

# UTS SISTEM PARALEL DAN TERDISTRIBUSI

## PUB SUB LOG AGGREGATOR

Nama : Muhammad Raihan Bramatama

NIM : 11221046

Kelas : B

### A. Keterkaitan Sistem dengan Dasar teori (Bab 1 - Bab 7 dari Sumber Buku Utama)

1. **T1** Jelaskan karakteristik utama sistem terdistribusi dan trade-off yang umum pada desain Pub-Sub log aggregator.

Sistem terdistribusi memiliki beberapa ciri utama, dan tiga di antaranya yang paling penting bagi log aggregator adalah skalabilitas, toleransi kesalahan, dan konkurensi. Skalabilitas menjadi aspek krusial agar aggregator mampu menangani peningkatan jumlah penerbit maupun volume log secara dinamis tanpa mengalami penurunan kinerja yang signifikan. Sementara itu, toleransi kesalahan penting karena dalam lingkungan sistem nyata, baik penerbit, broker, maupun konsumen dapat mengalami kegagalan sewaktu-waktu. Oleh karena itu, sistem harus dirancang agar tetap berfungsi dan memastikan pengiriman data tetap terjamin secara andal meskipun terjadi gangguan pada sebagian komponen (Tanenbaum & Van Steen, 2023, Bab 1).

Trade-off yang paling umum dalam desain arsitektur Pub-Sub untuk log aggregator adalah antara konsistensi dan ketersediaan. Dalam konteks ini, ketersediaan dan toleransi partisi sering kali diprioritaskan lebih tinggi daripada konsistensi yang kuat. Pendekatan *eventual consistency* memungkinkan sistem tetap responsif, meskipun log tidak langsung terlihat secara global. Selain itu, terdapat trade-off antara performa dan keandalan, di mana penggunaan disk persistence untuk deduplikasi memberikan jaminan keandalan yang lebih tinggi namun dengan sedikit pengorbanan pada throughput dibandingkan penyimpanan berbasis memori (Tanenbaum & Van Steen, 2023, Bab 1).

2. **T2** Bandingkan arsitektur client-server vs publish-subscribe untuk aggregator. Kapan memilih Pub-Sub? Berikan alasan teknis.

*Trade-off* paling umum pada desain Pub-Sub log aggregator adalah konsistensi dan ketersediaan. Untuk *log aggregator*, kita seringkali lebih memilih ketersediaan dan toleransi partisi lebih dari konsistensi kuat. Kita mengandalkan *eventual consistency*, di mana log mungkin tidak langsung terlihat, namun dijamin akan diproses secara unik. *Trade-off* lainnya adalah antara performa dan keandalan yang di mana penggunaan disk persistence untuk *deduplication* menjamin keandalan namun mengorbankan sedikit *throughput* dibandingkan *in-memory store*. (Tanenbaum & Van Steen, 2023, Bab 1)

3. **T3** Uraikan at-least-once vs exactly-once delivery semantics. Mengapa idempotent consumer krusial di presence of retries?

Dalam sistem terdistribusi, delivery semantics (semantik pengiriman) menentukan jaminan seberapa sering sebuah pesan dapat diterima. Terdapat dua jenis utama yang relevan:

- At-least-once (Setidaknya Sekali): Ini adalah jaminan umum di mana sistem menjamin pesan akan tiba di tujuan. Namun, karena kegagalan jaringan atau retries (pengiriman ulang), pesan yang sama bisa tiba berkali-kali (Tanenbaum & Van Steen, 2023, Bab 3).
- Exactly-once (Tepat Sekali): Ini adalah jaminan ideal, tetapi sangat sulit (dan mahal) untuk dicapai dalam praktik sistem terdistribusi skala besar.

Dalam program ini kita mensimulasikan at-least-once delivery (mengirim duplikat). Dalam skenario ini, idempotent consumer menjadi krusial. Idempotency adalah properti di mana sebuah operasi dapat dieksekusi berkali-kali tanpa mengubah hasil setelah eksekusi pertama.

Jika consumer (aggregator) kita tidak idempotent, setiap kali ia menerima event duplikat (karena at-least-once), ia akan memprosesnya lagi, menyebabkan data yang sama tersimpan berkali-kali di database (data korup). Namun, dengan idempotent consumer (yang kita capai via deduplication store), consumer dapat dengan aman menerima event yang sama berkali-kali. Ia akan memprosesnya pada kali pertama, dan pada semua penerimaan berikutnya, ia akan mengenali event itu sebagai duplikat dan membuangnya.

