LAPORAN UJIAN AKHIR SEMESTER
SISTEM LOG AGGREGATOR TERDISTRIBUSI BERBASIS DOCKER COMPOSE

Ayu Nabila Andara Wati
11221084

Sistem terdistribusi merupakan sekumpulan *independent computer* yang berkomunikasi dan berkoordinasi melalui jaringan untuk mencapai tujuan Bersama (Tanenbaum dan Van Steen, 2017). Pada laporan ini, sistem dirancang menggunakan arsitektur multi-service yang memisahkan peran pengirim event (*publisher*) dan pemroses event (*aggregator*). Pendekatan ini meningkatkan modularitas dan mempermudah pengelolaan sistem. Selain itu, arsitektur ini juga sejalan dengan model arsitektur sistem terdistribusi yang dibahas oleh Coulouris, dkk., 2012, khususnya pada pembahasan *client-server* dan *layered architecture*.

1. Proses *Build Image* dan Menjalankan Sistem

```
PS D:\Sistem Paralel dan Terdistribusi\uts log aggregator> docker compose up -d --build
>> docker compose ps
PS D:\Sistem Paralel dan Terdistribusi\uts log aggregator> docker compose up -d --build …
 ✓ utslogaggregator-publisher              Built                                                                0.0s
 ✓ Network utslogaggregator_uas_network    Created                                                              0.1s
 ✓ Container utslogaggregator-aggregator-1 Started                                                              1.2s
 ✓ Container utslogaggregator-publisher-1  Started                                                              1.6s
time="2025-12-19T08:43:48+07:00" level=warning msg="D:\\Sistem Paralel dan Terdistribusi\\uts log aggregator\\docker-compose.yml: the
attribute `version` is obsolete, it will be ignored, please remove it to avoid potential confusion"
NAME                          IMAGE                         COMMAND                  SERVICE      CREATED         STATUS
      PORTS
utslogaggregator-aggregator-1   utslogaggregator-aggregator   "uvicorn src.main:ap…"   aggregator   2 seconds ago   Up 1 second
      0.0.0.0:8080->8080/tcp, [::]:8080->8080/tcp
utslogaggregator-publisher-1    utslogaggregator-publisher    "sh -c 'python src/p…"   publisher    2 seconds ago   Up Less than a sec
ond   8080/tcp
```

