Sistem Terdistribusi (ST)

Pertemuan 4 : Logikal dan Vektor Clock

Alauddin Maulana Hirzan

Fakultas Teknologi Informasi dan Komunikasi
Universitas Semarang

Outline

Latar Belakang

Sistem Terdistribusi dalam Industri Modern

Sistem terdistribusi digunakan pada hampir seluruh infrastruktur skala besar:

1. Google – Google Spanner, Bigtable
2. Amazon Web Services – DynamoDB, EC2 orchestration
3. Netflix – microservices berbasis event

Karakteristik utama:

1. Banyak node independen
2. Berkomunikasi melalui jaringan
3. Tidak berbagi memori atau clock global

Masalah Fundamental *Global Clock*

Setiap node memiliki:

1. CPU clock sendiri
2. OS time sendiri
3. Drift dan latency jaringan berbeda

Akibatnya:

- `Node A: 12:00:01`
- `Node B: 11:59:59`

Padahal Event di B mungkin terjadi setelah event di A

Clock Drift

Clock drift adalah perbedaan kecepatan clock antar node akibat:

1. variasi kristal oscillator
2. temperatur hardware
3. load CPU

Secara matematis:

$\frac{dC}{dt} \neq 1$

Clock Skew

Clock skew adalah selisih waktu absolut antar node:

$Skew = |Clock_A \check{} Clock_B|$

Dalam praktik cloud:

- Skew beberapa milidetik sudah dapat merusak sistem konsistensi kuat

Contoh:

- Distributed database menganggap write lama sebagai write terbaru

Mengapa Timestamp Lokal Bermasalah

Timestamp OS :

$2026 - 03 - 2614 : 03 : 01.021$

Namun timestamp tersebut:

1. tidak mempertimbangkan network delay
2. tidak menjamin urutan sebab-akibat

Contoh

Contoh:

- `A: write(x=1) at 10:00`
- `B: read(x) at 09:59`

Secara timestamp:

- read terjadi sebelum write

Namun secara realitas:

- read terjadi setelah write

Masalah ini dikenal sebagai: Causality violation

Konsep Event

Sistem terdistribusi dimodelkan sebagai:

$S = \{P1, P2, P3, ..., Pn\}$

Setiap proses menghasilkan *event*

$Ei = \{e1, e2, e3, ..., en\}$

Dimana:

1. S adalah Sistem Terdistribusi
2. P adalah Proses
3. E adalah Event

Jenis Event

Terdapat tiga jenis event

1. Event Lokal
   - update variable
   - write log
   - execute instruction
2. Event Pengiriman Pesan
   - send(m)
3. Event Penerimaan Pesan
   - receive(m)

Happened-Before Relation

Konsep ini diperkenalkan oleh: *Leslie Lamport* dalam artikel: *Time, Clocks, and the Ordering of Events in a Distributed System*

Definisi Formal

Relasi *happened-before* ditulis:

$A \rightarrow B$

Artinya: Event A secara kausal terjadi sebelum Event B. Relasi ini bukan sekadar waktu, melainkan hubungan sebab-akibat

Aturan *Happened-Before*

1. Urutan dalam Proses yang Sama

Jika: *e1 terjadi sebelum e2 pada proses yang sama*

Maka:

$e1 \rightarrow e2$

Contoh:

- `Process P1:`
- `e1:  x = 1`
- `e2:  send(x)`

Aturan *Happened-Before*

2. Pengiriman dan Penerimaan Pesan

Jika:

- `e1 = send(m)`
- `e2 = receive(m)`

Maka:

$e1 \rightarrow e2$

Ini adalah fondasi:

- causal consistency
- distributed transactions
- message ordering

Aturan *Happened-Before*

3. Transitivitas

Jika:

$A \rightarrow B$
$B \rightarrow C$

Maka:

$A \rightarrow C$

Relasi ini membentuk: Partial Order

Event yang Tidak Memiliki Relasi

Jika:

$A \not\rightarrow B$ dan $B \not\rightarrow A$

Maka event disebut: Concurrent Event

Contoh:

$P1 : write(x)$
$P2 : write(y)$

Dampak Iindustri

Konsep ini digunakan dalam:

1. Distributed Database
2. Google Spanner
3. Apache Cassandra

Untuk:

1. menentukan konflik write
2. multi-version concurrency control
3. Distributed Tracing

Mengapa *Happened-Before* Penting

Tanpa mekanisme ini:

1. debugging tidak mungkin dilakukan
2. log tidak dapat dipercaya
3. transaksi terdistribusi tidak dapat diserialisasi

Dalam sistem microservices:

User request $\rightarrow$ API Gateway $\rightarrow$ Service A $\rightarrow$ DB

Menentukan urutan kausal:

1. menentukan root cause failure
2. menentukan SLA violation

Hubungan Logical Clock

Happened-before menjadi dasar untuk:

- Lamport Logical Clock, dan
- Vector Clock

Keduanya digunakan untuk:

- memberikan timestamp yang konsisten secara kausal
- menggantikan ketergantungan pada wall-clock

Definisi Logical Clock

Dalam sistem terdistribusi, logical clock digunakan untuk menggantikan ketergantungan pada waktu fisik yang tidak sinkron. Logical clock memberikan mekanisme untuk mengurutkan kejadian berdasarkan hubungan kausal, bukan berdasarkan jam sistem.

Logical clock digunakan untuk:

Memberikan Timestamp Logis pada Setiap Event

Setiap event pada setiap proses diberikan nilai numerik:

Event e $\rightarrow$ timestamp C(e)

Timestamp ini:

- tidak merepresentasikan waktu nyata
- tetapi merepresentasikan urutan kausal

Definisi Logical Clock

Menentukan Urutan Parsial Kejadian

Logical clock menghasilkan Partial Order yang memenuhi:

Jika $A \rightarrow B$ maka $C(A) < C(B)$

Namun tidak semua event dapat dibandingkan:

$C(A) < C(B)$ tidak menjamin $A \rightarrow B$

Definisi Logical Clock

Mendukung Konsistensi dan Sinkronisasi

Logical clock digunakan dalam:

1. distributed database replication
2. distributed logging
3. distributed transaction ordering

Contoh implementasi:

1. Apache Cassandra menggunakan logical timestamp dalam conflict resolution
2. sistem message queue memastikan pesan diproses sesuai causal ordering

Lamport Logical Clock

Ide Utama

Setiap proses memiliki:

Ci = logical counter lokal

Counter:

- bertambah setiap terjadi event
- disinkronkan melalui pesan

Tujuan:

Jika A $\rightarrow$ B maka C(A) $<$ C(B)

Aturan *Lamport Clock*

Untuk setiap proses Pi, terdapat tiga aturan utama:

1. Aturan Event Lokal
2. Aturan Pengiriman Pesan
3. Aturan Penerimaan Pesan

Aturan *Lamport Clock*

Aturan Event Lokal

Sebelum setiap event:

$Ci = Ci + 1$

Contoh:

$C1 = 0$

Event $e1 \rightarrow C1 = 1$

Event $e2 \rightarrow C1 = 2$

Aturan *Lamport Clock*

Aturan Pengiriman Pesan

Saat proses mengirim pesan:

$send(m, timestamp = Ci)$

Struktur pesan:

$m = \{payload, timestamp\}$

Aturan *Lamport Clock*

Aturan Penerimaann Pesan

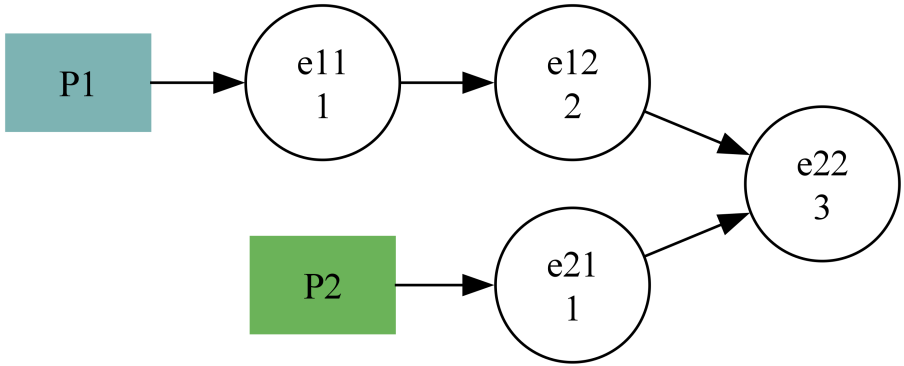
Jika proses menerima pesan dengan timestamp T:

$Ci = \max(Ci, T) + 1$

Hal ini memastikan:

$\text{send}(m) \rightarrow \text{receive}(m)$

Ilustrasi *Lamport Clock*



Contoh *Lamport Clock*

Misalkan terdapat tiga proses:

$P1, P2, P3$

Urutan Event

P1: e1 $\rightarrow$ send(m1, P2)
P2: receive(m1) $\rightarrow$ send(m2, P3)
P3: receive(m2)

Contoh *Lamport Clock*

Simulasi Counter

1. P1
   - `e1: C1 = 1`
   - `send(m1): C1 = 2`
2. P2
   - `receive(m1): C2 = 3`
   - `send(m2): C2 = 4`
3. P3
   - `receive(m2): C3 = 5`
4. Hasil
   - `C(e1) = 1`
   - `C(send m1) = 2`
   - `C(receive m1) = 3`
   - `C(send m2) = 4`
   - `C(receive m2) = 5`

Definisi Vector Clock

Batasan *Lamport Clock*

Lamport clock memiliki keterbatasan: tidak dapat membedakan causal ordering dan concurrent events

Vector Clock

Untuk mengatasi hal tersebut dikembangkan: Vector Clock

Vector clock memungkinkan sistem:

1. menentukan apakah dua event memiliki hubungan sebab-akibat
2. atau benar-benar terjadi secara paralel

Cara Kerja

Pada vector clock:

1. setiap proses menyimpan vektor timestamp
2. vektor menyimpan pengetahuan proses terhadap seluruh sistem

Jika terdapat N proses, maka:

Vi = [v1, v2, ..., vN]

Makna:

Vi[k] = jumlah event pada proses Pk yang diketahui oleh Pi

Dengan demikian:

- vector clock tidak hanya menyimpan waktu lokal
- tetapi juga sejarah komunikasi antar proses

Struktur Vector Clock

Setiap proses Pi memiliki:

Vi = [v1, v2, ..., vN]

Contoh sistem dengan 3 proses:

P1 $\rightarrow$ V1 $\rightarrow$ [0, 0, 0]
P2 $\rightarrow$ V2 $\rightarrow$ [0, 0, 0]
P3 $\rightarrow$ V3 $\rightarrow$ [0, 0, 0]

Contoh sistem dengan 2 proses:

P1 $\rightarrow$ V1 $\rightarrow$ [0, 0]
P2 $\rightarrow$ V2 $\rightarrow$ [0, 0]

Interpretasi Vector Clock

Misal diketahui:

V1 = [5, 2, 1]

Artinya:

1. P1 telah melakukan 5 event
2. mengetahui 2 event dari P2
3. mengetahui 1 event dari P3

Vector clock menyimpan: Causal History

Aturan Vector Clock

Vector clock mengikuti tiga aturan dasar.

1. Event Lokal
2. Pengiriman Pesan
3. Penerimaan Pesan

Aturan Vector Clock

1. Event Lokal

Untuk setiap event lokal pada proses Pi:

$Vi[i] = Vi[i] + 1$

Contoh:

1. `V1 = [0,0,0]`
2. `event lokal`
3. `V1 = [1,0,0]`

Aturan Vector Clock

2. Pengiriman Pesan

Saat proses mengirim pesan:

send(m, Vi)

Seluruh vektor disertakan dalam pesan:

$m = \{payload, vector\_timestamp\}$

Hal ini memungkinkan penerima mengetahui semua event yang diketahui pengirim

Aturan Vector Clock

3. Penerimaan Pesan

Saat proses Pi menerima pesan dari Pj:

- Langkah 1:
  - Untuk setiap k:

```
Vi[k] = max(Vi[k], Vj[k])
```

- Langkah 2:

```
Vi[i] = Vi[i] + 1
```

Ini memastikan:

send(m) $\rightarrow$ receive(m)

serta menyatukan pengetahuan kedua proses.

