

# SIFAT-SIFAT *ROLLBACK RECOVERY* MENGGUNAKAN *UNCOORDINATED CHECKPOINTING* BERBASIS *CAUSALITY STRENGTH*

Junianto Sesa\*

## Abstract

Fault tolerance approach is the most popular computing application on computer devices in which depends on checkpoint uncoordinated. This alternative approach is based on checkpoint uncoordinated and logging message requiring all records, imposing works, memories and overhead becomes significant to communication. Recent studies have found that many applications on computer are send-determinism which can possibly design a new fault tolerance protocol. Thus, this research uses checkpoint uncoordinated protocol based causality strength, a send-determinism feature to record one part of the messages without restarting the process systematically when the error occurs. By drawing the protocol and proving its validity are required as the effective methods of this research. With this alternative approach, the protocol can functionally work where the only small portion of the message is recorded and domino effect does not occur.

**Keywords :** Causality Strength, Domino Effect, Rollback Recovery, Uncoordinated Checkpointing

## Abstrak

Pendekatan toleransi kesalahan yang paling populer untuk aplikasi komputasi pada perangkat komputer bergantung pada *checkpoint uncoordinated*. Alternatif pendekatan tersebut berdasarkan pada *checkpoint uncoordinated* dan *logging* pesan mengharuskan pencatatan semua pesan, memaksakan pekerjaan memori/penyimpanan tinggi dan overhead yang signifikan pada komunikasi. Baru-baru ini telah diamati bahwa banyak aplikasi pada komputer bersifat *send-determinism* yang memungkinkan untuk mendesain protokol toleransi kesalahan baru. Sehingga penelitian ini menggunakan protokol *checkpoint uncoordinated* berbasis *causality strength* yang bersifat *send-determinism* yang hanya mencatat satu bagian dari pesan dan tidak perlu me-restart secara sistematis semua proses ketika kegagalan terjadi. Untuk menunjukkan bahwa penelitian ini berjalan sesuai dengan metode yang digunakan yaitu dengan menggambarkan protokol dan membuktikan kebenarannya. Dengan menggunakan pendekatan tersebut, dapat ditunjukkan bahwa protokol ini benar-benar berhasil dimana hanya mencatat sebagian kecil dari pesan dan tidak terjadi efek domino.

**Kata kunci :** *Causality Strength*, Efek Domino, *Rollback Recovery*, *Uncoordinated Checkpointing*

## 1. Pendahuluan

Perkembangan teknologi yang sangat pesat di era globalisasi saat ini telah memberikan banyak manfaat dalam kemajuan diberbagai aspek, Hampir semua aspek kegiatan manusia saat ini bergantung pada komputer. Seperti dalam kegiatan perkantoran, perbankan, perdagangan, pendidikan, bisnis, dan sebagainya. Namun demikian, dengan berkembangnya teknologi tentu tidak lepas dari beberapa kekurangan yang terus menjadi perhatian peneliti dibidang komputasi. Salah satu yang sering menjadi perhatian peneliti yaitu masalah sistem terdistribusi.

Sistem terdistribusi adalah suatu sistem jaringan komputer yang saling terhubung sedemikian sehingga seolah-olah menjadi sebuah komputer yang digunakan oleh satu pengguna. Sebuah sistem yang terdiri dari beberapa komponen software atau hardware yang saling bebas dimana berkomunikasi dan berkoordinasi melalui *message* baik terhubung maupun tidak terhubung yang

\*Departemen Matematika, Fakultas Matematika dan Ilmu Pengetahuan Alam  
Pascasarjana Universitas Hasanuddin  
[jhunzea@gmail.com](mailto:jhunzea@gmail.com)

terlihat satu kesatuan dan dirancang untuk menghasilkan fasilitas komputasi terintegrasi. Perintah yang efisien untuk *Pesan* sederhana dalam sistem komputasi *mobile*. Perintah ini membutuhkan sumber daya minimal pada *hosts mobile* dan jaringan nirkabel. Perintah ini bersifat terskala dan dapat dengan mudah menangani perubahan dinamis dan jumlah *hosts seluler* yang berpartisipasi dalam sistem serta menawarkan penundaan yang tidak perlu, kinerja *Pesan* yang rendah dan pengeluaran biaya yang optimal [1].