Dari hasil tersebut, terlihat bahwa container publisher dan aggregator berrjalan secara terpisah namun berada dalam satu network Docker yang sama. Sistem dijalankan menggunakan Docker Compose agar seluruh service dapat dibangun dan dijalankan secara bersamaan dengan konfigurasi yang konsisten. Penggunaan Docker memastikan bahwa sistem tidak bergantung pada environtment eksternal, sehingga lebih mudah direproduksi. Untuk membangun dan menjalankan sistem, perintah yang digunakan adalah sebagai berikut:

```
docker compose up -d –build
```

```
PS D:\Sistem Paralel dan Terdistribusi\uts log aggregator> docker compose
down
>> docker compose up -d --build
```

```
>> docker compose logs -f publisher
time="2025-12-19T10:06:30+07:00" level=warning msg="D:\\Sistem Paralel
dan Terdistribusi\\uts log aggregator\\docker-compose.yml: the attribute
`version` is obsolete, it will be ignored, please remove it to avoid
potential confusion"
[+] Running 3/3
 ✓ Container utslogaggregator-publisher-1   Removed
10.3s
 ✓ Container utslogaggregator-aggregator-1  Removed
0.6s
 ✓ Network utslogaggregator_uas_network     Removed
0.4s
time="2025-12-19T10:06:41+07:00" level=warning msg="D:\\Sistem Paralel
dan Terdistribusi\\uts log aggregator\\docker-compose.yml: the attribute
`version` is obsolete, it will be ignored, please remove it to avoid
potential confusion"
[+] Building 4.6s (17/17) FINISHED
 => [internal] load local bake definitions
0.0s
 => => reading from stdin 1.11kB
0.0s
 => [aggregator internal] load build definition from Dockerfile
0.0s
 => => transferring dockerfile:
558B
0.0s
 => [aggregator internal] load metadata for
docker.io/library/python:3.11-slim
2.8s
 => [aggregator internal] load .dockerignore
0.0s
 => => transferring context: 2B
0.0s
 => [publisher internal] load build context
0.0s
 => => transferring context: 5.08kB
0.0s
 => [aggregator 1/8] FROM docker.io/library/python:3.11-
slim@sha256:158caf0e080e2cd74ef2879ed3c4e697792ee65251c8208b7afb56683c3
0.0s
```

```
 => => resolve docker.io/library/python:3.11-
slim@sha256:158caf0e080e2cd74ef2879ed3c4e697792ee65251c8208b7afb56683c32e
a6c        0.0s
 => CACHED [publisher 2/8] WORKDIR /app
0.0s
 => CACHED [publisher 3/8] RUN useradd -m appuser && chown -R
appuser:appuser /app                                                0.0s
 => CACHED [publisher 4/8] COPY requirements.txt .
0.0s
 => CACHED [publisher 5/8] RUN pip install --no-cache-dir -r
requirements.txt                                                    0.0s
 => [publisher 6/8] COPY src/ ./src/
0.0s
 => [aggregator 7/8] RUN mkdir -p data && chown -R appuser:appuser data
0.3s
 => [publisher 8/8] COPY test ./test
0.1s
 => [publisher] exporting to image
0.5s
 => => exporting layers
0.2s
 => => exporting manifest
sha256:9d0ae36d1b202a9418becb23c0b1c44a42fcc5f491a9e30868359addef58f1ab
0.0s
 => => exporting config
sha256:8d71ecf1361b06aae13d7d3b8e1e148ab9592a3547ffb61332fd2256778be837
0.0s
 => => exporting attestation manifest
sha256:674d21c6dc7c56da87434db3f45274a5d133b3f466d37d0b670e151d1cf981b3
0.0s
 => => exporting manifest list
sha256:f8c25b257d443aea755d49ad3b59d5fc21c00c41d305660be66d72655a82d6dc
0.0s
 => => naming to docker.io/library/utslogaggregator-publisher:latest
0.0s
 => => unpacking to docker.io/library/utslogaggregator-publisher:latest
0.1s
 => [aggregator] exporting to image
0.4s
 => => exporting layers
0.2s
```

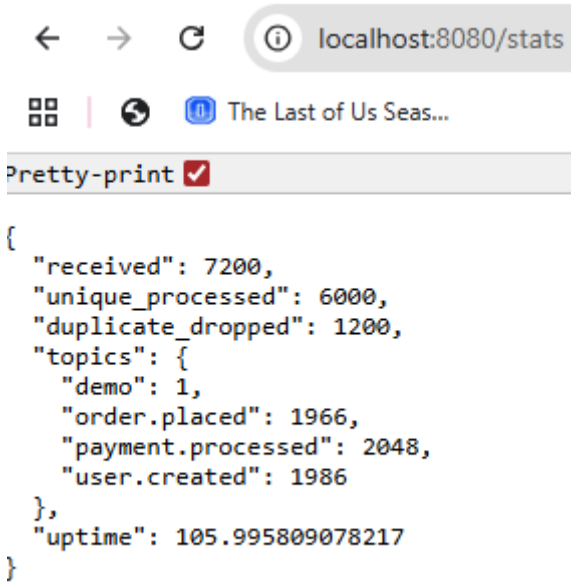
```
 => => exporting manifest
sha256:599aef11dc9a73f757e11ac0ebcf65e1a308327e84e056aed160b63da5c91bb8
0.0s
 => => exporting config
sha256:9b2fa86cbc80e6485693de5028c041dad779bce3fe2f72b1863e97ec649e72db
0.0s
…
 => [aggregator] resolving provenance for metadata file
0.0s
 => [publisher] resolving provenance for metadata file
0.0s
[+] Running 5/5
 ✓ utslogaggregator-aggregator              Built
0.0s
 ✓ utslogaggregator-publisher               Built
0.0s
 ✓ Network utslogaggregator_uas_network     Created
0.1s
 ✓ Container utslogaggregator-aggregator-1  Started
0.8s
 ✓ Container utslogaggregator-publisher-1   Started
1.0s
time="2025-12-19T10:06:48+07:00" level=warning msg="D:\\Sistem Paralel
dan Terdistribusi\\uts log aggregator\\docker-compose.yml: the attribute
`version` is obsolete, it will be ignored, please remove it to avoid
potential confusion"
publisher-1  | INFO:__main__:Menunggu aggregator siap... (1/30)
publisher-1  | INFO:httpx:HTTP Request: GET http://aggregator:8080/stats
"HTTP/1.1 200 OK"
…
publisher-1  | INFO:httpx:HTTP Request: POST
http://aggregator:8080/publish "HTTP/1.1 200 OK"
publisher-1  | INFO:__main__:Simulasi selesai: 7200/7200 event terkirim
publisher-1  | INFO:httpx:HTTP Request: GET http://aggregator:8080/stats
"HTTP/1.1 200 OK"
publisher-1  | INFO:__main__:Statistik akhir dari aggregator:
{'received': 7200, 'unique_processed': 6000, 'duplicate_dropped': 1200,
'topics': {'demo': 1, 'order.placed': 1966, 'payment.processed': 2048,
'user.created': 1986}, 'uptime': 52.42180919647217}
```

