Implementasi *Optimistic Concurrency Control* pada Sistem Aplikasi *E-Commerce* Berdasarkan Arsitektur Micro*Service*s Menggunakan Kubernetes

Ammar Dwi Anwari, Rizky Januar Akbar, dan Royyana Muslim Ijtihadie
Departemen Teknik Informatika, Institut Teknologi Sepuluh Nopember (ITS)
e-mail: rizky@if.its.ac.id

***Abstrak*—*MicroService* memiliki banyak pendekatan dalam penerapannya. Salah satunya dengan membuat setiap *Service* bersifat *isolated*. Untuk memenuhi sifat *isolated* tersebut komunikasi dilakukan secara asinkronus dimana setiap *Service* berkomunikasi menggunakan bantuan dari event bus. Duplikasi data akan sering terjadi dikarenakan *Service* bersifat *isolated* yaitu setiap *Service* tidak bisa mengambil data pada database yang bukan miliknya. Oleh karena itu duplikasi data harus tetap sinkron di setiap *Service*. Permasalahan muncul pada saat dilakukan *scaling*. *Service* yang di scaling memproses event secara konkuren sehingga urutan eksekusi setiap *event* bisa saja tidak terurut. Hal ini memungkinkan keadaan nilai dari suatu data menjadi tidak konsisten diantara masing-masing database tiap *Service*. *Optimistic Concurrency Control* sebagai solusi terhadap masalah konsistensi data yang terjadi. Hasil dari solusi yang diterapkan membuat nilai data menjadi sinkron disetiap database *Service* dalam keadaan scaling.**

Kata Kunci*—*MicroServices, Concurrency Control, Optimistic Concurrency Control, Event Bus.

I. PENDAHULUAN

SEBUAH aplikasi yang memiliki banyak pengguna aktif membutuhkan infrastruktur yang memadai untuk mencegah kegagalan server. Dari masalah tersebut berbagai jenis arsitektur seperti *microServices* sering diterapkan diberbagai perusahaan maupun institusi. Dalam penerapan *microServices* terdapat banyak pendekatan salah satunya adalah dengan pendekatan komunikasi secara sinkronus. Komunikasi secara sinkronus terdapat kekurangan karena setiap *Service* memiliki *database* sendiri sehingga apabila *Service* A membutuhkan data di *Service* B maka *Service* A perlu melakukan *request* ke *Service* B dimana disaat melakukan hal tersebut akan terjadi *latency*. Kekurangan lainnya yaitu apabila *Service* B *down* maka *Service* A tidak dapat melakukan *request* ke *Service* B. Untuk menghindari kekurangan tersebut maka setiap *Service* haruslah bersifat *isolated* atau independen dengan *Service* lainnya. Untuk menerapkan sifat *isolated* tersebut komunikasi dilakukan secara asinkronus.

Komunikasi secara asinkronus sangat berhubungan dengan *event driven architecture* sehingga mekanisme sistem berdasarkan pada teori yang terdapat pada arsitektur *event driven*. Dalam pendekatan komunikasi asinkronus atau *isolated* duplikasi data akan sering terjadi dikarenakan setiap *Service* tidak bisa mengambil data pada *database* yang bukan miliknya. Oleh karena itu duplikasi data harus tetap sinkron di setiap *Service*.

Masalah yang muncul pada isu *concurrency* dapat digambarkan pada Gambar 1. *Service* yang di *scaling* memproses *event* secara konkuren sehingga urutan eksekusi setiap *event* bisa saja tidak terurut. Hal ini memungkinkan keadaan nilai dari suatu data menjadi tidak konsisten diantara masing-masing *database* tiap *Service*. Asumsi *Service* A dan *Service* B memiliki data X dengan nilai Y. *Service* A melakukan beberapa kali perubahan nilai pada data X. Pada saat melakukan perubahan, *Service* A mengirim *event* untuk melakukan perubahan pada *Service* B. Akibat dari *scaling* urutan proses *event* yang selesai dieksekusi di *Service* B menjadi tidak sesuai sehingga nilai data X menjadi tidak sesuai antara kedua *Service* A dan B.

Rumusan masalah dari makalah berikut ini adalah bagaimana menanggulangi *concurrency issue* karena ketidakkonsistenan dengan menerapkan metode *optimistic concurrency control*, dan mengimplementasikan mekanisme sistem *row versioning*. Penelitian ini membahas implementasi *optimistic concurrency control* dengan melakukan *row versioning* pada *database* sebagai solusi terhadap masalah konsistensi data.

II. URAIAN SISTEM

Sebelum membahas permasalahan dari makalah ini. Sistem perlu diuraikan sehingga memudahkan penjabaran analisis permasalahan. Dari uraian didapatkan informasi kondisi sistem, dan kondisi lingkungan yang menjadi dasar permasalahan.