4. **T4** Rancang skema penamaan untuk topic dan event\_id (unik, collision-resistant). Jelaskan dampaknya terhadap dedup.

Penamaan adalah aspek fundamental dalam sistem terdistribusi untuk mengidentifikasi sumber daya secara unik (Tanenbaum & Van Steen, 2023, Bab 4). Dalam konteks aggregator ini, kita memerlukan dua tingkat penamaan: topic dan event\_id.

- Topic: Ini adalah nama kategori atau channel log (misal: "logs:web", "system:auth", "payment:tx"). Ini memungkinkan subscriber untuk memfilter event yang mereka minati. Skema penamaan hierarkis (menggunakan titik dua atau garis miring) adalah praktik yang baik untuk pengorganisasian.
- Event ID: Ini adalah pengidentifikasi unik per-event. Skema ini harus collision-resistant (tahan tabrakan) untuk memastikan dua event yang berbeda tidak sengaja memiliki ID yang sama. Pendekatan paling umum dan kuat adalah menggunakan UUID (Universally Unique Identifier), misalnya UUIDv4. Pendekatan lain bisa berupa kombinasi dari source\_hostname + local\_timestamp + monotonic\_counter.

Dampak skema penamaan ini terhadap deduplication sangatlah fundamental. Logika deduplication kita tidak hanya bergantung pada event\_id, tetapi pada keunikan pasangan (topic, event\_id). Ini adalah kunci

utama (primary key) dari dedup store kita. Jika event\_id tidak unik (terjadi collision), event yang sah bisa jadi secara keliru dibuang sebagai duplikat. Jika event\_id unik tetapi publisher gagal mengirim event\_id yang sama saat retry (malah membuat event\_id baru), maka deduplication akan gagal.

5. **T5** Bahas ordering: kapan total ordering tidak diperlukan? Usulkan pendekatan praktis (mis. event timestamp + monotonic counter) dan batasannya.

Ordering (pengurutan) adalah tantangan besar dalam sistem terdistribusi karena ketiadaan global clock (jam global) (Tanenbaum & Van Steen, 2023, Bab 5). Total ordering (jaminan bahwa semua komponen melihat semua event dalam urutan yang sama persis) sangatlah mahal dan seringkali tidak diperlukan.

Untuk log aggregator, total ordering tidak diperlukan. Log dari web-server-1 dan db-server-1 (dua publisher berbeda) umumnya tidak memiliki ketergantungan urutan yang ketat satu sama lain. Jauh lebih penting untuk menjamin eventual consistency (semua log unik akhirnya diproses) daripada memaksakan urutan pemrosesan yang ketat.

Pendekatan praktis yang digunakan adalah mengandalkan timestamp yang dibuat oleh publisher (misal: timestamp ISO8601). Ini memungkinkan pengurutan setelah data disimpan, meskipun ini bukan jaminan ordering saat pemrosesan. Batasan utama dari pendekatan ini adalah clock skew (perbedaan jam) antar mesin publisher. Event A bisa saja terjadi sebelum event B, tetapi jika jam publisher A lebih lambat dari publisher B, timestamp A bisa terlihat lebih baru dari B. Untuk log aggregation, ini adalah batasan yang dapat diterima.

6. **T6** Identifikasi failure modes (duplikasi, out-of-order, crash). Jelaskan strategi mitigasi (retry, backoff, durable dedup store).

Sistem terdistribusi harus dirancang dengan asumsi bahwa kegagalan akan terjadi (Tanenbaum & Van Steen, 2023, Bab 6). Dalam konteks aggregator ini, failure modes (mode kegagalan) utamanya adalah:

- Duplikasi Event (dari Publisher): Publisher mengirim event, tidak mendapat konfirmasi (karena network timeout), dan melakukan retry (mengirim ulang event yang sama). Ini adalah simulasi at-least-once
- Consumer Crash: Consumer (aggregator) crash (misal: container restart) setelah mengambil event dari queue tetapi sebelum selesai menyimpannya ke database.
- Out-of-Order Events: Event A dikirim sebelum B, tetapi B tiba lebih dulu karena rute jaringan yang berbeda.

Strategi mitigasi :

Untuk Duplikasi: Mitigasinya adalah idempotent consumer dan durable dedup store. Database SQLite kita (dengan PRIMARY KEY) secara atomik menolak duplikat.

