Sistem Terdistribusi

TIK-604

Toleransi Kegagalan

Kuliah 12: 13 s.d 15 Mei 2019

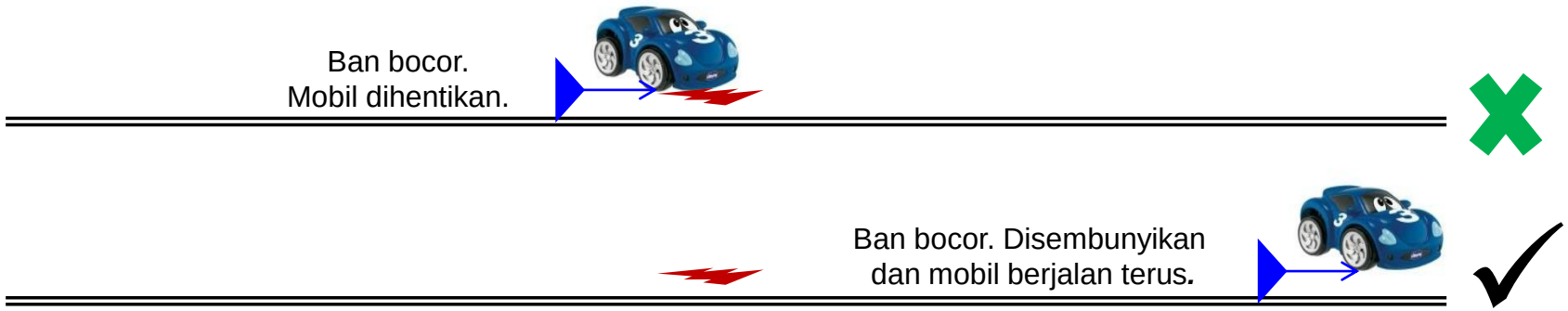
Husni

Hari ini...

- Pembahasan terakhir:
  - Replikasi
- Diskusi saat ini:
  - Toleransi Kegagalan (*Fault Tolerance*)
    - Definisi
    - Mendeteksi dan meyembunyikan kegagalan
      - 2PC
    - Penanganan Kegagalan Byzantine
- Pengumuman:
  - abc.

Fault-Tolerance

- Sistem dapat dirancang dalam suatu cara yang secara otomatis dapat pulih dari kegagalan (kerusakan) parsial (sebagian)



- ***Fault-tolerance*** adalah properti yang memungkinkan suatu system untuk terus beroperasi dengan benar bahkan jika suatu kerusakan (*failure*) terjadi pada waktu operasi.
- Sebagai Contoh, TCP dirancang untuk menyediakan komunikasi dua-arah yang *reliable* (dapat diandalkan) dalam jaringan packet-switched, bahkan dalam kehadiran link komunikasi yang tidak sempurna dan berbeban-lebih (*overloaded*).

Apa Itu Kegagalan?

- Kegagalan merupakan suatu deviasi (simpangan) dari jalan yang telah ditetapkan (*specified behavior*)
  - Misal, menekan pedal rem tidak menghentikan mobil ➔ kegagalan rem (dapat menjadi *bencana besar!*)
  - Misal, membaca suatu sector disk tidak memperoleh konten ➔ kegagalan disk (bukan bencara cukup besar)

- Banyak kegagalan disebabkan oleh perilaku spesifik yang salah
  - Ini biasanya terjadi ketika perancang gagal mengatasi skenario yang membuat sistem melakukan salah
  - Ini terutama benar dalam sistem yang kompleks dengan banyak interaksi yang halus

Karakteristik Kegagalan

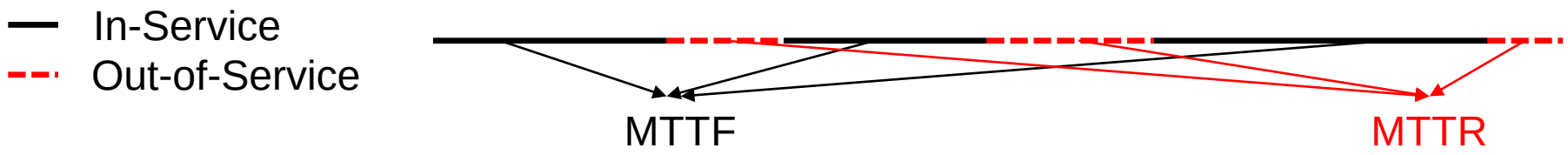
- Kegagalan dapat digolongkan sebagai *transient* atau *persistent*

- Transient Failures:
  - Also referred to as “soft failures” or “Heisenbugs”
  - Occur temporarily then disappear
  - Manifested only in a very unlikely combination of circumstances
  - Typically go away upon rolling back and/or retrying/rebooting
  - E.g., Frozen keyboard or window, race conditions and deadlocks, etc.