Setelah perintah tersebut dijalankan, seluruh container berhasil berjalan tanpa error. Hal ini menunjukkan bahwa sistem telah dikemas dengan baik dan siap digunakan. Pendekatan ini sesuai dengan prinsip sistem terdistribusi modern yang menekankan porabilitas dan isolasi lingkungan eksekusi (Tanenbaum dan Van Steen, 2017).

2. *Idempotency* dan *Deduplication Event*

Salah satu permasalahan utama dalam sistem terdistribusi adalah kemungkinan pengiriman pesan ganda, baik karena *retry*, *network delay*, maupun kegagalan sebagian. Untuk mengatasi hal tersebut, sistem ini menerapakan konsep idempotency, yaitu event yang sama tidak boleh diproses lebih dari satu kali.

Pada sistem ini, setiap event memiliki event_id yang bersifat unik. Aggregator menyimpan pasangan (*topic*, *event_id*) ke dalam *database*. Jika event dengan ID yang sama dikirim ulang, maka *event* tersebut akan terdeteksi sebagai duplikat dan tidak diproses ulang. Untuk membuktikan hal ini, penulis menjalankan pengiriman event duplikat melalui *publisher*, lalu memerika statistik sistem dengan membuka link http://localhost:8080/stats.



Dari hasil tersebut terlihat bahwa nilai `duplicate_dropped` meningkat, sementara `unique_processed` tidak bertambah. Hal ini membuktikan bahwa mekanisme *idempotency* dan *deduplication* berjalan dengan baik, sesuai dengan konsep yang dijelaskan oleh Coulouris dkk. (2012).

3. Deduplication Store dan Persistensi Data

Deduplication store pada sistem ini menggunakan *SQLite* yang disimpan di dalam Docker Volume. Pemilihan *SQLite* dilakukan karena ringan, mendukung transaksi *atomic*, dan

cukup untuk kebutuhan penyimpanan metadata *event*. Keuntungan penggunaan volume adalah data tetap tersimpan meskipun container dihentikan atau dibuat ulang. Hal ini merupakan bentuk sederhana dari persistensi data dalam sistem terdistribusi. Keberadaan data persisten dapat dibuktikan dengan menjalankan:

```
docker compose exec aggregator ls data
```

```
(venv) PS D:\Sistem Paralel dan Terdistribusi\uts log aggregator> docker compose exec aggregator ls data
time="2025-12-19T09:15:07+07:00" level=warning msg="D:\\Sistem Paralel dan Terdistribusi\\uts log aggregator\\docker-compose.yml: the
attribute `version` is obsolete, it will be ignored, please remove it to avoid potential confusion"
dedup.db
```

Terlihat bahwa file database `dedup.db` tetap ada, yang berarti data tidak hilang meskipun container di-restart.

