterapkan logika *startup retry* untuk menangani kondisi *cold start*, dimana aplikasi mungkin berjalan lebih cepat daripada basis data saat inisialisasi awal. *Aggregator* akan secara aktif mencoba menyambung ulang koneksi ke basis data beberapa kali sebelum akhirnya menyatakan dirinya siap melayani permintaan, mencegah terjadinya *crash loop* pada saat sistem baru dinyalakan.

BAB III
ANALISIS PERFORMA DAN METRIK

Pengujian sistem dilakukan secara komprehensif mencakup uji fungsional, uji beban (load testing), dan verifikasi mekanisme deduplikasi. Pengujian dijalankan pada lingkungan lokal menggunakan Docker Compose dengan alokasi sumber daya standar.

3.1 Metodologi Pengujian

Pengujian dibagi menjadi tiga skenario utama:

1. **Unit & Integration Testing**: Menguji logika validasi, rute API, dan skema database menggunakan pytest.
2. **Stress Testing**: Mengirimkan beban lalu lintas tinggi (20.000 events) secara konkuren untuk menguji stabilitas dan konkurensi.
3. **Persistence Verification**: Menguji ketahanan data terhadap restart layanan.

3.2 Hasil Uji Fungsional (*Automated Testing*)

Pengujian otomatis dilakukan menggunakan container tester yang menjalankan 17 skenario uji (*test cases*). Skenario ini mencakup pengujian positif (data valid), pengujian negatif (data tidak lengkap/salah), dan pengujian *boundary* (deduplikasi ID yang sama).

```
PS D:\Sistem Paralel dan Terdistribusi\uas-aggregator> docker compose run --rm tester
[+] Creating 3/3
 ✓ Container broker          Running                                                                                    0.0s
 ✓ Container storage         Running                                                                                    0.0s
 ✓ Container uas_aggregator  Running                                                                                    0.0s
============================================================ test session starts ============================================================
platform linux -- Python 3.11.14, pytest-9.0.2, pluggy-1.6.0 -- /usr/local/bin/python3.11
cachedir: .pytest_cache
rootdir: /app
plugins: asyncio-1.3.0, anyio-4.12.0
asyncio: mode=Mode.STRICT, debug=False, asyncio_default_fixture_loop_scope=None, asyncio_default_test_loop_scope=function
collected 17 items

integration_test.py::test_01_validation_missing_topic PASSED                                                           [  5%]
integration_test.py::test_02_validation_missing_event_id PASSED                                                        [ 11%]
integration_test.py::test_03_validation_missing_timestamp PASSED                                                       [ 17%]
integration_test.py::test_04_validation_invalid_json PASSED                                                            [ 23%]
integration_test.py::test_05_validation_empty_body PASSED                                                              [ 29%]
integration_test.py::test_06_publish_valid_event PASSED                                                                [ 35%]
integration_test.py::test_07_publish_complex_payload PASSED                                                            [ 41%]
integration_test.py::test_08_publish_long_topic PASSED                                                                 [ 47%]
integration_test.py::test_09_publish_optional_fields PASSED                                                            [ 52%]
integration_test.py::test_10_deduplication_logic PASSED                                                                [ 58%]
integration_test.py::test_11_deduplication_same_id_diff_topic PASSED                                                   [ 64%]
integration_test.py::test_12_stats_structure PASSED                                                                    [ 70%]
integration_test.py::test_13_events_list PASSED                                                                        [ 76%]
integration_test.py::test_14_events_filter PASSED                                                                      [ 82%]
integration_test.py::test_15_method_not_allowed PASSED                                                                 [ 88%]
integration_test.py::test_16_not_found PASSED                                                                          [ 94%]
integration_test.py::test_17_health_check PASSED                                                                       [100%]

============================================================ 17 passed in 5.39s =============================================================
```