Perhatian utama dalam sistem terdistribusi adalah memastikan tingkat keandalan dan ketersediaan yang telah ditentukan. Sistem ini cenderung gagal karena tingginya kompleksitas. Oleh karena itu *fault tolerance* menjadi isu utama diatasi dalam merancang sistem ini [6]. Terdapat model yang dirancang untuk aplikasi serial atau solusi berbasis *coordinated checkpoint*. Model ini di skenario berdasarkan *uncoordinated checkpointing* dikombinasikan dengan *message logging*. Diperkenalkan dua poin kunci untuk meminimalkan *fault tolerance overhead* untuk aplikasi paralel, yang pertama adalah penggunaan *factor* mewakili hubungan ketergantungan antar proses, kedua adalah menggunakan interval checkpoint khusus untuk masing-masing proses. Percobaan menunjukkan bahwa model tersebut berkinerja dengan baik seperti penelitian sebelumnya untuk aplikasi serial atau *coordinated checkpointing* sambil menjalankan aplikasi paralel dengan menggunakan *uncoordinated checkpoint* dikombinasikan dengan *messages logging*, Model interval checkpoint ini secara efektif meminimalkan *overhead*. Bahkan, kesalahan prediksi *overhead* lebih kecil dari 5% untuk semua aplikasi yang diuji [4].

## 2. Konsep Dasar Pesan

Lamport mendefinisikan hubungan klasik “terjadi sebelum”, dilambangkan dengan " $\rightarrow$ ".

**Definisi 2.1** (Yoshida, 2001)

Dua kejadian  $a$  dan  $b$  memiliki relasi  $a \rightarrow b$  jika dan hanya jika memenuhi salah satu syarat berikut :

1. Jika  $a$  dan  $b$  kejadian pada objek yang sama dan  $a$  terjadi sebelum  $b$ .
2. Jika  $a$  adalah kejadian pengiriman *Pesan* pada satu objek dan  $b$  adalah kejadian menerima *Pesan* yang sama dari objek yang lain.
3. Terdapat kejadian  $c$  sedemikian sehingga  $a \rightarrow c$  dan  $c \rightarrow b$

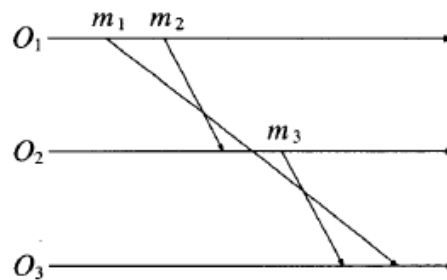
Dua kejadian  $a$  dan  $b$  dikatakan bersamaan jika  $a \rightarrow b$  dan  $b \rightarrow a$  dan dinotasikan dengan  $a || b$ .

Takaichi Yoshida mendefinisikan peristiwa pemesanan kausal yang mana terkait dengan yang terjadi sebelum hubungan. Peristiwa pemesanan kausal didefinisikan sebagai berikut:

**Definisi 2.2** (Yoshida, 2001)

Jika  $send(m_1) \rightarrow send(m_2)$ , dan  $m_1, m_2$  memiliki tujuan objek yang sama, maka  $accept(m_1) \rightarrow accept(m_2)$  harus dipenuhi. Dimana  $send(m)$  dinotasikan sebagai kejadian pengiriman dari *Pesan*  $m$ , dan  $accept(m)$  dinotasikan sebagai kejadian penerimaan *Pesan*  $m$ .

Definisi ini mengatakan bahwa jika kejadian pengiriman *Pesan*  $m_1$  "terjadi sebelum" peristiwa pengiriman *Pesan*  $m_2$ , objek tujuan dari kedua *Pesan* ini harus menerima *Pesan*  $m_1$  sebelum *Pesan*  $m_2$ .



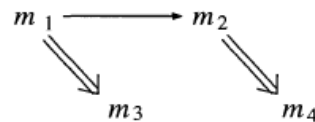
**Gambar 2.1** Kejadian konvensional pengurutan kausal

Mengirim *Pesan* berarti menghubungkan *Pesan* ke antrian dari tujuan objek, sedangkan menerima *Pesan* berarti mengambil *Pesan* dari antrian *Pesan* ke objek tujuan dan mulai memprosesnya. Objek tidak dapat mengontrol urutan penerimaan, bagaimanapun kejadiannya.