Karakteristik Kegagalan

- Kegagalan dapat digolongkan sebagai *transient* atau *persistent*

- Persistent Failures:
  - Persist until explicitly repaired
  - Retrying does not help
  - E.g., Burnt-out chips, software bugs, crashed disks, broken Ethernet cable, etc.
  - Durations of failures and repairs are random variables
  - Means of distributions are *Mean Time To Fail* (MTTF) and *Mean Time To Repair* (MTTR)



Availability vs. Reliability

- Ada perbedaan mendasar antara *availability* dan *reliability*
  - Ketersediaan mengacu pada probabilitas bahwa suatu sistem beroperasi dengan benar pada saat tertentu
    - Availability = MTTF/(MTTF+MTTR)
  - Keandalan mengukur berapa lama suatu sistem dapat beroperasi tanpa gangguan

- *Sistem yang sangat tersedia (highly-available, HA) adalah sistem yang kemungkinan besar akan bekerja pada waktu tertentu*
- *Sistem yang sangat andal (highly-reliable) adalah sistem yang kemungkinan besar akan terus bekerja tanpa gangguan selama periode waktu yang relatif lama*

Availability vs. Reliability

- Contoh:
  - Sebuah sistem yang down selama 1 ms setiap jam memiliki ketersediaan lebih dari 99,9999%, tetapi sangat tidak dapat diandalkan
  - Sebuah sistem yang tidak pernah crash tetapi dimatikan selama dua minggu setiap bulan Agustus memiliki keandalan yang tinggi, tetapi hanya ketersediaan 96%

Jenis Sistem	Availability (%)	Downtime Setahun
Workstation Konvensional	99	3.6 Hari
Sistem High-Available (HA)	99.9	8.4 Jam
Sistem Fault-Resilient	99.99	1 Jam
Sistem Fault-Tolerant	99.999	5 menit

Jenis Kegagalan

Jenis Kegagalan	Deskripsi
• Crash Failure	• Server berhenti tetapi telah bekerja dengan benar sampai ia terhenti

• Omission Failure
  • Receive Omission
  • Send Omission
	• Server gagal merespon request yang masuk
	  • Server gagal menerima message yang masuk
	  • Server gagal mengirimkan message

• Timing Failure	• Respon Server berada diluar interval waktu yang ditentukan

• Response Failure
  • Value Failure
  • State Transition Failure
	• Respon server tidak tepat
	  • Nilai responnya salah
	  • Server menyimpang dari aliran kendali yang benar