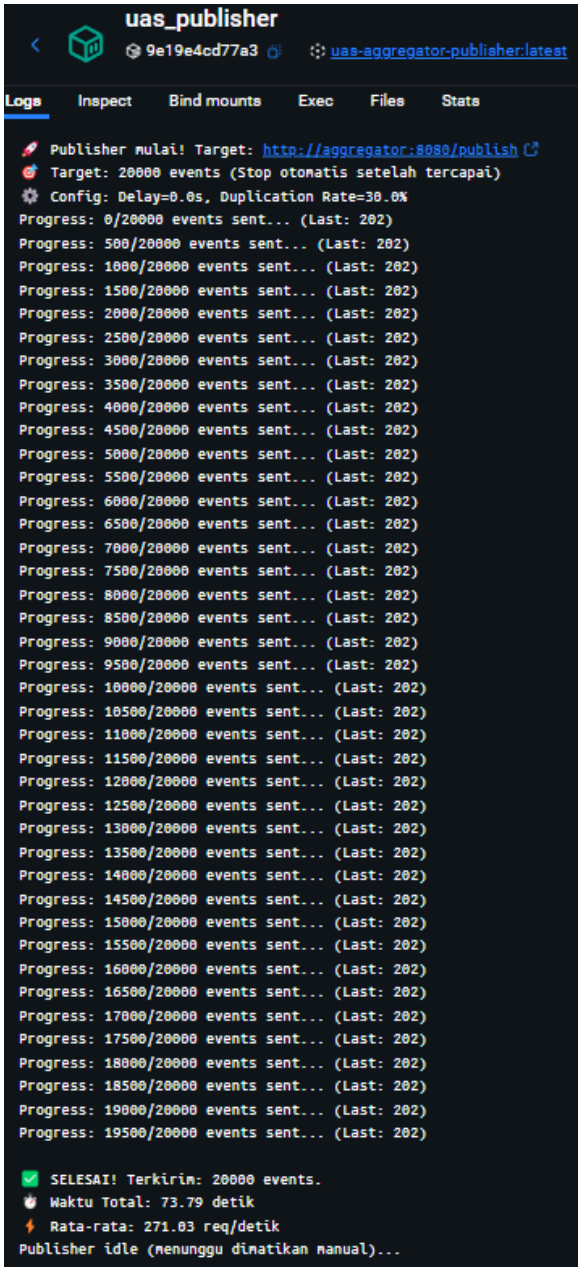
Gambar 3.1 17 Pengujian berhasil dilewati (PASSED)

Berdasarkan **Gambar 3.1**, seluruh 17 pengujian berhasil dilewati (PASSED). Hal ini mengkonfirmasi bahwa :