Karena tidak dapat melihat kejadian internal apapun dalam suatu objek, maka dibatasi dari peristiwa eksternal, yaitu peristiwa yang sesuai untuk pengiriman, dan penerimaan *Pesan*. Sehingga didefinisikan hubungan *causality strength* antar *Pesan* sebagai berikut:

**Definisi 2.3** (Yoshida, Takaichi. 2001)

1. Jika  $\text{send}(m_2)$  terjadi segera setelah  $\text{send}(m_1)$  pada objek yang sama maka  $m_1 \rightarrow m_2$ .
2. Jika  $\text{send}(m_2)$  terjadi segera setelah  $\text{accept}(m_1)$  pada objek yang sama maka  $m_1 \Rightarrow m_2$ .
3. Jika  $m_1 \rightarrow m_3$  dan  $m_3 \rightarrow m_2$  maka  $m_1 \rightarrow m_2$
4. Jika  $m_1 \Rightarrow m_3$  dan  $m_3 \Rightarrow m_2$  maka  $m_1 \Rightarrow m_2$



**Gambar 2.2** Struktur pohon relasi *causality strength*

**Definisi 2.4** Eksekusi yang valid dalam pengiriman pesan deterministik adalah eksekusi di mana setiap proses mengirim urutan pesan yang valid dan kausalitas diperhitungkan dalam pengiriman pesan.

**Definisi 2.5** (Interval *checkpoint* dan *Round*): Interval *checkpoint*  $I_p^i$  adalah himpunan peristiwa antara  $C_p^i$  dan  $C_p^{i+1}$ .  $i$  adalah nomor *Round* dari interval *checkpoint*.

**Definisi 2.6** Misalkan  $R$  menjadi himpunan pesan yang dirollback dari proses  $P$ .  $m \in R$  jika dan hanya jika  $\text{receive}(m) \in V_p$  dan  $\text{receive}(m) \in I_p^y$  dengan  $y \geq x$ . Dilambangkan dengan  $R|(q;p)$  subhimpunan pesan dalam  $R$  yang diterima di saluran  $(q;p)$ .

**Definisi 2.7** Misalkan  $\mathcal{L}$  merupakan himpunan pesan yang dilog dari suatu proses  $q.m \in \mathcal{L}$  jika dan hanya jika untuk proses  $p$ ,  $receive(m) \in I_p^x$  dan  $receive(m) \in I_p^y$  dengan  $y \geq x$ . Dilambangkan dengan  $\mathcal{L}|(q;p)$  subhimpunan pesan di  $\mathcal{L}$  yang dikirim pada saluran  $(q;p)$ .

**Definisi 2.8** Misalkan  $\ell$  menjadi himpunan pesan yang diputar ulang dari proses  $p$ .  $m \in \ell$  jika dan hanya jika  $send(m) \in V_p$  dan  $send(m) \in I_p^y$  dengan  $y \geq x$ . Dilambangkan dengan  $\ell|(p;q)$  subhimpunan pesan dalam  $\ell$  yang dikirim pada saluran  $(p;q)$ . Perlu dicatat bahwa  $\mathcal{L} \cap \ell = \emptyset$ . *Date* saat ini pada proses  $p$  dilog dengan  $date_p(p)$ . Tanggal dari suatu peristiwa  $e \in V_p$  adalah  $date_p(p)$  ketika  $e$  terjadi.

**Definisi 2.9** Misalkan  $\Theta$  menjadi kumpulan pesan *OrStan*, dan pesan  $m$  di saluran  $(p;q)$ .  $m \in \Theta$  jika dan hanya jika  $date(send(m)) > date_p(p)$  dan  $date(receive(m)) < date_p(q)$ . Dilambangkan dengan  $\Theta|(p;q)$  subhimpunan pesan di  $\Theta$  diterima di saluran  $(p;q)$ . Perlu dicatat bahwa  $\Theta \cap R = \emptyset$  dan  $\Theta \subset \ell$ .

**Definisi 2.10** Misalkan  $St(p)$  merupakan Stage dari proses  $p$ . Misalkan  $V_p^i$  menjadi rangkaian kejadian pada proses  $p$  dalam Stage  $i$ . Kejadian  $e \in V_p^i$  jika dan hanya jika  $St(p) = i$  ketika  $e$  terjadi.

**Definisi 2.11** Stage pesan  $St_m(m) = k$  di mana  $send(m) \in V_p^k$ .

**Definisi 2.12** Misalkan  $\Theta_{=k}$  adalah himpunan pesan tidak stabil dalam Stage  $k$ . Pesan  $m \in \Theta_{=k}$  jika dan hanya jika  $St_m(m) = k$ . Kami mendefinisikan  $\Theta_{\leq k}$  (atau operator lain) sebagai himpunan proses yang *OrStan* dalam Stage yang lebih rendah dari atau sama dengan  $k$ .

**Definisi 2.13** Misalkan  $R_{=k}$  adalah himpunan pesan yang di *rollback* dalam Stage  $k$ .  $m \in R_{=k}$  jika dan hanya jika  $receive(m) \in V_p^k$ .