A. Deskripsi Umum Sistem

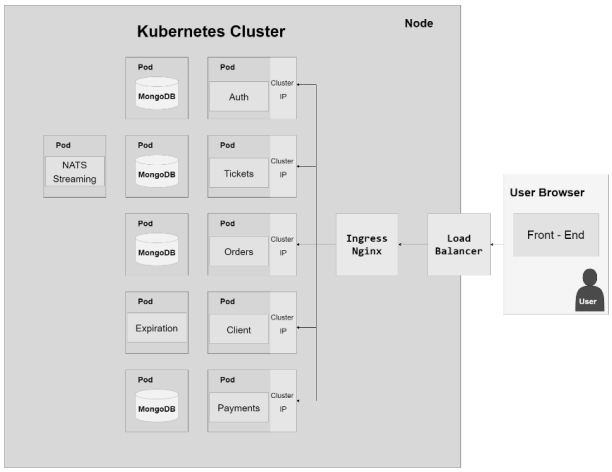
Sistem ini merupakan rancangan aplikasi *web marketplace* atau *e-commerce* untuk penjualan tiket online dengan arsitektur *microServices*. Komunikasi antar *Service* menggunakan pendekatan komunikasi asinkronus diamana membuat setiap *Service* menjadi bersifat *isolated*. Sistem menggunakan *message broker NATS Streaming* sebagai protokol untuk berkomunikasi secara asinkronus. Sistem rancangan aplikasi *e-commerce* yang dibuat meliputi berbagai *Service* yaitu *Service client (frontend), auth, tickets, orders, expiration,* dan *payments.* Pada sistem diterapkan metode *optimistic concurrency control* sebagai solusi ketidakkonsistenan nilai pada data.

B. MicroServices

MicroServices adalah *Service* yang kecil atau *autonomous* yang bekerja bersamaan. Kuncinya adalah kecil dan *autonomous*. Dimana *MicroServices* adalah aplikasi perangkat lunak yang kompleks terdiri dari satu atau lebih *Services* [1]. Pendekatan *microServices* sangat bervariasi salah satunya dengan komunikasi secara asinkronus.

Tabel 1.
Rancangan *services*

Kode	*Service*	Deskripsi
S01	*Auth Service*	Melayani proses pendaftaran pengguna, autentikasi pengguna, dan *logout* pada aplikasi.
S02	*Tickets Service*	Melayani pengelolaam tiket seperti menampilkan, penambahan, pengubahan dan penghapusan data tiket.
S03	*Orders Service*	Melayani proses pemesanan tiket atau mebuat pesanan, menampilkan riwayat pemesanan pengguna, menampilkan detail pesanan, menghentikan pesanan.
S04	*Payments Service*	Melayani proses pembayaran tiket.
S05	*Expiration Service*	Melayani proses perhitungan waktu batas kadaluarsa pembayaran tiket.
S06	*Client Service*	Melayani tampilan website yang dapat dilihat oleh pengguna di browser.



Gambar 1. Diagram arsitektur sistem.

Implementasi *Asynchronous Event-Based Collaboration*. Sebelumnya secara sinkronus menggunakan *REST* sebagai alat bantu untuk implementasi *request* atau *response pattern*. Alat bantu untuk implementasi asinkronus adalah *message broker*. Ada dua bagian yang perlu diperhatikan yaitu bagaiman cara *microServices emit events*, dan cara consumers tahu jika *event* telah terjadi. Sederhananya *message brokers* seperti *RabbitMQ* mencoba menangani kedua masalah tersebut. *Producers* menggunakan *API* untuk *publish event* ke broker dan broker menangani *subscriptions* dimana *consumers* diinformasikan bahwa *event* telah tiba [2].

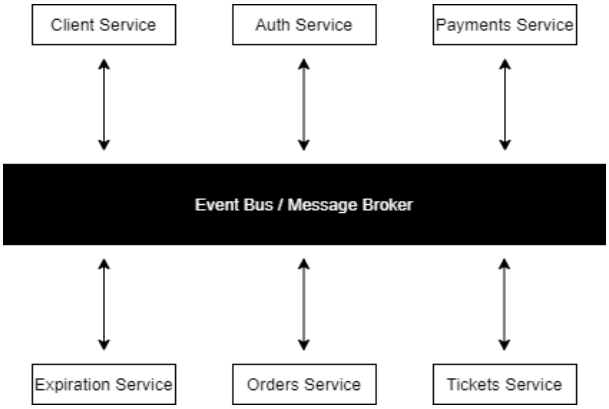
C. Service

Berbeda dengan arsitektur monolitik, arsitektur ini membagi *Service* menjadi beberapa *Service* dengan lingkup fungsionalitas yang lebih kecil ke dalam sebuah *container*. *Service* yang dibutuhkan dari sistem aplikasi ini dijelaskan pada Tabel 1.

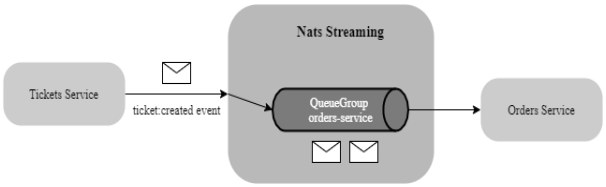
D. Infrastruktur

Sistem aplikasi dengan arsitektur *microServices* deterapkan dengan *tools kubernetes*. Gambaran umum infrastruktur aplikasi dijelaskan pada Gambar 1.