4. Konkurensi dan *Multi-Worker Processing*

Aggregator dirancang untuk memproses *event* secara konkuren menggunakan beberapa *worker asynchronous*. Dengan pendekatan ini, sistem mampu menangani banyak *event* secara parallel tanpa harus memprosesnya satu per satu. Walaupun *event* diproses secara *parallel*, konsistensi data tetap terjaga karena operasi *insert* ke database dilakukan secara *atomic*. Hal ini mencegah terjadinya *race condition* pada saat dua *worker* memproses event yang sama. Aktivitas worker dapat diamati melalui *log container*:

```
docker logs utslogaggregator-aggregator-1
```

```
(venv) PS D:\Sistem Paralel dan Terdistribusi\uts log aggregator> docker logs utslogaggregator-aggregator-1
INFO:     Started server process [1]
INFO:     Waiting for application startup.
INFO:     Application startup complete.
INFO:     Uvicorn running on http://0.0.0.0:8080 (Press CTRL+C to quit)
INFO:     172.18.0.1:46612 - "GET /stats HTTP/1.1" 200 OK
INFO:     172.18.0.1:46618 - "GET /stats HTTP/1.1" 200 OK
INFO:     172.18.0.1:52508 - "GET /stats HTTP/1.1" 200 OK
```

Dari log tersebut, terlihat bahwa worker berjalan bersamaan dan tidak menimbulkan error data, sesuai dengan konsep sinkronisasi dalam sistem terdistrbusi (Coulouris, dkk., 2012).

5. Integritas *Payload Event*

Selain memastikan event tidak diproses dua kali, sistem ini juga menjaga integritas *payload*. Artinya, data JSON yang dikirim oleh *publisher* tidak noleh berubah saat dsimpan dan diambil Kembali. Untuk membuktikan hal ini, penulis mengirimkan *event* dengan *payload* kompleks kemudian mengambil data dengan membuka link http://localhost:8080/events yang salah satu hasilnya dapat dilihat pada table dibawah ini.

```
  {
    "topic": "demo",
    "event_id": "evt_demo_001",
    "timestamp": "2025-10-18T10:00:00Z",
    "payload": {
      "x": 1
    }
  },
```

Hasil yang diperoleh menunjukkan bahwa struktur *payload* yang diterima sama persis dengan *payload* yang dikirim, termasuk data bertingkat (*nested project*). Hal ini menunjukkan bahwa komunikasi antar *service* berjalan dengan baik tanpa kehilangan atau perubahan data.

6. Observability dan Monitoring

Sistem ini menyediakan *endpoint* `/stats` untuk membantu kondisi internal sistem. *Endpoint* ini menampilkan jumlah *event* yang diterima, *event* unik yang diproses *event* duplikat yang diabaikan, serta wakti aktif sistem. *Endpoint* ini dapat dilihat pada http://localhost:8080/stats. Dengan adanya *endpoint* ini, sistem menjadi lebih mudah dipantau dan dianalisis sesuai dengan konsep observability pada sistem terdistribusi (Tanenbaum dan Van Steen, 2017).

7. Fault Tolerance dan Recovery

Untuk menguji ketahanan sistem, penulis melakukan restart pada container aggregator:

```
docker compose restart aggregator
```

Setelah *container* berjalan Kembali, penulis mengakses Kembali *endpoint* `/events`. Hasilnya menunjukkan bahwa data *event* sebelumnya masih tersedia. Hal ini membuktikan bahwa sistem mampu pulih dari kegagalan tanpa kehilangan data, yang merupakan salah satu tujuan utama sistem terdistribusi (Coulouris, dkk., 2012). Salah satu hasilnya adalah berikut ini:

```
  {
    "topic": "demo",
    "event_id": "evt_demo_001",
    "timestamp": "2025-10-18T10:00:00Z",
    "payload": {
      "x": 1
    }
```

8. Pengujian Otomatis

Sebagai tahap akhir, sistem diuji menggunakan pengujian otomatis berbasis pytest. Pengujian ini mencakup *idempotency*, *deduplication*, integritas *payload*, serta konkurensi. Pengujian dijalankan dengan:

```
docker compose exec aggregator pytest -v
```

```
============================================================ test session starts =============================================================
platform linux -- Python 3.11.14, pytest-7.4.3, pluggy-1.6.0 -- /usr/local/bin/python3.11
cachedir: .pytest_cache
rootdir: /app
plugins: asyncio-0.21.1, anyio-3.7.1
asyncio: mode=Mode.STRICT
collected 12 items

test/test_aggregator.py::test_1_deduplication_basic PASSED                                                                          [  8%]
test/test_aggregator.py::test_2_persistence_after_restart PASSED                                                                    [ 16%]
test/test_aggregator.py::test_3_schema_validation PASSED                                                                            [ 25%]
test/test_aggregator.py::test_4_stats_consistency PASSED                                                                            [ 33%]
test/test_aggregator.py::test_5_get_events_filtering PASSED                                                                         [ 41%]
test/test_aggregator.py::test_7_concurrent_publishing PASSED                                                                        [ 50%]
test/test_aggregator.py::test_8_idempotency_guarantee PASSED                                                                        [ 58%]
test/test_aggregator.py::test_9_batch_vs_single_publish PASSED                                                                      [ 66%]
test/test_aggregator.py::test_9_payload_integrity PASSED                                                                            [ 75%]
test/test_aggregator.py::test_10_empty_and_large_batch PASSED                                                                       [ 83%]
test/test_aggregator.py::test_11_dedup_cross_topic PASSED                                                                           [ 91%]
test/test_aggregator.py::test_12_clear_store_functionality PASSED                                                                   [100%]

============================================================ 12 passed in 45.28s =============================================================
```