### 3. Rollback Recovery

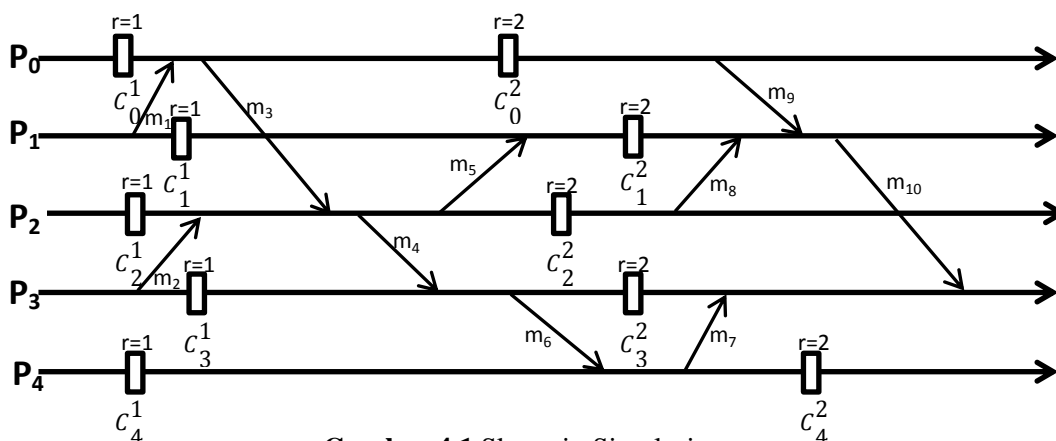
*Rollback recovery* menghidupkan kembali sistem ke keadaan yang konsisten setelah mengalami kegagalan, membuat toleransi kesalahan dengan menyimpan secara berkala keadaan proses selama eksekusi bebas kegagalan, dan memperlakukan aplikasi sistem terdistribusi sebagai kumpulan proses yang berkomunikasi melalui jaringan.

Ketergantungan dapat memaksa beberapa proses yang tidak gagal dilepaskan, fenomena ini disebut "efek domino". Jika setiap proses mengambil *checkpoint* secara bebas, maka sistem tidak dapat menghindari efek domino, skema ini disebut uncoordinated checkpoint.

- ❖  $C_i^k$  : *Checkpoint* secara umum ke  $k$  pada proses  $P_i$ .
- ❖  $C_i^0$  : Sebuah proses  $P_i$  dengan mengambil sebuah *checkpoint*  $C_i^0$  sebelum mulai dieksekusi.

## 4. Hasil dan Pembahasan

Protokol yang tidak terkoordinasi akan mengalami efek domino karena proses perlu di rollback dan efek domino dapat memaksa beberapa proses untuk memulai kembali dari awal eksekusi. apabila protokol tidak rollback proses orphan, maka proses tersebut tidak mengalami efek domino. Namun, karena protokol rollback proses pengiriman pesan yang orphan, maka dapat menciptakan efek domino. Pesan yang dikirim kembali sebelum checkpoint memaksa pengirim untuk rollback ke checkpoint sebelumnya. Untuk mengidentifikasi pesan-pesan ini, protokol menggunakan *round*.



Gambar 4.1 Skenario Simulasi

Jika proses  $P_1$  gagal, ia akan kembali ke *checkpoint* terakhirnya  $C_1^2$  (2 sebagai nomor *checkpoint*). Proses  $P_0$  dan  $P_2$  harus *rollback* untuk memutar ulang pesan  $m_8$  dan  $m_9$ . Pesan  $m_8$  dan  $m_9$  merupakan pesan yang dirollback. Ketika  $P_1$  rollback, pesan  $m_{10}$  menjadi orphan. Namun, proses  $P_3$  tidak perlu melakukan rollback karena  $m_{10}$  akan diputar ulang terlepas dari urutan penerimaan  $m_8$  dan  $m_9$  selama eksekusi ulang (kausal kuat).

Gambar 4.2 berisi tentang protokol yang dieksekusi oleh proses aplikasi. Gambar 4.3 menyajikan proses khusus, yang disebut proses *recovery*, yang digunakan untuk *recovery*.