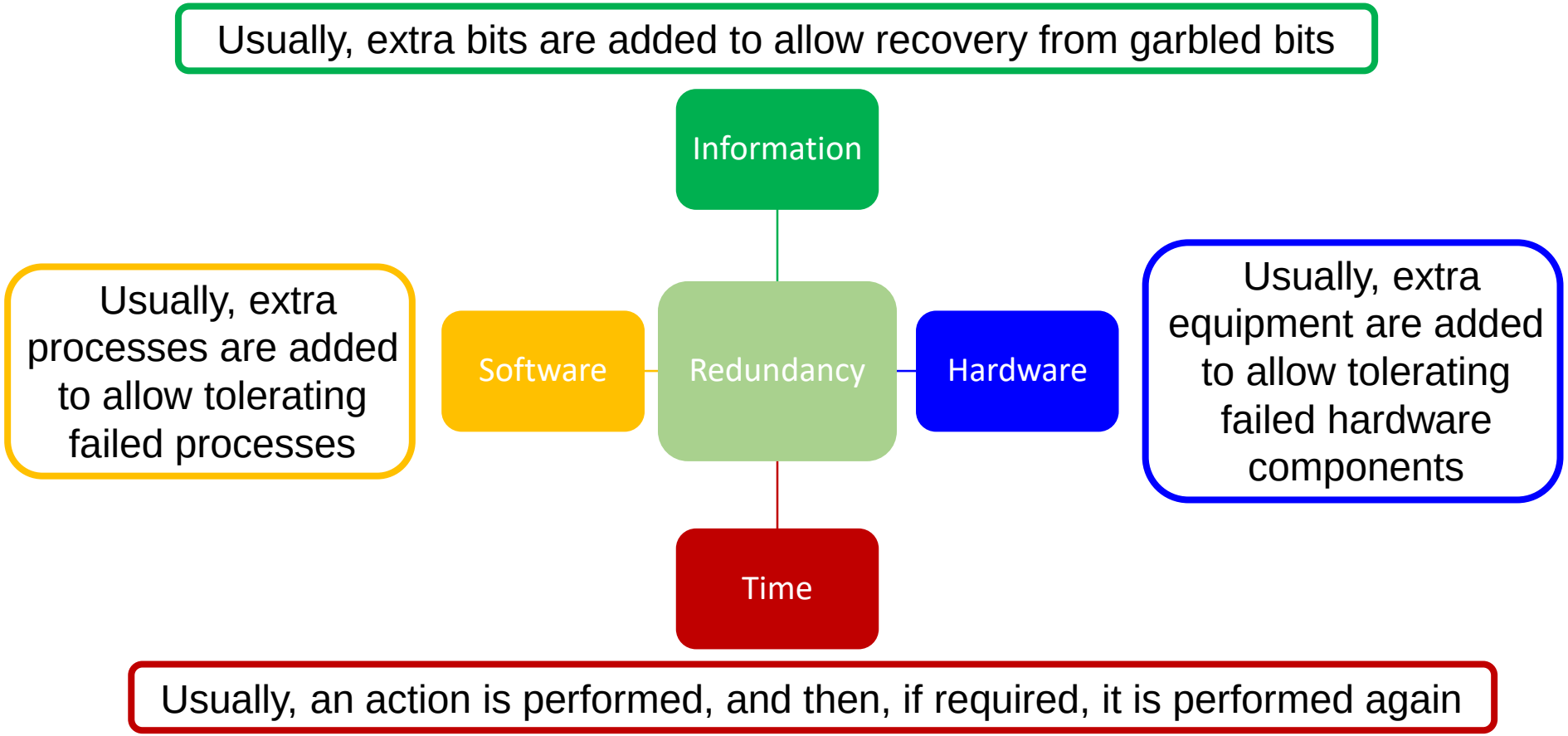
• Byzantine Failure	• Server dapat menghasilkan respon sembarang (berubah-ubah) pada waktu sembarang

Jenis Kegagalan

Jenis Kegagalan	Deskripsi
Secara umum dikenal sebagai Kegagalan *Fail-Stop* atau *Fail-Fast*	
Secara umum dikenal sebagai Kegagalan *Fail-Silent*	
Secara umum dikenal sebagai Kegagalan *Sembarangan* atau *Byzantine*	

Penyembunyian Kegagalan

- Teknik kunci untuk menutupi kegagalan adalah menggunakan *redundancy*



Deteksi Kegagalan

- Tapi, kegagalan perlu dideteksi sebelum bisa ditutupi

- Suatu sub-system Deteksi (terutama untuk kegagalan fail-stop atau fail-silent):
  - Biasanya dapat dilakukan sebagai efek samping dari bertukar informasi secara teratur dengan server
  - Idealnya harus dapat membedakan antara kegagalan jaringan dan server
    - Suatu proses, P, yang tidak dapat mencapai server dapat memeriksa dengan proses lain pada apakah mereka dapat mencapai server
      - Jika setidaknya satu proses lain menunjukkan bahwa ia dapat mencapai server, P dapat menganggap bahwa itu adalah kegagalan jaringan (dengan asumsi semua proses tidak berbahaya / tidak salah)