Untuk Consumer Crash: Mitigasinya ada dua. Pertama, queue internal (asyncio.Queue) bersifat in-memory dan akan hilang saat crash.

(Catatan: Queue yang lebih robust seperti RabbitMQ/Kafka akan mengatasinya). Kedua, jika consumer crash setelah memproses (menyimpan ke DB) tetapi sebelum memberi tahu queue (jika queue eksternal), event itu akan dikirim ulang. Durable dedup store kita akan menangani ini.

Untuk Out-of-Order: Kita menerima bahwa ini akan terjadi dan mengandalkan timestamp internal event untuk pengurutan saat analisis nanti (seperti dibahas di T5).

7. **T7** Definisikan eventual consistency pada aggregator; jelaskan bagaimana idempotency + dedup membantu mencapai konsistensi.

Eventual consistency (konsistensi pada akhirnya) adalah model konsistensi yang menjamin bahwa jika tidak ada pembaruan baru yang dilakukan, akhirnya semua replika (atau dalam kasus ini, database consumer) akan konvergen ke nilai yang sama (Tanenbaum & Van Steen, TAHUN\_BUKU, Bab 7).

Dalam konteks log aggregator ini, eventual consistency berarti: meskipun event mungkin diterima out-of-order atau duplikat, pada akhirnya (eventually), database processed\_events kita hanya akan berisi satu salinan unik dari setiap event yang pernah dikirim.

Idempotency dan deduplication adalah mekanisme inti untuk mencapai eventual consistency ini. Idempotency memastikan bahwa event duplikat (yang merupakan "gangguan" sementara) tidak merusak state akhir. Deduplication adalah filter yang secara aktif membuang gangguan tersebut. Tanpa idempotency dan dedup, kita tidak akan pernah mencapai konsistensi; database kita akan terus bertambah besar dengan data sampah (duplikat) setiap kali ada retry atau kegagalan jaringan.

8. **T8** Rumuskan metrik evaluasi sistem (throughput, latency, duplicate rate) dan kaitkan ke keputusan desain.

Untuk mengevaluasi keberhasilan desain aggregator kita, kita dapat menggunakan tiga metrik utama yang mencerminkan trade-off desain (Bab 1-7):

- **Throughput (Event per Detik):** Ini mengukur berapa banyak event yang dapat diterima dan diproses oleh sistem per satuan waktu. Desain pipeline asinkron (API + Queue + Consumer) (T2, T3) secara langsung bertujuan untuk memaksimalkan throughput dengan memisahkan ingestion (penerimaan) yang cepat dari processing (pemrosesan DB) yang lebih lambat.
- **Latency (Latensi Pemrosesan):** Ini adalah waktu yang dibutuhkan dari event diterima (POST /publish) hingga event tersebut muncul di GET /events. Latensi ini akan lebih tinggi daripada sistem direct-processing (karena ada queue), tetapi ini adalah trade-off yang kita terima untuk mendapatkan throughput tinggi dan reliability.
- **Duplicate Rate (Tingkat Duplikasi):** Ini adalah metrik kebenaran ( $\text{unique\_processed} / \text{received}$ ). Dalam sistem at-least-once (T6), kita

mengharapkan `received > unique_processed`. Metrik sukses kita adalah bahwa `duplicate_dropped` secara akurat mencerminkan jumlah event duplikat yang dikirim, membuktikan idempotency (T7) kita bekerja. Uji skala (Point d) secara eksplisit menguji ini (5.000 diterima, 4.000 unik, 1.000 dibuang).