### Local Variables:

- 1:  $Status_i \leftarrow Running$  // Status dari proses, Running, Blok atau RolledBack
- 2:  $TS_i \leftarrow 1; Round_i \leftarrow 1; Stage_i \leftarrow 1$  // TS dari kejadian saat ini, nomor Round dan nomor stage pada proses  $P_i$
- 3:  $NonAck_i \leftarrow \emptyset$ ; // daftar pesan yang dikirim oleh proses  $P_i$  dan belum dijawab
- 4:  $Logs_i \leftarrow \emptyset$ ; // daftar pesan yang dicatat pada  $P_i$
- 5:  $SPR_i \leftarrow [\perp, \dots, \perp]$  // Data tentang pesan (sent per round).  $SPR_i[Round_{send}] \cdot TS$  adalah TS  $P_i$  pada awal dari  $Round_{send}$ .
- 6:  $RPS_i \leftarrow [\perp, \dots, \perp]$  // Data tentang pesan (received per stage)
- 7:  $RL_i \leftarrow [\perp, \dots, \perp]$  //  $RL_i[j] \cdot round$  di mana  $P_j$  harus *rollback* setelah kegagalan;  $RL_i[j] \cdot TS$  adalah *TS* yang sesuai
- 8:  $OrphCount_i \leftarrow [\perp, \dots, \perp]$  //  $OrphCount_i[stage]$  adalah jumlah proses yang akan mengirim ulang pesan *orphan* ke  $P_i$  dalam stage  $stage$
- 9:  $OrphStages_i \leftarrow \emptyset$ ; // Stage di mana pesan *orphan* telah diterima oleh  $P_i$
- 10:  $LogStages_i \leftarrow \emptyset$ ; // Stage di mana  $P_i$  telah mencatat pesan untuk diputar ulang

## Junianto Sesa

```

11:  $ReplayLogged_i \leftarrow [1, \dots, 1]$  //  $ReplayLogged_i[stage]$  adalah daftar pesan yang
// dicatat oleh  $P_i$  untuk diputar ulang dalam stage  $stage$ 
12:
13: Upon sending message  $msg$  to  $P_j$ 
14:   wait until  $Status_i = Running$ 
15:    $TS_i \leftarrow TS_i + 1$ 
16:    $NonAck_i \leftarrow NonAck_i \cup [ (P_j, Round_i, TS_i, msg)$ 
17:    $Send(msg, TS_i, Round_i, Stage_i)$  to process  $P_j$ 
18:
19: Upon receiving ( $msg, TS_{send}, Round_{send}, Stage_{send}$ ) from  $P_j$ 
20: if  $TS_{send} > RPS_i[Stage_i][j].TS$  then //  $msg$  diterima untuk pertama kalinya
21:   if  $Round_{send} < Round_i$  then // dicatat
22:      $Stage_i \leftarrow Max(Stage_i, Stage_{send} + 1)$ 
23:   else
24:      $Stage_i \leftarrow Max(Stage_i, Stage_{send})$ 
25:      $TS_i \leftarrow TS_i + 1$ 
26:      $RPS_i[Stage_i][j].TS \leftarrow TS_{send}$ 
27:      $Send(Ack, Round_i, TS_{send})$  to  $P_j$ 
28:     Deliver  $msg$  to the application
29:   else if  $\exists stagesuch\ that\ TS_{send} = RPS_i[stage][j].TS$  then // pesan terakhir dari satu
stage
30:      $OrphCount_i[stage] \leftarrow OrphCount_i[stage] - 1$ 
31:     if  $OrphCount_i[stage] = 0$  then // Proses menerima semua pesan duplikat untuk
stage ini
32:        $Send(NoOrphanStage, stage)$  to the recovery process
33:
34: Upon receiving ( $Ack, Round_{recv}, TS_{send}$ ) from  $P_j$ 
35:   Remove ( $P_j, Round_{send}, TS_{send}, msg$ ) from  $NonAck_i$ 
36:   if  $Round_{send} < Round_{recv}$  then // mencatat
37:      $Logs_i \leftarrow Logs_i \cup [ (P_j, Round_{send}, TS_{send}, Stage_{send}, Round_{recv}, msg)$ 
38:   else
39:      $SPR_i[Round_{send}][P_j].round_{recv} \leftarrow Round_{recv}$ 
40:
41: Upon checkpoint
42:   Save ( $Round_i, ImagePs_i, ReceivedPerStage_i, SentPerRound_i, Logs_i, Stage_i, TS_i$ ) on
stable storage
43:    $Round_i \leftarrow Round_i + 1$ 
44:    $Stage_i \leftarrow Stage_i + 1$ 
45:    $SPR_i[Round_i].TS \leftarrow TS_i$ 
46:
47: Upon failure of process  $P_i$ 
48:   Get last ( $Round_i, ImagePs_i, RPS_i, SPR_i, Logs_i, Stage_i, TS_i$ ) from stable storage
49:   Restart from  $ImagePs_i$ 
50:    $Status_i \leftarrow RolledBack$ 
51:    $Send(Rollback, Round_i, TS_i)$  to all application processes and to the recovery
process
52:    $Send(SPR_i)$  to the recovery process
53:
54: Upon receiving ( $Rollback, Round_{rb}, TS_{rb}$ ) from  $P_j$ 
55:    $Status_i \leftarrow Blocked$ 
56:    $Send(SPR_i)$  to the recovery process
57:
58: Upon receiving ( $RL_{recov}$ ) from the recovery process
59:   if  $RL_{recov}[i].round < Round_i$  then // yang akan di rollback
60:     Get ( $RL_{recov}[i].round, ImagePs_i, RPS_i, SPP_i, Logs_i, Stage_i, TS_i$ ) from stable
storage
61:      $Status_i \leftarrow RolledBack$ 