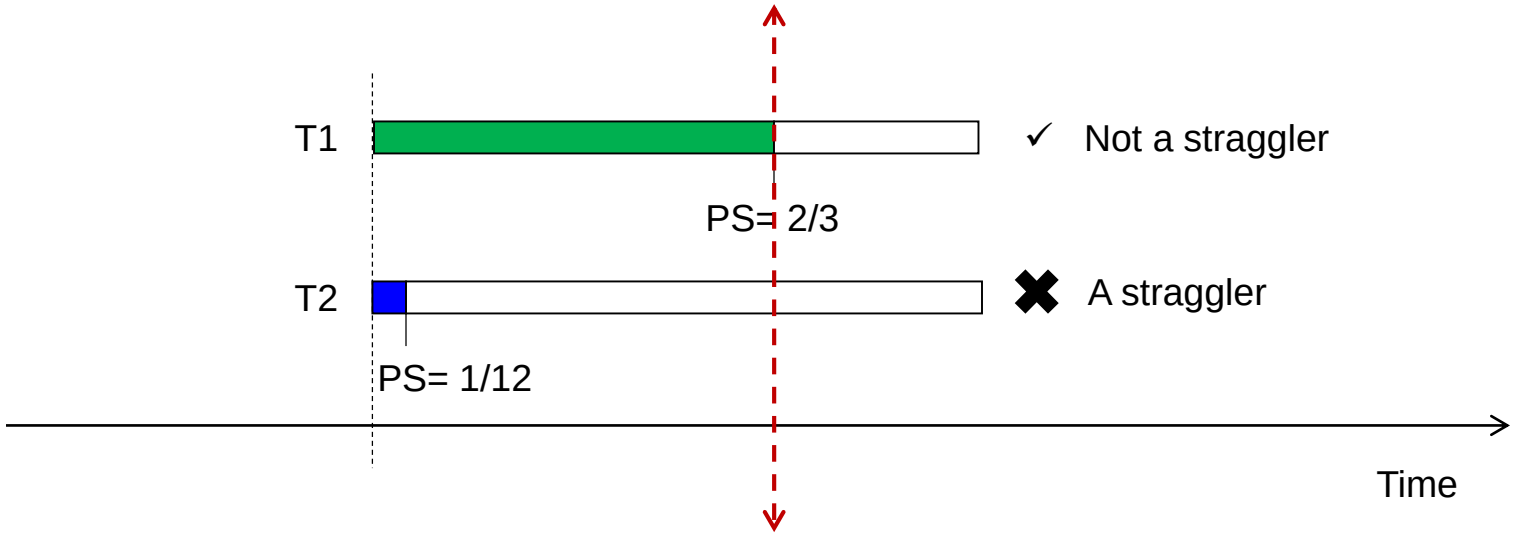
Contoh 1: Eksekusi Spekulatif dalam Hadoop

- A MapReduce job is dominated by the slowest task
- MapReduce attempts to locate slow tasks (or *stragglers*) and run *replicated* (or *speculative*) tasks that will optimistically commit before corresponding stragglers
- In general, this strategy is known as *task resiliency* or *task replication* (as opposed to *data replication*)
  - In Hadoop it is called *speculative execution*
- Only one copy of a straggler is allowed to be speculated
- Whichever task (among the two tasks) commits first, its results are exploited, and the other task is killed

Tetapi, Bagaimana Mendeteksi Stragglers?

- Hadoop monitors the progresses of all tasks and assigns each task a *progress score* between 0 and 1

- A task is marked as a straggler if its progress score (PS) < (average – 0.2) after running at least 1 minute



Contoh 2: Atomic Multicasting

- Atomic multicasting mengharuskan pemenuhan dua kondisi:
  1. Suatu message harus disampaikan ke *semua atau tidak sama sekali* dari proses-proses (atau situs-situs replikas)
     - Properti ini dikenal sebagai *atomicity*
  2. Semua messages harus dikirimkan *dalam urutan sama ke semua* proses (atau situs replika)
     - Properti ini dikenal sebagai *consistent ordering*

- Properti atomicity (valensi) memerlukan *reliable* multicasting karena ia menjamin bahwa SEMUA (atau tidak sama sekali) dari proses akan menerima pesan multicast tersebut.

Pengurutan Pesan

- Seperti sebelumnya, pada dasarnya ada tiga jenis *message ordering*:
  1. Pengurutan Sequential (Berurutan atau FIFO)
     - Messages yang dikirim *dari proses yang sama* dikirimkan dalam urutan yang sama sebagaimana mereka dikirimkan pada setiap proses yang menerima
  2. Pengurutan Causal (sebab musabab)
     - Jika message m_1 *causally* sebelum message m_2, m_1 disampaikan sebelum m_2 pada setiap proses yang menerima
  3. Pengurutan Total
     - Messages disampaikan *dalam urutan yang sama* pada setiap proses yang menerima.