Relasi Antar Event

Dengan vector clock, relasi kausal dapat ditentukan secara matematis.

A terjadi sebelum B (A $\rightarrow$ B)

Jika:

$VA[k] \leq VB[k]$
$VA[k] < VB[k]$

Event Konkuren

Jika:

$VA \not\leq VB$
$VB \not\leq VA$

Contoh Vector Clock - Relasi Kausal

Diketahui:

VA = [2, 1] dan VB = [2, 3]

Perbandingan:

$2 \leq 2 = \checkmark \rightarrow$ sama dengan
$1 \leq 3 = \checkmark \rightarrow$ kurang dari

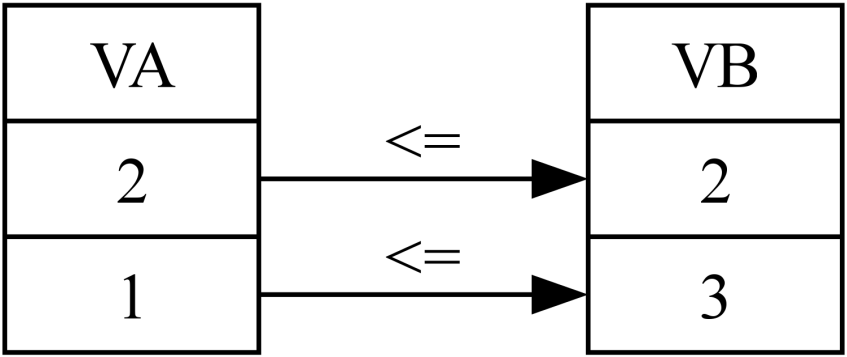
Ada Elemen Lebih Kecil

$1 < 3$

Sehingga:

VA $\rightarrow$ VB (event B mengetahui semua event A, B terjadi setelah A secara kausal)

Ilustrasi Interpretasi



Contoh Vector Clock - Event Konkuren

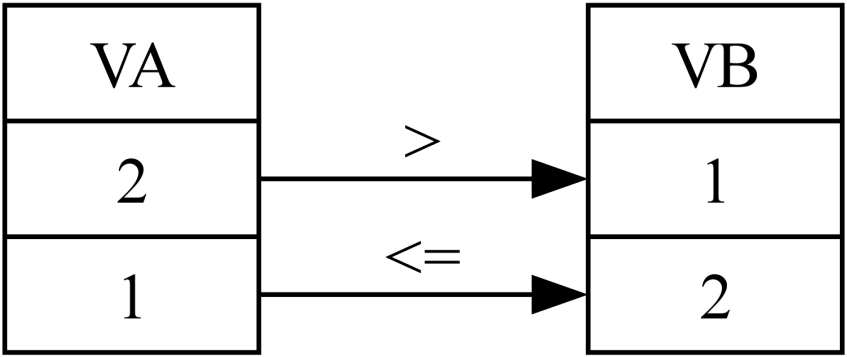
Diketahui:

VA = [2, 1] dan VB = [1, 2]

Perbandingan:

$2 \leq 1 = \times \rightarrow$ lebih dari
$1 \leq 2 = \checkmark \rightarrow$ kurang dari

Ilustrasi Interpretasi



Kompleksitas Vector Clock

Jika terdapat N proses:

- ukuran pesan: O(N)
- ukuran penyimpanan: O(N)

Hal ini menjadi masalah pada sistem skala besar dengan ribuan node

Untuk mengatasi hal ini, industri mengembangkan:

- version vector
- dotted version vector

Lastly

Terima Kasih