```

## Junianto Sesa

```

62:   for all stagesuch that  $RPS_i[stage][j].TS > RL_{recov}[j].TS$  do // Mencari stage
      yang memiliki pesan orphan
63:    $OrphStages_i \leftarrow OrphStages_i \cup stage$ 
64:    $OrphCount_i[stage] \leftarrow OrphCount_i[stage] + 1$ 
65:   for all  $(P_j, Round_{send}, TS_{send}, Stage_{send}, Round_{recv}, msg)$  in  $Logs_i$  such that  $Round_{recv} \geq$ 
       $RL_{recov}[j].round$  do //Looking for logged messages to be replayed
66:      $LogStages_i \leftarrow LogStages_i \cup Stage_{send}$ 
67:      $ReplayLogged_i[Stage_{send}] \leftarrow ReplayLogged_i[Stage_{send}] \cup (P_j, Round_{send}, TS_{send},$ 
       $Stage_{send}, Round_{recv}, msg)$ 
68:     Send  $(Orphan, Status_i, Stage_i, OrphStages_i, LogStages_i)$  to the recovery process
69:
70: Upon receiving  $(ReadyStage, Stage)$  from the recovery process
71:   if  $ReplayLogged_i[Stage] \neq \emptyset$  then // Kirim pesan yang sudah dicatat jika ada
72:      $Replay\ msgs \in ReplayLogged_i[Stage]$ 
73:     if  $(Status_i = RolledBack \wedge Stage_i = Stage + 1) \vee (Status_i = Blocked \wedge Stage_i = Stage)$ 
      then
74:        $Status_i = Running$ 

```

**Gambar 4.2** Algoritma Proses

### Local Variables:

```

1:  $DependencyTable \leftarrow [1, \dots, 1]$  //DependencyTable[j][Roundsend][k].roundrecv adalah SPR dari
      proses j
2:  $RolledBackStage \leftarrow \emptyset$  // RolledBackStage[stage] berisi daftar proses
      rollback yang diblok dalam stage stage
3:  $BlockedStage \leftarrow \emptyset$  // berisi daftar proses yang tidak Rollback dan pesan
      yang dicatat diblok dalam stage stage
4:  $NbOrphanStage \leftarrow \emptyset$  // NbOrphanStage[stage] adalah jumlah proses yang
      memiliki setidaknya satu pesan orphan dalam stage
      stage
5:
6: Upon receiving  $(Rollback, Round_{rb}, TS_{rb})$  from  $P_j$ 
7:    $RL_{recov}[j].round \leftarrow Round_{rb}$ 
8:    $RL_{recov}[j].TS \leftarrow TS_{rb}$ 
9:   wait until  $DependencyTable$  is complete
10:   $RL_{tmp} \leftarrow [1, \dots, 1]$ 
11:  repeat
12:    for all  $P_j$  such that  $RL_{tmp}[j] \neq RL_{recov}[j]$  do
13:       $RL_{tmp}[j] \leftarrow RL_{recov}[j]$ 
14:      for all  $P_k$  such that  $DependencyTable_i[k][Round_{send}][j].Round_{recv} \geq$ 
       $RL_{recov}[j].round$  do
15:         $RL_{recov}[k].round \leftarrow \min(RL_{recov}[k].round, Round_{send})$  // Jika Roundsend
      diambil,  $RL_{recov}[k].TS$  juga diperbaharui
16:      until  $RL_{recov} = RL_{tmp}$ 
17:      Send  $RL_{recov}$  to all application processes
18:
19: Upon receiving  $SPR_j$  from  $P_j$ 
20:    $DependencyTable[j] \leftarrow SPR_j$ 
21:
22: Upon receiving  $(OrphanNotification, Status_j, Stage_j, OrphStages_j, LogStages_j)$  from  $P_j$ 
23:   if  $Status_j = Rolled-back$  then
24:      $RolledBackStage[Stage_j] \leftarrow RolledBackStage[Stage_j] \cup P_j$ 
25:   else
26:      $BlockedStage[Stage_j] \leftarrow BlockedStage[Stage_j] \cup P_j$ 
27:   for all stage  $\in LogStages_j$  do
28:      $BlockedStage[stage] \leftarrow BlockedStage[stage] \cup P_j$ 