Jenis *Reliable Multicasting*

- Umumnya ada perbedaan antara *enam* jenis *reliable multicasting*

Jenis Multicasting	Pengurutan Message Dasar	Penyampaian Terurut Total?
Reliable multicasting	None	No
FIFO multicasting	FIFO-ordered delivery	No
Causal multicasting	Causal-ordered delivery	No
Atomic multicasting	None	Yes
FIFO atomic multicasting	FIFO-ordered delivery	Yes
Causal atomic multicasting	Causal-ordered delivery	Yes

Transaksi Atomik Terdistribusi

- Atomic multicasting adalah contoh masalah umum yang dikenal sebagai *distributed atomic transactions*
  - Diberikan suatu transaksi dengan banyak aksi
    - Semua atau tidak satupun aksi dilaksanakan (*committed*)
    - Jika semua aksi dilaksanakan, maka akan dilaksanakan dalam urutan yang sama pada semua situs replika

- Protokol distributed atomic transaction yang popular dikenal sebagai *two-phase commit protocol* (2PC), yang berisi:
  - Satu *koordinator*
  - Banyak *partisipan*

Protokol Two-Phase Commit: 2PC

- 2PC terdiri dari dua fase, masing-masing melibatkan dua langkah:

Fase I: Voting (Pengumpulan Suara)	
Langkah 1	• Koordinator mengirimkan suatu pesan VOTE_REQUEST ke semua partisipan.
Langkah 2	• Ketika partisipan menerima pesan VOTE_REQUEST, ia mengembalikan pesan VOTE_COMMIT kepada Koordinator yang mengindikasikan bahwa ia siap untuk secara local melaksanakan bagiannya dari transaksi tersebut, atau pesan VOTE_ABORT.

Protokol Two-Phase Commit

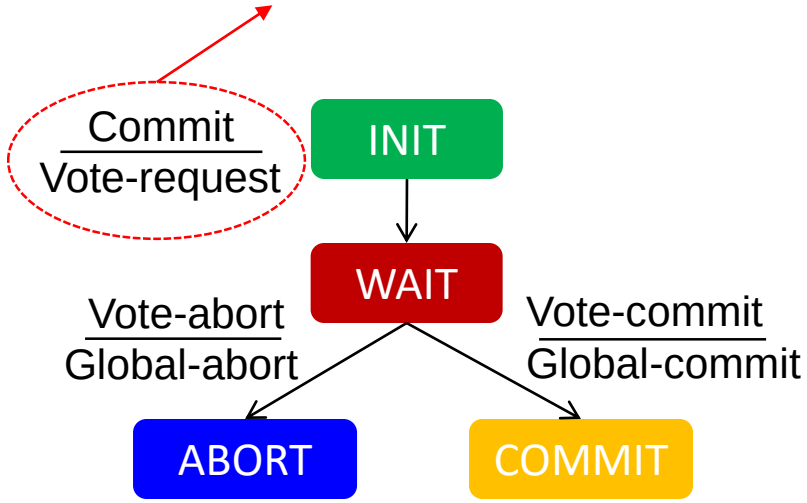
Phase II: Decision Phase

Langkah 1

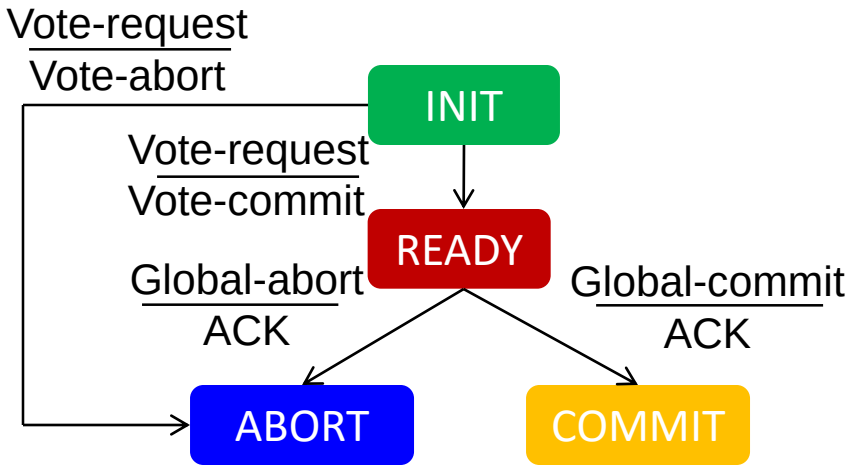
Langkah 2

Mesin Status Terbatas: 2PC

Catatan: Istilah di atas dan di bawah garis menunujukkan apa yang telah diterima dan dikirim