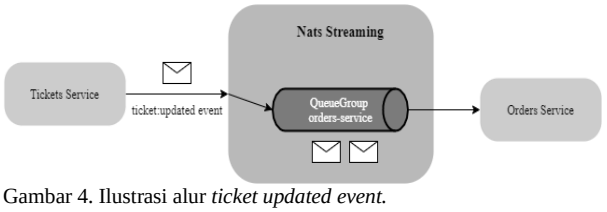
Pada Gambar 1 Infrastruktur yang digunakan didalam *kubernetes cluster* adalah pod dari masing-masing *Service* yaitu *Auth, Tickets, Orders, Client, Expiration* dan *Payments*. Kemudian pod dari *MongoDB* untuk masing-masing *Service* kecuali untuk *Expiration* dan *Client Service* karena tidak membutuhkan *database* dan pod untuk *Nats Streaming* server. Kemudian *Resource ingress-nginx* dan *load balancer* agar sistem dapat diakses dari luar *cluster*.



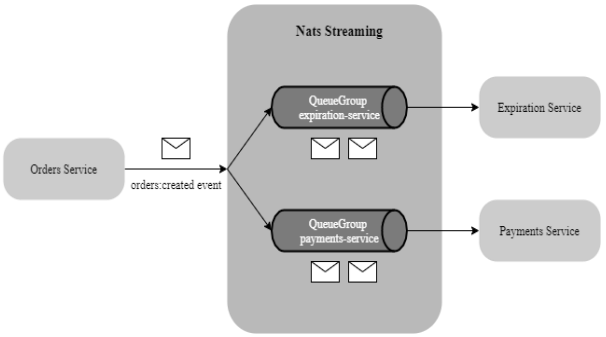
Gambar 2. Desain sistem dengan *event* bus.



Gambar 3. Ilustrasi alur *ticker created event.*



Gambar 4. Ilustrasi alur *ticket updated event.*

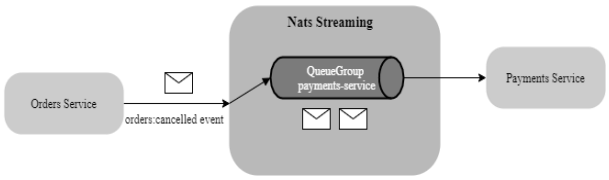


Gambar 5. Ilustrasi alur *order created event.*

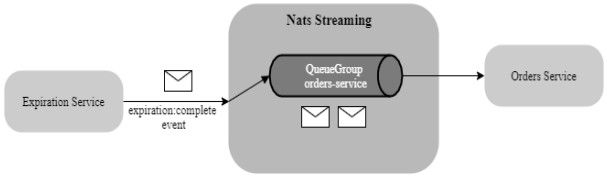
E. Event Bus

Event bus menggunakan konsep *publish* dan *subscribe*, desain *publish* dan *subscribe* digunakan untuk memudahkan komunikasi antar *Service* dan distribusi data, jadi apabila suatu *Service* melakukan *publish event* maka *Service* tidak perlu tahu kemana dia akan mengirimnya melainkan menjadi tugas *event bus* yang mendistribusikan *event* kepada *Service* yang melakukan *subscribe* ke *event* tersebut. Berikut desain sistem *event bus* terhadap *Service*.

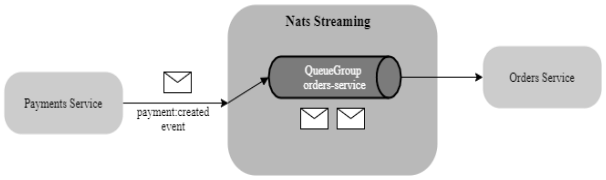
Dari Gambar 2 sistem tampak tidak perlu tahu ke *Service* mana event harus dipublikasikan melainkan hanya fokus pada *event bus*. Kemudian sistem di desain secara otomatis melakukan *revoke event* apabila suatu *Service* yang melakukan *subscribe* ke *event* tersebut *error* dalam selang waktu yang ditentukan. Untuk *event state* konsistensi apabila *Service* yang melakukan *subscribe* kesuatu *event* sedang *down*, apabila *Service* tersebut hidup kembali maka *event bus*



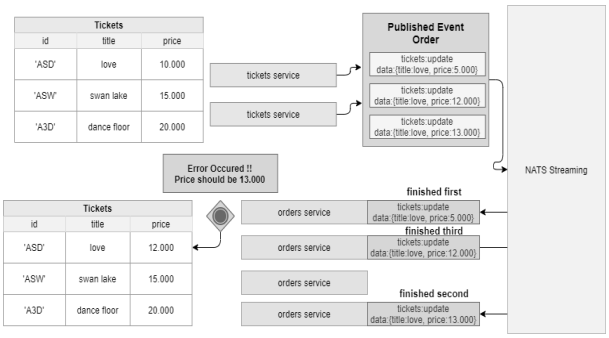
Gambar 8. Ilustrasi alur *order cancelled event*.