```

## Junianto Sesa

```

29:   for all stage ∈ OrphStagesj do
30:     NbOrphanStage[stage] ← NbOrphanStage[stage] + 1
31:   if OrphanNotification has been received from all application processes then
32:     Start NotifyStages
33:
34:   Upon receiving (NbOrphanStage, Stage) from Pj
35:     NbOrphanStage[Stage] ← NbOrphanStage[Stage] - 1
36:     Start NotifyStages
37:
38:   NotifyStages
39:   for all stagesuch that ∃stage' ≤ stage ∧ NbOrphanStage[stage'] > 0 do //
    pemberitahuan untuk stage yang tidak bergantung pada pesan orphan
40:     Send (ReadyStage, stage) to all processes in BlockedStage[stage]
41:     Send (ReadyStage, stage) to all processes in RolledBackStage[stage+ 1]

```

**Gambar 4.2** Algoritma Recovery

**Lemma 4.1** Misal diberikan 2 proses  $p$  dan  $q$ ,  $p$  mengirim pesan di saluran  $(p; q)$ . Jika proses  $q$  rollback maka  $R|(p, q) \setminus \ell|(p; q) \in \mathcal{L}|(p; q)$ .

**Bukti :** Misal  $m \in R|(p; q)$  dan  $C_q^i$  checkpoint kembali pada proses  $q$ . Karena  $m \in R|(p; q)$  dan  $receive(m) \in I_q^k$  di mana  $k \geq i$ . Jika  $m \notin \ell|(p; q)$  dan  $send(m) \in I_p^j$  di mana  $j < k$ , maka  $m \in \mathcal{L}$  berdasarkan definisi pesan yang dilog.

**Proposisi 4.1** Misalkan  $m_i$  dan  $m_j$  merupakan dua pesan. Jika  $m_i \rightarrow m_j$  maka  $St_m(m_i) \leq St_m(m_j)$ .

**Proposisi 4.2** Misalkan  $m_i$  dan  $m_j$  merupakan dua pesan. Jika  $m_i \Rightarrow m_j$  maka  $St_m(m_i) \leq St_m(m_j)$ .

**Proposisi 4.3** (Mengirim kondisi pesan yang diputar ulang): Misalkan  $m_j \in \ell$ .  $m_j$  dikirim jika dan hanya jika  $\Theta_{<St_m(m_j)} = \emptyset$ .

**Proposisi 4.4** (Mengirim kondisi pesan yang tidak diputar ulang): Misalkan  $m_j \notin \ell$ .  $m_j$  dikirim jika dan hanya jika  $\Theta_{\leq St_m(m_j)} = \emptyset$ .

Sekarang diberikan kondisi untuk memutar ulang pesan yang dirollback dari stage  $x$ :

**Lemma 4.2**  $\forall m_i \in R_{=x}$ ,  $m_i$  dikirim kembali jika, dan hanya jika,  $\Theta_{<x} = \emptyset$ .

**Bukti :** Berdasarkan Lemma 4.1 jika  $m_i \in R$  maka  $m_i \in \ell$  atau  $m_i \in \mathcal{L}$ .

- 1) Jika  $m_i \in \ell$ ,  $St_m(m_i) \leq x$  (Baris 24 Gambar 4.18),  $m_i$  dikirim jika  $\Theta_{<x} = \emptyset$ . menurut Proposisi 3.



- 2) Jika  $m_i \in \mathcal{L}$ ,  $m_i$  dikirim jika  $\Theta_{\leq St_m(m_i)} = \emptyset$ . menurut Proposisi 4.4. Karena  $m_i \in \mathcal{L}$  dan  $m_i \in R_{=x}$  maka  $St_m(m_i) < x$  (sebagai stage penerima selalu lebih besar dari salah satu pesan yang terekam yang diterima: baris 21-22 gambar 4.18). Jadi  $m_i$  dikirim jika  $\Theta_{<x} = \emptyset$ .