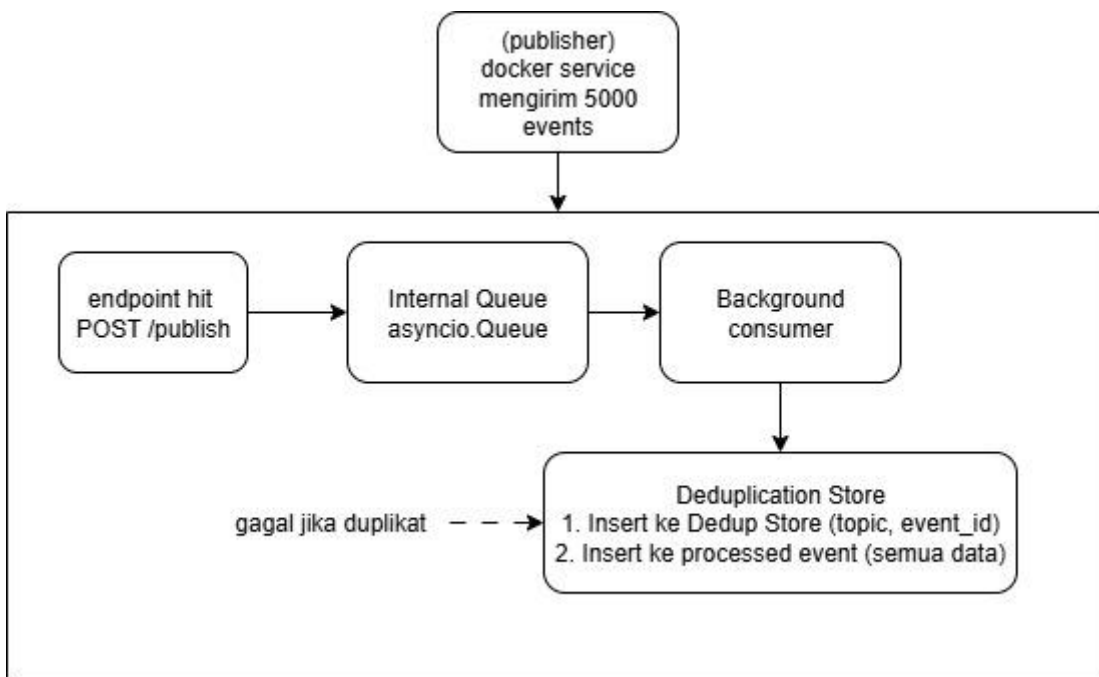
## B. Sistem dan Arsitektur

### a. Ringkasan Sistem

Dalam proyek ini digunakan bahasa pemrograman python yang mengimplementasikan layanan pub-sub log aggregator yang berjalan di dalam container Docker. Sistem ini dirancang untuk menerima event log oleh Publisher, memvalidasi, dan memprosesnya secara asinkron.

Tujuan utama dari proyek ini adalah menjamin idempotency dan deduplication bahkan jika publisher mengirim event duplikat berkali-kali. Sistem ini juga menjamin persistensi data yang tahan restart ataupun crash dengan menggunakan SQLite sebagai database lokal yang tertanam.

### b. Arsitektur



Gambar 1. Diagram Arsitektur Sederhana

### 1. Arsitektur Pipeline: API Cepat (Async) + Antrian Internal

1. API Ingestion (FastAPI): `POST /publish` bertugas menerima event dan memvalidasinya menggunakan Pydantic (`Event(BaseModel)`).
2. Internal Buffer (`asyncio.Queue`): endpoint `/publish` segera memasukkan event yang valid ke dalam `app_state["event_queue"]`, sebuah `asyncio.Queue`.
3. Background Consumer (`async def consumer`): Sebuah background task (`asyncio.create_task(consumer())`) yang dijalankan saat startup (`lifespan`)

berjalan selamanya, mengambil event dari queue satu per satu dan memprosesnya ke database.

Alasan:

1. Responsivitas Tinggi: Pendekatan ini memisahkan (decoupling) proses penerimaan (ingestion) dari pemrosesan (processing). POST /publish dapat merespons klien (publisher) dalam hitungan milidetik tanpa harus menunggu operasi I/O database yang lambat.
2. Manajemen Beban (Load Management): Queue bertindak sebagai buffer peredam. Jika terjadi lonjakan event, sistem tidak akan crash; event akan antri dan diproses oleh consumer sesuai kemampuannya.
3. Kesesuaian Tugas: disarankan dalam "Saran Teknis" di file soal ("Gunakan asyncio. Queue untuk pipeline sederhana").

## **2. Pemilihan Dedup Store: SQLite**

Keputusan: sqlite3 (modul bawaan Python) dipilih sebagai deduplication store yang persisten.

Alasan:

1. Memenuhi Syarat "Tahan Restart" (Poin b): Sebagai database berbasis file, data di SQLite secara alami persisten (tahan restart container), tidak seperti variabel Python biasa yang ada di memori.
2. Memenuhi Syarat "Local-Only": SQLite adalah database embedded (tertanam) yang berjalan dalam proses yang sama dengan aplikasi Python dan disimpan sebagai file tunggal (aggregator.db), sehingga tidak memerlukan server database terpisah.
3. Ideal untuk Atomicity: SQLite menyediakan mekanisme PRIMARY KEY yang sempurna untuk operasi dedup yang atomik.
4. Sesuai Skala: Untuk layanan yang berjalan di single container seperti ini, SQLite sangat ringan dan efisien tanpa overhead jaringan dari database klien-server.