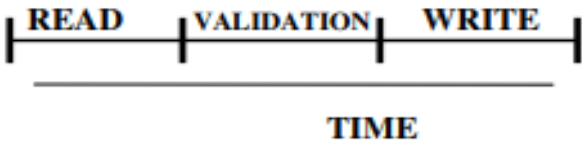
Gambar 9. lustrasi alur *expiration complete event*.



Gambar 10. Ilustrasi alur *payment created event.*



Gambar 11. Ilustrasi *concurrency issue.*



Gambar 12. OCC *phase.*

akan mengirim ulang semua *event* yang terlewatkan.

1) Ticket Created Event

Pada *event* ini berfungsi untuk menduplikasi *ticket* setiap kali pengguna membuat *ticket*. Berikut ilustrasi alur *ticket created event*.

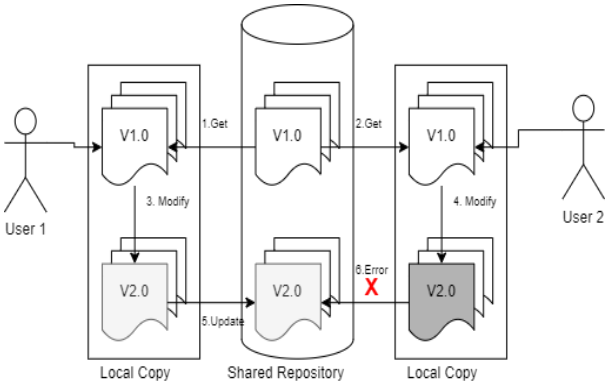
Pada Gambar 3 *Ticket Service* mempublikasikan *event* setiap kali pengguna membuat tiket kemudian *event* tersebut diteruskan ke *queueGroup* dan diterima oleh *orders Service* dan menduplikasikan data tiket. *Subscriber* untuk *event* ini hanyalah *orders Service* mengingat pada *orders Service* membutuhkan data tiket.

2) Ticket Updated Event

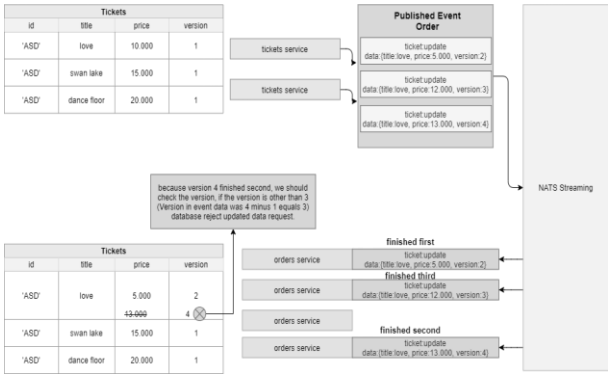
Pada *event* ini berfungsi untuk mensinkronisas data tiket setiap kali pengguna mengubah data tiket di *ticket Service*. Pada Gambar 4 *Ticket Service* mempublikasikan *event* setiap kali pengguna merubah tiket kemudian *event* tersebut diteruskan ke *queueGroup* dan diterima oleh *orders Service* kemudian merubah data tiket yang bersangkutan.

3) Order Created Event

Pada *event* ini berfungsi untuk menduplikasikan data *order* dan melakukan perhitungan mundur waktu kadaluarsa *order*. Pada Gambar 5 *Order Service* mempublikasikan *event* setiap kali pengguna membuat pesanan tiket kemudian *event* tersebut diteruskan ke *queueGroup* dan diterima oleh *expiration Service* dan *payments Service*. Pada *expiration Service subscriber* memproses waktu perhitungan kadaluarsa pesanan. sedangkan pada *payments Service subcriber* menduplikasikan data *order* untuk keperluan pembayaran.



Gambar 6. Ilustrasi *optimistic concurrency control*.



Gambar 7. Ilustrasi alur sistem dengan *optimistic concurrency control*.

Tabel 2.
Paramater uji coba pertama

Nama	Nilai
Jumlah *Replica Orders Service*	3
Jumlah *Replica Tickets Service*	3
Batasan Waktu	10s
Jumlah *Request Per Second*	10
Penerapan *Optimistic Concurrency Control*	Ya

4) Order Cancelled Event

Pada *event* ini berfungsi merubah status pesanan menjadi *cancelled*. Pada Gambar 6 *Orders Service* mempublikasikan *event* setiap pengguna melakukan *cancel* pada pesanan kemudian *event* tersebut diteruskan ke *queueGroup* dan diterima oleh *payments Service* dan merubah status pesanan menjadi *cancelled* sehingga pengguna tidak dapat melakukan pembayaran pada pesanan tersebut.