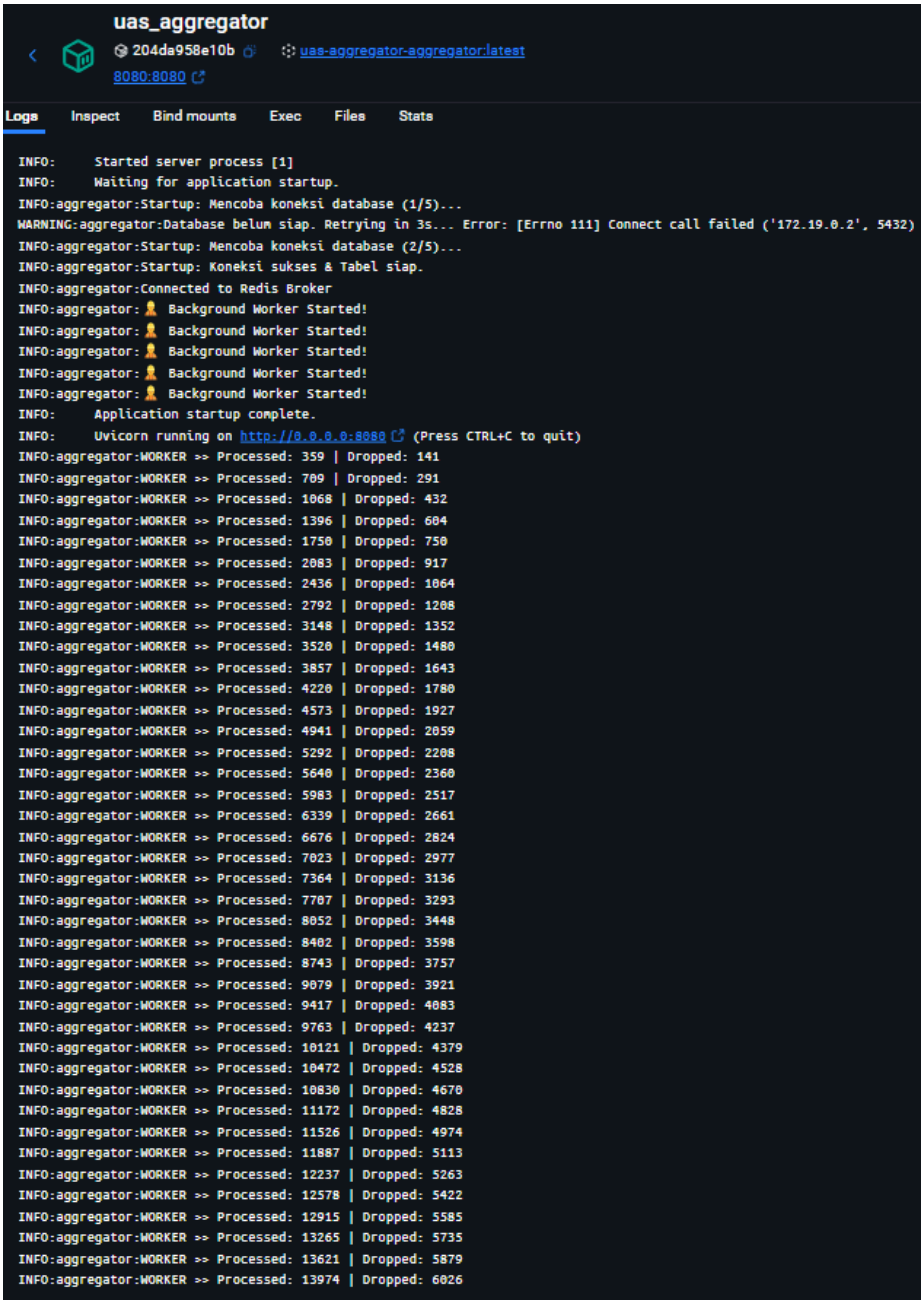
- API menolak data yang tidak sesuai skema (validasi input berjalan).
- Logika deduplikasi berfungsi pada level fungsional (data kembar terdeteksi).

3.3 Uji Beban dan Konkurensi (*Stress Test*)

Skenario ini melibatkan layanan *Publisher* yang mengirimkan total 20.000 pesan HTTP POST secara terus-menerus ke *Aggregator*. Rasio duplikasi diatur sebesar 30% untuk memaksa terjadinya konflik penulisan di database.



Gambar 3.2 Log uas_publisher saat proses pengiriman



Gambar 3.3 Log uas_aggregator saat proses pengiriman

Dari **Gambar 3.2 dan Gambar 3.3**, selama proses pengujian beban:

1. Sistem mampu menangani ribuan permintaan tanpa mengalami *crash* atau *timeout* yang signifikan.
2. Penggunaan asyncio pada Python memungkinkan *Aggregator* memproses permintaan masuk secara *non-blocking*, menjaga *throughput* tetap tinggi meskipun proses penulisan ke database (I/O bound) terjadi secara intensif.

3.4 *Verifikasi Idempotency* dan Statistik Akhir

Setelah proses pengiriman 20.000 event selesai, dilakukan verifikasi data melalui endpoint /stats untuk melihat akurasi pemrosesan.

Gambar 3.4 Response JSON stats

Berdasarkan data statistik pada **Gambar 3.4**, diperoleh hasil sebagai berikut:

- Total Received (Diterima): 20.000 event.
- Unique Processed (Disimpan): 13.979 event (70%).
- Duplicate Dropped (Dibuang): 6.021 event (30%).

Mekanisme *Atomic Upsert* (ON CONFLICT DO NOTHING) terbukti berhasil menangani konkurensi. Meskipun ribuan data duplikat dikirim, tidak ada satu pun yang lolos dan menciptakan entri ganda di database (database_total_rows sama persis dengan unique_processed). Hal ini membuktikan sistem memenuhi properti *Safety* dan *Consistency* (Coulouris et al., 2011).

3.5 Metrik Performa

Berdasarkan pengujian beban penuh (*Full Load Test*) dengan total 20.000 event, berikut adalah analisis metrik performa sistem yang diperoleh dari endpoint /stats:

Gambar 3.5 Hasil metrik performa (Throughput, Latency, dan Duplicate Rate) dari endpoint /stats.