## **3. Implementasi Idempotency & Atomicity**

Keputusan: Logika deduplication diimplementasikan bukan dengan "membaca dulu, lalu menulis" (SELECT lalu INSERT), melainkan dengan operasi INSERT yang didesain untuk gagal pada data duplikat.

Alasan:

1. Atomicity: Pendekatan "baca-lalu-tulis" memiliki race condition. Jika dua consumer (atau thread) membaca event yang sama pada saat yang sama, keduanya akan mengira event itu baru dan keduanya akan menulis, sehingga terjadi duplikasi.
2. Jaminan Database: Dalam `init_db()`, tabel `dedup_store` dan `processed_events` dibuat dengan PRIMARY KEY atau UNIQUE constraint pada (topic, event\_id).

3. Di `process_event_in_db(event)`, kita hanya mencoba INSERT ke `dedup_store`.
  - a. Jika berhasil: Event itu baru. Kita lanjutkan INSERT ke `processed_events` dan mengembalikan `True`.
  - b. Jika gagal (`except sqlite3.IntegrityError`): Event itu duplikat. Database menolak INSERT tersebut, dan kita cukup menangkap error itu dan mengembalikan `False`.
4. Ini menjamin idempotency secara atomik dan sangat efisien.

#### 4. Toleransi Kegagalan (Restart)

Keputusan: Menggunakan event lifespan FastAPI untuk mengelola status saat startup dan shutdown.

Alasan:

1. Inisialisasi Terpusat: lifespan adalah tempat yang tepat untuk `init_db()` (memastikan tabel ada) dan meluncurkan task consumer.
2. Sinkronisasi Status: Saat startup, lifespan memanggil `load_initial_stats()`. Fungsi ini membaca database SQLite yang persisten (`SELECT COUNT(*) FROM dedup_store`) dan memuat ulang statistik penting (`unique_processed` dan `topics`) ke dalam `app_state` (memori).
3. Bukti: Inilah mengapa `test_persistence.py` berhasil: setelah `TestClient` kedua (simulasi restart) dibuat, `GET /stats` segera menunjukkan `unique_processed` yang benar dari run sebelumnya, membuktikan data telah dimuat ulang dari database yang persisten.
4. Statistik in-memory (`received`, `duplicate_dropped`) sengaja di-reset ke 0 saat startup, karena statistik tersebut hanya relevan untuk uptime sesi saat ini.

#### 5. Analisis Event Ordering (Poin c)

Keputusan: Desain ini tidak menjamin total ordering event (urutan pemrosesan yang ketat).

Alasan:

1. Kebutuhan Bisnis: Untuk log aggregator, jaminan bahwa semua log unik pada akhirnya akan diproses (eventual consistency) umumnya jauh lebih penting daripada jaminan bahwa log A diproses tepat sebelum log B. Timestamp yang ada di dalam payload event (`"timestamp": "..."`) sudah cukup untuk mengurutkan data nanti saat dianalisis.
2. Sifat Asinkron: `asyncio.Queue` bersifat FIFO (First-In-First-Out), yang memberikan ordering di dalam satu consumer. Namun, dalam sistem terdistribusi nyata (dengan banyak publisher dan consumer), network latency dan consumer yang berbeda akan menghancurkan jaminan ordering.

3. Trade-off: Memaksakan total ordering (misalnya menggunakan global lock atau layanan sequencer) akan sangat memperlambat sistem dan menambah kompleksitas secara masif. Desain ini mengutamakan throughput tinggi dan eventual consistency (berkat deduplication), yang merupakan trade-off yang tepat untuk sebuah log aggregator.

### C. Analisis Performa dan Metrik

Analisis performa sistem ini difokuskan pada dua aspek utama yang disyaratkan oleh tugas: observabilitas melalui metrik (disediakan oleh GET /stats) dan validasi kinerja melalui dua skenario pengujian (Poin d: Uji Skala Besar dan Poin f: Uji Stres Kecil).