5) Expiration Complete Event

Pada *event* ini berfungsi untuk merubah status pesanan menjadi *cancelled* hampir sama halnya dengan *order cancelled event* bedanya hanya pada waktu *event* dipanggil. Pada Gambar 7 *Expiration Service* mempublikasikan *event* setiap kali perhitungan waktu kadaluarsa habis, *event* tersebut diteruskan ke *queueGroup* dan diterima oleh *orders Service* kemudian merubah status pesanan menjadi *cancelled*.

Tabel 9.
Hasil uji coba pertama

Nama *Service*	Jumlah Data Tiket
Tickets Service	100
Orders Service	100

Tabel 10.
Parameter uji coba kedua

Nama	Nilai
Jumlah *Replica Orders Service*	3
Jumlah *Replica Tickets Service*	3
Batasan Waktu	10s
Jumlah *Request Per Second*	100
Penerapan *Optimistic Concurrency Control*	Ya

Tabel 11.
Hasil uji coba kedua

Nama *Service*	Jumlah Data Tiket
Tickets Service	1000
Orders Service	1000

Tabel 12.
Parameter uji coba ketiga

Nama	Nilai
Jumlah *Replica Orders Service*	3
Jumlah *Replica Tickets Service*	3
Batasan Waktu	10s
Jumlah *Request Per Second*	1000
Penerapan *Optimistic Concurrency Control*	Ya

Tabel 13.
Hasil uji coba ketiga

Nama *Service*	Jumlah Data Tiket
Tickets Service	10.000
Orders Service	10.000

Tabel 14.
Parameter uji coba keempat

Nama	Nilai
Jumlah *Replica Orders Service*	1
Jumlah *Replica Tickets Service*	1
Batasan Waktu	10s
Jumlah *Request Per Second*	10
Penerapan *Optimistic Concurrency Control*	Tidak

Tabel 3.
Hasil uji coba keempat

Nama *Service*	Jumlah Data Tiket
Tickets Service	100
Orders Service	100

Tabel 4.
Parameter uji coba kelima

Nama	Nilai
Jumlah *Replica Orders Service*	1
Jumlah *Replica Tickets Service*	1
Batasan Waktu	10s
Jumlah *Request Per Second*	100
Penerapan *Optimistic Concurrency Control*	Tidak

Tabel 5.
Hasil uji coba kelima

Nama *Service*	Jumlah Data Tiket
Tickets Service	1000
Orders Service	1000

Tabel 6.
Parameter uji coba keenam

Nama	Nilai
Jumlah *Replica Orders Service*	1
Jumlah *Replica Tickets Service*	1
Batasan Waktu	10s
Jumlah *Request Per Second*	1000
Penerapan *Optimistic Concurrency Control*	Tidak

Tabel 7.
Hasil uji coba keenam

Nama *Service*	Jumlah Data Tiket
Tickets Service	10.000
Orders Service	10.000

Tabel 8.
Parameter uji coba ketujuh

Nama	Nilai
Jumlah *Replica Orders Service*	3
Jumlah *Replica Tickets Service*	3
Batasan Waktu	10s
Jumlah *Request Per Second*	10
Penerapan *Optimistic Concurrency Control*	Tidak

6) Payment Created Event

Pada *event* ini berfungsi untuk merubah status pesanan menjadi *complete*. Pada Gambar 8 *Payments Service* mempublikasikan *event* setiap kali pengguna menyelesaikan pembayaran kemudian *event* tersebut diteruskan ke *queueGroup* dan diterima oleh *orders Service* kemudian merubah status pesanan menjadi *completed*.

III. ANALISIS PERMASALAHAN

A. Concurrency Issue (Data Consistency)

Alur *event* dianalisis lebih lanjut khususnya *event ticket created* dan *event ticket updated.* Seperti yang diketahui *subsciber event ticket created* menduplikasi data tiket, disini yang bertindak sebagai *subscriber* adalah *orders Service* maka dapat diasumsikan pada *database* yang dimiliki *orders Service* memiliki data tiket yang sama dengan data tiket yang dimiliki *tickets Service*. Begitu juga dengan *event ticket updated* maka setiap perubahan data tiket di *tickets Service* maka data tiket yang dimiliki *orders Service* juga berubah.

Permasalahan timbul pada saat dilakukan *scaling*. Mengingat kemampuan eksekusi setiap *worker* bisa saja berbeda-beda (*out of order*). Jika eksekusi *event* tidak sesuai dengan urutan maka data bisa menjadi tidak konsisten. Berikut alur permasalahan pada sistem aplikasi yang dibuat dijelaskan pada Gambar 9.