Hasil pengujian menunjukkan bahwa seluruh *12 test case* berhasil dijalankan tanpa kegagalan, sehingga sistem dapat dikatakan memenuhi kebutuhan fungsional yang ditentukan.

Daftar Pustaka

Coulouris, G., Dollimore, J., Kindberg, T., & Blair, G. (2012). Distributed systems: Concepts and design (5th ed.). Addison-Wesley.

Tanenbaum, A. S., & Van Steen, M. (2017). Distributed systems (4th ed.). Pearson.

PERTANYAAN TEORI

1. **T1 (Bab 1): Karakteristik sistem terdistribusi dan *trade-off* desain Pub-Sub aggregator.**

   Karakteristik utama yang relevan pada sistem log aggregator adalah *concurrency*, *partial failure*, dan *indepedensi node*. *Publisher* dan *Aggregator* berjalan sebagai *container* terpisah, sehingga kegagalan pada satu *service* tidak serta-merta menghentikan *service* lainnya. Namun, kondisi ini juga memunculkan tantangan seperti inkonsistensi data dan dupliaksi pesan.

   Dalam desain Pub-Sub aggregator, *trade off* utama terletak pada reliabilitas vs kompleksitas. Sistem ini memilih skema *at-least-once delivery* yang lebih sederhana terhadap kegagalan jaringan, namun berpotensi menghasilkan *event* duplikat. Untuk mengatasi hal tersebut, sistem memindahkan kompleksitas ke sisi aggregator melalui mekanisme *idempotency* dan *deduplication* (Coulouris dkk., 2012).

2. **T2 (Bab 2): Kapan memilih arsitektur *publish–subscribe* dibanding *client–server*? Alasan teknis.**

   Arsitektur *publish-subscribe* dipilih ketika sistem membutuhkan *loose coupling* antara pengirim dan penerima pesan. Sedangkan, client-server mengharuskan klien alamat dan status server secara langsung. Secara teknis, Pub-Sub lebih unggul Ketika jumlah produser berskala besar, *event* bersifat asinkron dan consumer dapat berubah tanpa memengaruhi *publisher*. Model ini juga lebih tahan terhadap perubahan evolusioner sistem, misalnya ketika aggregator dikembangkan untuk menambahkan analitik baru. Sebaliknya, *client–server* lebih cocok untuk interaksi sinkron dan permintaan yang membutuhkan respons langsung dan deterministik. Dalam konteks sistem ini, Pub-Sub memungkinkan simulasi beban tinggi tanpa memblokir *publisher*, karena komunikasi bersifat *non-blocking* dan *batch-based* (Coulouris dkk., 2012).

3. **T3 (Bab 3): *At-least-once* vs *exactly-once delivery*; peran idempotent consumer.**

   At-least-once delivery menjamin bahwa setiap pesan akan dikirim setidaknya satu kali, namun memungkinkan pengiriman ulang jika terjadi kegagalan jaringan atau *timeout*. Sebaliknya, *exactly-once delivery* menjamin pesan diproses tepat satu kali, tetapi membutuhkan koordinasi kompleks seperti *distributed transaction* atau *two-phase commit*, yang mahal dan sulit diskalakan.