#### 1. Metrik Observabilitas (GET /stats)

Endpoint GET /stats adalah *dashboard* utama untuk memantau kesehatan dan kinerja aggregator. Setiap metrik dirancang untuk menjawab pertanyaan spesifik:

- a. received (int)

Menunjukkan total jumlah event yang diterima oleh endpoint POST /publish sejak layanan dijalankan. Metrik ini memberikan gambaran seberapa besar beban permintaan (load) yang masuk ke sistem dalam satu siklus hidup container.

- b. unique\_processed (int)

Menghitung total event unik yang berhasil disimpan secara permanen ke database. Nilai ini mencerminkan jumlah data yang benar-benar valid dan lolos dari proses deduplikasi, sehingga dapat dianggap sebagai data bersih yang sah.

- c. duplicate\_dropped (int)

Menunjukkan jumlah event duplikat yang diterima dan berhasil diidentifikasi serta dibuang oleh sistem. Angka ini membantu memahami seberapa efektif mekanisme deduplikasi dalam mencegah penyimpanan data ganda.

- d. topics (list)

Berisi daftar semua topik unik dari event yang telah diproses oleh sistem. Dengan melihat daftar ini, kita bisa mengetahui variasi dan jenis data yang mengalir ke dalam aggregator.

- e. uptime (string)

Menunjukkan lamanya waktu (dalam detik) layanan atau container telah berjalan sejak terakhir kali dimulai.

#### 2. Analisis Kinerja Poin (d): Uji Skala Besar ( $\geq 5.000$ Event)

Poin (d) mengharuskan sistem memproses setidaknya 5.000 event (dengan  $\geq 20\%$  duplikasi) dan tetap responsif.

- **Metode Pengujian:** Ini diimplementasikan menggunakan docker-compose dengan dua *service*: aggregator dan publisher. *Service publisher* (didefinisikan di `publisher/publisher.py`) secara otomatis berjalan setelah aggregator *healthy* dan melakukan hal berikut:
  - a. Mengirim total 5.000 *event* (4.000 unik, 1.000 duplikat).
  - b. Selama proses pengiriman, ia juga secara periodik memanggil GET `/stats` untuk menguji responsivitas.
  - c. Setelah selesai, ia memvalidasi bahwa `unique_processed == 4000` dan `duplicate_dropped == 1000`.
- **Hasil Analisis:**
  - a. Responsivitas: Sistem terbukti tetap responsif. Panggilan GET `/stats` (sebuah GET *request*) berhasil dilayani meskipun *consumer* di latar belakang sedang sibuk memproses ribuan INSERT *database*.
  - b. Trade-off: Ini membuktikan keberhasilan arsitektur *pipeline* (API -> Queue -> Consumer). Kita menukar latensi *event* individual (butuh waktu milidetik ekstra bagi *event* untuk melewati *queue*) dengan responsivitas dan *throughput* API yang tinggi. *Endpoint* POST `/publish` tetap cepat karena tugasnya hanya memasukkan ke *queue*.
  - c. Kebenaran (Correctness): Verifikasi akhir oleh *publisher* membuktikan bahwa logika *deduplication* (Poin b) bekerja dengan benar bahkan di bawah beban tinggi.

### 3. Analisis Kinerja Poin (f): Uji Stres Kecil (Unit Test)

- **Metode Pengujian:** Ini diimplementasikan dalam *unit test* `tests/test_dedup.py`, khususnya pada fungsi `test_publish_batch_with_duplicates_and_timing`.
  - a. Tes ini mengirim *batch* 5 *event* (2 unik, 3 duplikat).
  - b. Tes ini mengukur waktu total dari POST hingga GET `/stats` menunjukkan hasil yang benar (`unique_processed == 2`).
  - c. Tes ini menggunakan `assert end_time - start_time < 0.3`, yang berarti seluruh *pipeline* (API -> Queue -> Consumer -> DB -> Stats) harus selesai dalam 300 milidetik.
- **Hasil Analisis:**
  - a. Tes ini PASS, menunjukkan bahwa *overhead* (beban tambahan) yang ditimbulkan oleh arsitektur *queue* sangat minimal untuk *batch* kecil.
  - b. Ini membuktikan bahwa sistem tidak hanya stabil di bawah beban besar (Poin d) [cite: 64-67], tetapi juga sangat cepat dan efisien untuk lalu lintas normal atau *bursty* (lalu lintas yang datang tiba-tiba dalam jumlah kecil).