Dari ilustrasi pada Gambar 9 dapat dilihat sistem di *scaling* pada *tickets Service* berjumlah dua dan *orders Service* berjumlah empat. Kemudian pada *ticket Service* terdapat sejumlah data *ticket* sama halnya dengan *orders Service*. *Ticket Service* melakukan beberapa kali perubahan data, dengan melakukan perubahan tersebut *ticket Service* mengirim sebuah *event ticket update*. Tampak pada ilustrasi

Tabel 15.
Hasil uji coba ketujuh

Nama *Service*	Jumlah Data Tiket
Tickets Service	100
Orders Service	100

Tabel 16.
Parameter uji coba kedelapan

Nama	Nilai
Jumlah *Replica Orders Service*	3
Jumlah *Replica Tickets Service*	3
Batasan Waktu	10s
Jumlah *Request Per Second*	100
Penerapan *Optimistic Concurrency Control*	Tidak

Tabel 17.
Hasil uji coba kedelapan

Nama *Service*	Jumlah Data Tiket
Tickets Service	1000
Orders Service	996

perubahan data dilakukan pada data *ticket* dengan *title love*. Perubahan dilakukan dengan mengganti nilai *price*. Urutan *ticket update event* yang di *publish* adalah perubahan nilai *price* menjadi 5.000, 12.000, dan 13.000. Nilai perubahan dibawa oleh *event* dalam bentuk *event data*. Dari ketiga *event* tersebut didistribusikan oleh *message broker* dan diterima oleh *orders Service*. Dikarenakan jumlah replikasi pada *orders Service* berjumlah empat maka *event* dapat diproses secara konkuren. Seperti yang dijelaskan sebelumnya bahwa waktu penyelesaian eksekusi pemrosesan *event* bisa berbeda-beda. Dari ilustrasi terlihat bahwa urutan penyelesaian *event ticket update* tidak sesuai dengan urutan perubahan yang dilakukan sebelumnya yaitu dengan urutan perubahan harga 5.000, 12.000, dan 13.000. Pada ilustrasi terlihat penyelesaian *event* terjadi dengan urutan pertama dengan perubahan *price* 5.000, urutan kedua dengan perubahan price 13.000, dan urutan ketiga dengan perubahan *price* 12.000. Dikarenakan ketidaksesuain urutan tersebut terlihat nilai akhir *price* pada data dengan *title love* menjadi 12.000 dimana harusnya adalah 13.000.

IV. METODE PENYELESAIAN

A. Strategi Penyelesaian dengan Pessimistic

Metode *optimistic concurrency control* Mekanisme yang berorientasi pada locking dapat disebut sebagai *pessimistic*. *Pessimistic* mengunci *database resource* meskipun transaksi tidak terjadi konflik. Dengan kata lain metode ini membuat eksekusi transaksi tidak bisa dilakukan secara konkuren. Mekanisme ini termasuk salah satu *Locking Oriented Concurrency Control* (LOCC) [3]. Mekanisme *locking* melindungi pemilik yang melakukan penguncian dari pengguna lain yang memodifikasi data. Asumsi Objek X dikunci maka pengguna lain tidak dapat mengakses objek tersebut. Jika penguncian tidak berujung pada penyelesaian transaksi maka dapat terjadi *deadlock*.

Dapat disimpulkan metode *pessimistic* dapat menimbulkan *deadlock* dan transaksi tidak dapat dieksekusi secara konkuren. Dari kesimpulan tersebut mematikan keunggulan dari *microServices* yang dapat melakukan *scaling* pada

Tabel 18.
Parameter uji coba kesembilan

Nama	Nilai
Jumlah *Replica Orders Service*	3
Jumlah *Replica Tickets Service*	3
Batasan Waktu	10s
Jumlah *Request Per Second*	1000
Penerapan *Optimistic Concurrency Control*	Tidak

Tabel 19.
Hasil uji coba kesembilan

Nama *Service*	Jumlah Data Tiket
Tickets Service	10.000
Orders Service	9.986

Service tertentu, sehingga metode ini tidak baik untuk diterapkan pada sistem.

B. Strategi Penyelesaian dengan Optimistic Concurrency Control

Yang perlu diperhatikan dalam *concurrency control algorithm* adalah memproses transaksi yang konflik dengan benar [4]. Basis ide pada mekanisme *optimistic concurrency control* (OCC) adalah eksekusi transaksi terdiri dari tiga fase yaitu *read*, *validation,* dan *write*.

Pada Gambar 10 fase *read*, transaksi membaca nilai pada data yang telah di *commit* sebelumnya kedalam *database*. Data kemudian diterima dalam bentuk *local copy* dengan versi atau granula, kemudian melakukan perubahan nilai pada data *local copy*. Semua operasi perubahan tersimpan didalam *tempopary update file* yang mana tidak diakses oleh transaksi yang lain. Operasi perubahan mengalokasikan *timestamp* atau versi pada masing-masing transaksi yang mana digunakan pada fase validasi nanti.

Validation phase, pada fase validasi transaksi divalidasi untuk memastikan perubahan yang dibuat tidak berdampak pada integritas dan konsistensi pada *database*. Jika proses validasi positif maka transaksi menuju fase *write*. Sebaliknya jika proses validasi negatif, transaksi harus diulang dan perubahan akan dibuang. Pada fase ini konflik dideteksi dengan mengecek apakah kondisi versi *local copy* dengan versi didalam *database* terpenuhi.