   Sistem ini secara sadar memilih *at-least-once delivery* karena lebih realistis dan *robust* di lingkungan terdistribusi. Risiko utama pendekatan ini adalah duplikasi *event*,

yang kemudian ditangani melalui *idempotent consumer*. Aggregator dirancang agar pemrosesan event dengan (`topic, event_id`) yang sama tidak menghasilkan efek samping berulang. Dengan demikian, walaupun *event* dikirim ulang, hasil akhir sistem tetap konsisten (Coulouris dkk., 2012).

4. **T4 (Bab 4): Skema penamaan topic dan event_id (unik, *collision-resistant*) untuk *dedup*.**

   Skema penamaan yang baik merupakan fondasi penting untuk deduplication. Pada sistem ini, *topic* digunakan sebagai representasi kategori log (misalnya *user.created*, *order.placed*), sedangkan *event_id* berfungsi sebagai identitas unik *event*. Kombinasi (`topic, event_id`) dipilih sebagai kunci deduplication karena secara semantik mewakili satu kejadian logis. Agar *collision-resistant*, `event_id` dihasilkan secara deterministik dan unik di sisi *publisher*. Walaupun implementasi saat ini menggunakan format *string* berurutan, konsepnya dapat diperluas dengan UUID atau *hash* berbasis waktu (Coulouris dkk., 2012).

5. **T5 (Bab 5): *Ordering* praktis (*timestamp* + *monotonic counter*); batasan dan dampaknya.**

   Ordering global sulit dicapai pada sistem terdistribusi karena tidak adanya *global clock* yang sempurna. Oleh karena itu, sistem ini menggunakan pendekatan praktis berupa timestamp berbasis waktu UTC, yang cukup untuk kebutuhan *log* dan *monitoring*. Ordering ini bersifat *best-effort*, bukan jaminan total order.

   Keterbatasan pendekatan ini adalah potensi ketidakteraturan ketika *event* dikirim hampir bersamaan atau terjadi *skew* waktu antar node. Namun, karena sistem tidak bergantung pada ordering ketat untuk konsistensi bisnis, dampaknya dapat diterima (Coulouris dkk., 2012).

6. **T6 (Bab 6): *Failure modes* dan mitigasi (*retry, backoff, durable dedup store, crash recovery*).**

   Sistem ini menghadapi beberapa *failure mode*, seperti kegagalan jaringan, *crash container*, dan *retry* berulang dari publisher. Mitigasinya dilakukan melalui *retry* otomatis di *publisher*, *backoff* sederhana, serta *durable dedup store* berbasis *SQLite*. Jika aggregator crash dan di-*restart*, data tetap tersedia karena disimpan di Docker volume.

7. **T7 (Bab 7): Eventual consistency pada aggregator; peran *idempotency* + *dedup*.**

Aggregator mengadopsi *eventual consistency*, di mana statistik dan daftar *event* mungkin tidak langsung konsisten, tetapi akan mencapai keadaan stabil setelah semua *event* diproses. *Idempotency* dan *deduplication* memastikan bahwa konsistensi akhir tercapai meskipun terjadi duplikasi.

8. **T8 (Bab 8): Desain transaksi: ACID*, isolation level*, dan strategi menghindari *lost-update*.**

   SQLite digunakan sebagai *dedup store* dengan properti *ACID* bawaan. Setiap *insert event* dilakukan secara atomik, sehingga mencegah *partial write*. *Isolation* dijaga melalui mekanisme locking internal *SQLite*, yang cukup untuk skala sistem ini. Desain ini mencegah *lost update* tanpa memerlukan *distributed transaction*.

9. **T9 (Bab 9): Kontrol konkurensi: locking/unique constraints/upsert; idempotent write pattern.**

   Kontrol konkurensi dilakukan melalui unique constraint pada (`topic, event_id`) dan pola *idempotent write*. Jika dua *worker* memproses *event* yang sama secara bersamaan, hanya satu yang berhasil *insert*, sementara lainnya dianggap duplikat.

10. **T10 (Bab 10–13): Orkestrasi *Compose*, keamanan jaringan lokal, persistensi (volume), *observability*.**

    Docker Compose digunakan untuk orkestrasi *multi-service*, dengan *network* internal untuk keamanan. Persistensi dijamin melalui volume, sementara observability disediakan melalui *logging* dan *endpoint* `/stats`. Pendekatan ini mencerminkan praktik standar sistem terdistribusi modern yang modular dan mudah dikelola.