Berdasarkan data pada Gambar 3.5 di atas, dapat diuraikan analisis sebagai berikut:

- **Throughput :** Sistem mencatatkan *throughput* rata-rata sebesar 145.37 event per detik (EPS). Angka ini merepresentasikan laju penulisan data efektif ke dalam basis data. Meskipun dibatasi oleh kecepatan I/O disk (Disk I/O *Bound*) pada container PostgreSQL, arsitektur Hybrid Asynchronous memastikan API tetap responsif menerima ribuan permintaan tanpa menolak koneksi.
- **Latency :** Rata-rata latensi tercatat sebesar 1.521,2 ms (±1,5 detik). Tingginya angka latensi ini dibandingkan waktu respons API adalah bukti bekerjanya mekanisme Backpressure. Latensi ini merupakan waktu tunggu di antrian (Queueing Delay) pada Redis. Data "diparkir" sementara di memori sebelum giliran ditulis ke DB, mencegah hilangnya data (data loss) saat lonjakan beban terjadi.
- **Akurasi Deduplikasi :** Tingkat duplikasi terukur (duplicate_rate_percent) adalah 30.13%. Angka ini sangat presisi mendekati konfigurasi target 30%. Hal ini memvalidasi bahwa logika Idempotency berbasis Unique Constraint pada tabel processed_events berhasil menyaring data ganda dengan akurasi tinggi meskipun diakses secara konkuren.

3.6 Pengujian Persistensi Data

Pengujian terakhir dilakukan dengan mematikan container (docker compose down) dan menyalakannya kembali.

Gambar 3.6 Statistik setelah container di-restart

Seperti terlihat pada **Gambar 3.6**, jumlah data (database_total_rows) tidak kembali ke nol setelah *restart*. Hal ini memverifikasi bahwa konfigurasi *Docker Named Volumes* telah berfungsi dengan benar, menjamin durabilitas data sesuai prinsip ACID (*Durability*).

BAB IV
KETERKAITAN

Implementasi sistem ini merefleksikan konsep-konsep inti dari buku Distributed Systems: Concepts and Design (Coulouris et al., 2011), khususnya pada bab-bab berikut:

4.1 Bab 1 & 2 (Karakteristik & Model)

Sistem mengadopsi model interaksi asinkron untuk mencapai *decoupling* antara pengirim dan penerima, karakteristik utama arsitektur terdistribusi modern untuk skalabilitas (Coulouris et al., 2011).

4.2 Bab 4 (Komunikasi Antar Proses)

Penggunaan API berbasis HTTP merepresentasikan komunikasi pesan diskrit yang mengabstraksi kompleksitas soket level rendah, memungkinkan interoperabilitas antar layanan (Coulouris et al., 2011).

4.3 Bab 6 (Indirect Communication)

Pola *Publish-Subscribe* diterapkan untuk memisahkan produsen data dari konsumennya, memungkinkan pengembangan sistem yang modular dan dapat diperluas (*extensible*) (Coulouris et al., 2011).

4.4 Bab 8 & 9 (Transaksi & Kontrol Konkurensi)

Fokus utama ada pada implementasi. Penerapan properti ACID (*Atomicity, Consistency, Isolation, Durability*) melalui transaksi basis data menjamin validitas data. Penggunaan mekanisme *ignore on conflict* adalah bentuk optimasi kontrol konkurensi untuk menghindari *overhead* penguncian global (*locking*) yang dapat menurunkan kinerja sistem (Coulouris et al., 2011).

4.5 Bab 10 (Sistem Terdistribusi Berkas/Penyimpanan)

Penggunaan volume persisten Docker mencerminkan kebutuhan akan penyimpanan yang stabil dan independen dari siklus hidup proses (*stateless process, stateful storage*) (Coulouris et al., 2011).

Daftar Pustaka

Coulouris, G., Dollimore, J., Kindberg, T., & Blair, G. (2011). *Distributed systems: Concepts and design* (5th ed.). Addison-Wesley.

Lampiran

Link Github : https://github.com/Wiraproject/uas-aggregator.git

Link Youtube : https://youtu.be/0Vthak9lGCE?si=eCJECqlepAxaRVwd