Write phase, pada fase ini perubahan secara permanen dilakukan terhadap data di *database* termasuk versi data juga ikut berubah. Fase ini hanya berlaku dalam mode transaksi *Read-Write* tidak untuk *Read-Only*. Keunggulan pada metode ini adalah metode ini sangat efektif dan efisisen ketika konflik jarang terjadi, transaksi yang konflik harus di *roll back*. Kelemahan dari metode ini adalah konflik sangat mahal untuk tangani dikarenakan transaksi yang konflik harus di *roll back*. Dari keunggulan dan kekurangan maka metode ini sangat cocok untuk diterapkan pada lingkungan yang jarang terjadi konflik [5].

Metode *optimistic concurrency control* Pendekatan *OCC* digunakan dengan *versioning* sistem. Gambar 11 menunjukkan ilustrasi pendekatan yang dilakukan. Berikut urutan penjelasan ilustrasi pada Gambar 11 berdasarkan cerita pengguna: (1) *User* 1 meminta data dari *shared repository* dengan versi V1.0. (2) Bersamaan dengan *User* 1, *User 2* meminta data dari *shared repository* dengan versi V1.0. (3) *User 1* melakukan perubahan pada data yang

diterima sebelumnya dan versi dari data tersebut menjadi versi V2.0. (4) *User 2* melakukan perubahan pada data yang diterima sebelumnya dan versi dari data tersebut menjadi versi V2.0. (5) *User 1* mengirim perubahan tersebut kepada *shared repository* dan *shared repository* menerima perubahan data tersebut menjadi versi V2.0. Pada tahap ini ketika *User* lain meminta data kepada *shared repository* maka data yang diterima adalah versi V2.0. (6) *User 2* mengirim perubahan tersebut kepada *shared repository* dan *shared repository* menolak perubahan data tersebut dikarenakan data yang dikirim adalah versi V2.0 sedangkan data di *shared repository* sudah menjadi versi V2.0.

Pada tahap ini sistem sudah menerapkan *optimistic concurrency control* dan perubahan data menjadi konsisten dan sesuai dengan urutan eksekusi.

1) Keunggulan

Keunggulannya meliputi *High Concurrency* (Metode OCC tidak menggunakan locking sehingga transaksi dapat dijalankan secara konkuren), *Deadlock free* (Tidak seperti *pessimistic* yang berorientasi pada mekanisme *locking*, sehingga *optimistic* dapat terhindari dari *deadlock*), serta *Resilient to Site Failure* (*Optimistic* menerima semua proses transaksi dan konflik akan diproses ulang dibelakang. Sehingga pada klien tidak menerima pesan *error*).

2) Kekurangan

Kekurangannya meliputi: (1) Terbatas pada sistem yang jarang terjadi konflik, jika optimistic diterapkan pada sistem yang sering terjadi konflik maka biaya pemrosesan akan menjadi semakin besar. (2) Terbatas pada sistem yang jarang terjadi konflik, transaksi yang konflik sulit untuk ditangani.

C. Penerapan Optimistic Concurrency Control pada Sistem

Seperti penjelasan sebelumnya *field version* digunakan sebagai indikator pengecekan *concurrency control*. Pengecekan *version* atau validasi dilakukan sebelum *database* mengeksekusi transaksi. Sistem di implementasikan dengan pengecekan *version* saat aksi *pre-save* guna mencegah terjadinya transaksi apabila versi tidak sesuai. Pada dasarnya sistem mengecek apakah versi yang diterima oleh *listener* adalah versi dikurangi satu pada data versi di *database Service* yang menerima *event*. Alur solusi yang diterapkan pada sistem dijelaskan oleh Gambar 12.

Dari ilustrasi pada Gambar 12 spesifikasi *scaling* dan urutan perubahan sama dengan yang dijelaskan oleh ilustrasi pada Gambar 9. Bedanya adalah *event* membawa data versi yang digunakan untuk validasi. Seperti yang dijelaskan sebelumnya bahwa waktu penyelesaian eksekusi pemrosesan *event* bisa berbeda-beda. Dari ilustrasi terlihat bahwa urutan penyelesaian *event ticket update* tidak sesuai dengan urutan perubahan yang dilakukan sebelumnya yaitu dengan urutan perubahan harga 5.000, 12.000, dan 13.000.

Pada ilustrasi terlihat penyelesaian *event* terjadi dengan urutan pertama dengan perubahan *price* 5.000, urutan kedua dengan perubahan *price* 13.000, dan urutan ketiga dengan perubahan *price* 12.000. Dikarenakan ketidaksesuain urutan tersebut sistem mengecek versi pada *database* yang sebelumnya menjadi versi 2. *Error* muncul pada proses eksekusi yang selesai dengan urutan kedua yaitu perubahan harga 12.000, pada *event* tersebut seharusnya selesai pada urutan ketiga akan tetapi *event* tersebut membawa data versi 4. Dari data versi tersebut terjadi konflik dikarenakan proses validasi gagal yakni versi yang harus dibawa adalah versi 3.