Mesin Status Terbatas dari **KOORDINATOR** dalam2PC



Mesin Status Terbatas dari **PARTISIPAN** dalam 2PC

Algortima 2PC

Tindakan oleh Koordinator:

```
write START_2PC to local log;
multicast VOTE_REQUEST to all participants;
while not all votes have been collected{
        wait for any incoming vote;
        if timeout{
                write GLOBAL_ABORT to local log;
                multicast GLOBAL_ABORT to all participants;
                exit;
        }
        record vote;
}
If all participants sent VOTE_COMMIT and coordinator votes COMMIT{
        write GLOBAL_COMMIT to local log;
        multicast GLOBAL_COMMIT to all participants;
}else{
        write GLOBAL_ABORT to local log;
        multicast GLOBAL_ABORT to all participants;
}
```

Pemulihan Koordinator

- Koordinator dapat gagal pada sembarang tahapan dalam 2PC

- Akan tetapi dikarenakan *logging* statusnya, ia dapat pulih sebagai berikut:

Status dalam Log	Tindakan Setelah Pemulihan (Recovery)
INIT	Abort
WAIT	Retransmit VOTE_REQUEST to participants
COMMIT	Retransmit GLOBAL_COMMIT to all participants
ABORT	Retransmit GLOBAL_ABORT to all participants

Protokol Two-Phase Commit

Tindakan oleh Partisipan

```
write INIT to local log;
Wait for VOTE_REQUEST from coordinator;
If timeout{
        write VOTE_ABORT to local log;
        exit;
}
If participant votes COMMIT{
        write VOTE_COMMIT to local log;
        send VOTE_COMMIT to coordinator;
        wait for DECISION from coordinator;
        if timeout{
                multicast DECISION_RQUEST to other participants;
                wait until DECISION is received; /*remain blocked*/
                write DECISION to local log;
        }
        if DECISION == GLOBAL_COMMIT { write GLOBAL_COMMIT to local log;}
        else if DECISION == GLOBAL_ABORT {write GLOBAL_ABORT to local log};
}else{
        write VOTE_ABORT to local log;
        send VOTE_ABORT to coordinator;
}
```

Protokol Two-Phase Commit

Tindakan untuk menangani permintaan keputusan:

```
/*executed by separate thread*/

while true{
        wait until any incoming DECISION_REQUEST is received; /*remain blocked*/
        read most recently recorded STATE from the local log;
        if STATE == GLOBAL_COMMIT
                send GLOBAL_COMMIT to requesting participant;
        else if STATE == INIT or STATE == GLOBAL_ABORT
                send GLOBAL_ABORT to requesting participant;
        else
                skip; /*participant remains blocked*/
}
```

- An *indefinite blocking window* can arise, whereby all sites who voted positively are blocked until outcome is known
- Can any clever protocol avoid this window?
  - No!
- All distributed commit protocols have an indefinite blocking window!

Pemulihan Partisipan

- Partisipan dapat gagal pada sembarang tahapan dalam 2PC

- Dikarenakan *logging* statusnya, ia dapat pulih sebagai berikut:

Status dalam Log	Tindakan setelah Pemulihan (Recovery)
INIT	Secara local gugur dan beritahukan koordinator
READY	Tidak dapat memutuskan sendiri apa yang harus dilakukan selanjutnya; karena itu hubungi lainnya
COMMIT	Kirim ulang keputusannya kepda koordinator
ABORT	Kirim ulang keputusannya kepda koordinator