V. UJI COBA DAN EVALUASI

Pengujian diimplementasikan dengan metode load testing. Dalam *load* testing beberapa parameter perlu diperhatikan. Hasil yang didapatkan berupa perbandingan jumlah data. Eksperimen dilakukan dengan beberapa kali uji coba dengan parameter yang berbeda-beda. Hasil uji coba ketiga ditunjukkan pada Tabel 7.

Dari beberapa kali uji coba yang dilakukan, dapat dilihat pada parameter uji coba ketujuh yang ditunjukkan oleh Tabel 14, parameter uji coba kedelapan pada Tabel 16, dan parameter uji coba kesembilan pada Tabel 18. Uji coba tersebut dilakukan dengan kondisi *scaling* tiga banding tiga dan sistem tidak menerapkan *optimistic concurrency control*. Hasil dari uji coba tersebut terlihat pada Tabel 15, Tabel 17, dan Tabel 19 yakni jumlah beberapa data tidak sama.

Pada hasil uji coba kedelapan yang ditunjukkan oleh Tabel 17 perbandingan jumlah data adalah 1000 dan 996, dan hasil uji coba kesembilan pada Tabel 19 perbandingan jumlah data adalah 10.000 dan 9.986. Dapat diasumsikan *concurrency issue* yaitu nilai data tidak sama terjadi pada kondisi uji coba dengan parameter *scaling* dan tidak menerapkan *concurrency control*. Maka dapat disimpulkan pada kondisi parameter tersebut adalah kondisi yang sangat mungkin terjadi *concurrency issue*.

Pada parameter uji coba pertama yang ditunjukkan oleh Tabel 2, parameter uji coba kedua pada Tabel 4, dan parameter uji coba ketiga pada Tabel 6. Uji coba tersebut dilakukan dengan kondisi parameter *scaling* tiga banding tiga dan solusi *optimistic concurrency control* diterapkan pada sistem. Terlihat hasil uji coba pada Tabel 3, Tabel 5, dan Tabel 6 tidak ada jumlah data yang tidak sama, masing-masing jumlah data hasil uji coba adalah satu banding satu. Maka dapat diasumsikan pada kondisi parameter *scaling* dan solusi diterapkan *concurrency issue* yaitu nilai data tidak sama dapat dihindari.

Pada uji coba keempat yang ditunjukkan oleh Tabel 8, parameter uji coba kelima pada Tabel 10, dan parameter uji coba keenam pada Tabel 12. Uji coba tersebut dilakukan dengan kondisi parameter tidak *scaling* yaitu satu banding satu dan solusi *optimistic concurrency control* tidak diterapkan. Dari hasil uji coba yang ditunjukkan oleh Tabel 9, Tabel 11, dan Tabel 13 tersebut tidak ada jumlah data yang tidak sama, masing-masing jumlah data hasil uji coba adalah satu banding satu.

Maka dapat diasumsikan bahwa meskipun sistem tidak menerapkan *concurrency control* asalkan tidak dilakukan *scaling* maka *concurrency issue* yaitu nilai data tidak sama tidak muncul.

VI. KESIMPULAN/RINGKASAN

Dari hasil uji coba yang telah dilakukan terhadap perancangan dan implementasi *Optimistic Concurrency Control* untuk menyelesaikan permasalahan *Concurrency Issue (Data Consistency)* dapat diambil beberapa kesimpulan sebagai berikut: (1) Menanggulangi *Concurrency Issue*

berupa ketidak konsistenan data seperti yang dijelaskan pada bab 3 analasis permasalahan, Penyelesaian dapat dilakukan dengan menerapkan metode *optimistic concurrency control* seperti yang dirancang pada bab 4 dan dibuktikan pada hasil dari uji coba pertama, kedua, dan ketiga. (2) Implementasi *optimistic concurrency control* diterapkan seperti desain perancangan sistem yang dijelaskan pada bab 4 subbab bagian B yaitu dengan sistem *row versioning* dan teknis implementasi diperjelas pada bab 4 subbab bagian C.

DAFTAR PUSTAKA

[1] K. Bakshi, "Microservices Based Software Architecture and Approaches," *EEE Aerosp. Conf.*, pp. 1–8, 2017.

[2] A. Balalaie, A. Heydarnoori, and P. Jamshidi, "Microservices architecture enables devops: Migration to a cloud-native architecture," *Ieee Softw.*, vol. 33, no. 3, pp. 42–52, 2016.

[3] D. A. Menascé and T. Nakanishi, "Optimistic versus pessimistic concurrency control mechanisms in database management systems," *Inf. Syst.*, vol. 7, no. 1, pp. 13–27, 1982.

[4] B. Bhargava, "Concurrency control in database systems," *IEEE Trans. Knowl. Data Eng.*, vol. 11, no. 1, pp. 3–16, 1999.

[5] S. K. Singh, *Database Systems: Concepts, Design and Applications*, 3rd ed. India: Pearson Education, 2009.