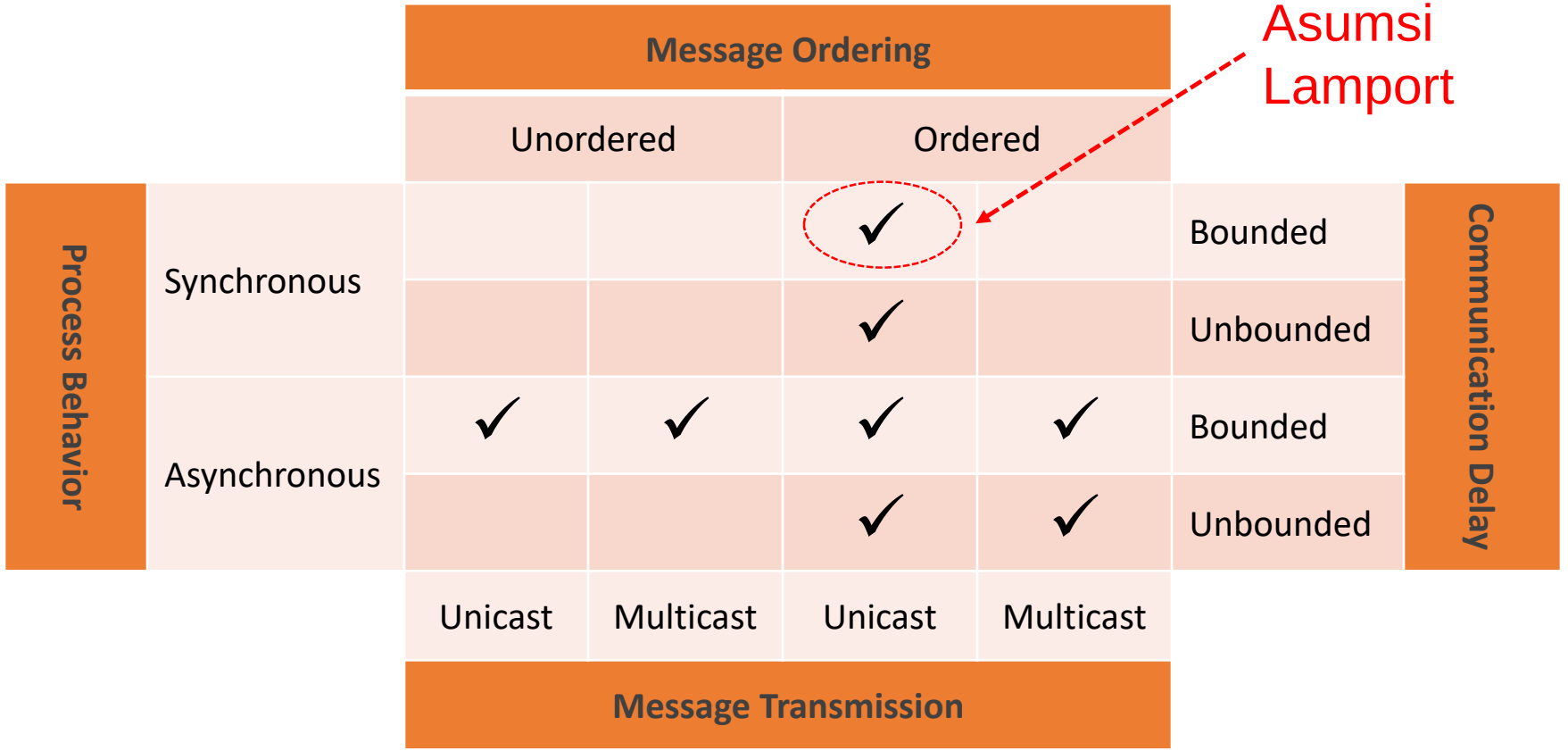
Penanganan Kegagalan Byzantine

- Kegagalan Bizantium sulit untuk dideteksi dan ditutupi
  - Server mungkin menghasilkan output yang seharusnya tidak pernah dihasilkan (karena perhitungan yang salah)
  - Lebih buruk lagi, server mungkin bekerja jahat (baik secara sendiri-sendiri atau bersama-sama dengan server lain) untuk menghasilkan jawaban yang sengaja salah

- Kegagalan Bizantium biasanya dapat ditangani dengan menggunakan protokol perjanjian (agreement atau konsensus)

Konsensus Dalam Sistem yang Salah

- Mencapai konsensus dalam sistem terdistribusi hanya dimungkinkan dalam keadaan berikut:



Masalah Jenderal Bizantium

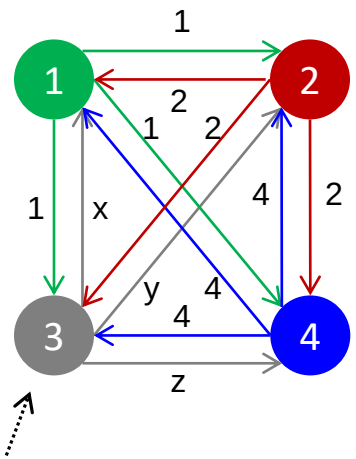
Byzantine Generals Problem

- Asumsi Lamport:
  - Terdapat *n* proses (atau jenderal)
  - Setiap proses *i* akan mengirimkan suatu value v_i (yaitu “attack” atau “wait” atau sesuatu yang lain) ke setiap proses lainnya
  - Ada paling banyak *m* proses yang gagal (faulty atau *traitors*)
  - Setiap proses akan membangun suatu vector V dengan panjang n
    - Jika proses *i* non-faulty, V[*i*] = v*i*
    - Jika tidak, V[*i*] tidak didefinisikan

Masalah Jenderal Bizantium

- **Kasus I: n = 4 dan m = 1**

Langkah1: Setiap proses mengirimkan value-nya ke proses lain



Proses Faulty

Langkah2: Setiap proses mengumpulkan value yang diterima dalam suatu vektor

1 Got(1, 2, x, 4)
2 Got(1, 2, y, 4)
3 Got(1, 2, 3, 4)
4 Got(1, 2, z, 4)

Langkah3: Setiap proses melewatkan vektornya ke setiap proses lain

1 Got

(1, 2, y, 4)
(a, b, c, d)
(1, 2, z, 4)

2 Got

(1, 2, x, 4)
(e, f, g, h)
(1, 2, z, 4)

4 Got

(1, 2, x, 4)
(1, 2, y, 4)
(i, j, k, l)

Masalah Jenderal Bizantium

Langkah 4:

- Setiap proses memeriksa elemen ke-*i* dari setiap vektor yang baru saja diterimanya
- Jika suatu value mempuyai *mayaritas*, value tersebut diletakkan ke dalam vektor hasil
- Jika value tidak mempunyai mayoritas, elemen yang bersesuaian dari vektor hasil ditandai UNKNOWN

Allgoritma mendeteksi traitor!

Vektor Hasil:
(1, 2, UNKNOWN, 4)

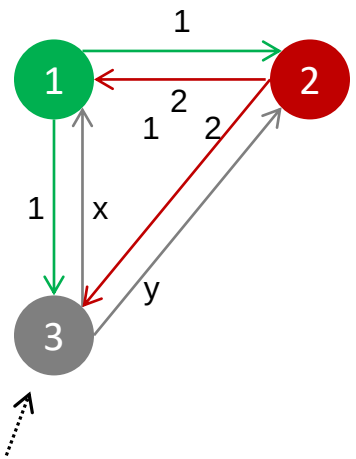
Vektor Hasil:
(1, 2, UNKNOWN, 4)

Vektor Hasil:
(1, 2, UNKNOWN, 4)

Masalah Jenderal Bizantium

- **Kasus II: n = 3 dan m = 1**

Langkah1: Setiap proses mengirimkan valuenya ke proses lain



Proses Faulty

Langkah2: Setiap proses menghimpun value yang diterima dalam vektor

1 Got(1, 2, x)
2 Got(1, 2, y)
3 Got(1, 2, 3)

Langkah3: Setiap proses melewatkan vektornya ke setiap proses lain

1 Got

(1, 2, y)
(a, b, c)

2 Got

(1, 2, x)
(d, e, f)

Masalah Jenderal Bizantium

Langkah 4:

- Setiap proses memeriksa elemen ke-*i* dari setiap vektor yang baru saja diterimanya
- Jika suatu value mempuyai *mayaritas*, value tersebut diletakkan ke dalam vektor hasil
- Jika value tidak mempunyai mayoritas, elemen yang bersesuaian dari vektor hasil ditandai UNKNOWN

Kesimpulan Tentang Byzantine Generals Problem

- *Lamport et al.* (1982) telah membuktikan bahwa di dalam suatu system dengan ***m*** proses yang gagal, suatu kesepakatan (agreement) dapat dicapai hanya jika ***2m+1*** proses yang berfungsi dengan benar hadir, sehingga total terdapat ***3m+1***
  - **BGP (n, m)** dapat dipecahkan **iff n ≥ (3m + 1)**
  - Dengan kata lain, suatu konsensus hanya mungkin dicapai jika masih terdapat lebih dari dua-per-tiga (2/3) proses bekerja dengan benar.