**Lemma 4.3** Semua pesan yang dirollback akhirnya dikirim ulang.

**Bukti :** Misalkan  $\Theta_{\min\_stage}$  adalah himpunan pesan *orphan* dengan stage terendah. Jadi  $\forall m_i \in R_{<\min\_stage}$ ,  $m_i$  dikirim menurut Lemma 2. Dan  $\forall m_j \in \Theta_{\min\_stage}$ ,  $m_j$  dikirim sesuai dengan Proposisi 3. Kemudian  $\Theta_{\min\_stage} = \emptyset$ ; dan  $\forall m_i \in R_{<\min\_stage+1}$ ,  $m_i$  terkirim. Dan seterusnya sampai  $R = \emptyset$  dan  $\Theta = \emptyset$ .

Sekarang akan dibuktikan bahwa kausalitas diperhitungkan pada pengiriman pesan. Kausalitas rusak jika untuk dua pesan  $m_i$  dan  $m_j$ ,  $m_i \rightarrow m_j$ ,  $m_j$  diterima sebelum  $m_i$  selama *recovery*. Jika semua proses kembali dalam keadaan yang tidak tergantung pada  $m_i$ , kondisi ini dijamin karena  $m_j$  tidak dapat dikirim sebelum  $m_i$  diputar kembali. Kami pertama kali menunjukkan bahwa, jika aplikasi dalam keadaan di mana  $m_i$  dan  $m_j$  keduanya dapat dikirim, terdapat pesan *orphan* di kausalitas antara  $m_i$  dan  $m_j$

**Lemma 4.4** Misalkan dua pesan  $m_i$  dan  $m_{i+x}$ ,  $x \geq 1$ , sedemikian sehingga  $m_i \rightarrow m_{i+x}$ . Jika  $m_i$  dan  $m_{i+x}$  keduanya dapat dikirim, maka  $\exists m \in \Theta$  sedemikian sehingga  $m_i \rightarrow \dots \rightarrow m \rightarrow \dots \rightarrow m_{i+x}$ .

**Bukti :** Karena  $m_i$  dan  $m_{i+x}$  keduanya dapat dikirim kemudian  $m_i \in R \cup \Theta$ .  $m_{i+x}$  dapat dikirim jika  $m_{i+(x-1)}$  telah diterima. Pengirim  $m_{i+(x-1)}$  rollback maka  $m_{i+(x-1)} \in \Theta$  atau bukan. Jika  $m_{i+(x-1)} \notin \Theta$ ,  $m_{i+(x-2)}$  diterima. Kemudian  $m_{i+(x-2)}$  merupakan *orphan* atau bukan. Dan seterusnya sampai  $m_{i+1}$  dapat dikirim jika  $m_i$  adalah *orphan* atau bukan. Karena  $m_i \in R \cup \Theta$ , dan pengirim  $m_{i+1}$  tidak rollback (jika tidak akan berhenti di  $m_{i+1}$ ), maka  $m_i \in \Theta$ .

Selanjutnya hubungan antara stage suatu pesan *orphan* dan yang diputar ulang.

**Lemma 4.5** Misalkan  $m_i$  dan  $m_j$  merupakan dua pesan sedemikian sehingga  $m_i \rightarrow m_j$ . Jika  $m_i \in \Theta$  dan  $m_j \in \ell$  maka  $St_m(m_i) < St_m(m_j)$ .

**Bukti :** Jika  $m_i \rightarrow m_j$  menurut Proposisi 1, maka  $St_m(m_i) \leq St_m(m_j)$ . Untuk membuktikan bahwa  $St_m(m_i) < St_m(m_j)$  akan dibuktikan bahwa  $St_m(m_i) = St_m(m_j)$  mengarah ke kontradiksi. Jadi diasumsikan bahwa  $St_m(m_i) = St_m(m_j)$ . Menurut baris 21 dan 44 pada Gambar 4.18, tidak ada checkpoint dan tidak ada pesan yang masuk pada jalur kausalitas dari  $m_i$  ke  $m_j$  (karena kedua kejadian ini adalah satu - satunya yang meningkatkan stage). Kemudian sesuai dengan baris 10 sampai 16 pada Gambar 4.19, jika  $m_j \in \ell$ ,  $m_i \in R$ . Ini tidak mungkin karena  $m_i \in \Theta$  dan  $\Theta \cap R = \emptyset$ . Sehingga dibuktikan bahwa pesan yang tergantung pada pesan *orphan* tidak dapat dikirim sebelum pesan *orphan* dikirim.

**Lemma 4.6** Jika  $m_i$  dan  $m_j$  merupakan dua pesan sedemikian sehingga  $m_i \rightarrow m_j$  dan  $m_i \in \Theta$ , maka  $m_j$  tidak dapat dikirim sebelum  $m_i$  dikirim.

**Bukti :** Jika  $m_i \in \ell$ ,  $m_j$  dikirim jika dan hanya jika  $\Theta_{<St_m(m_j)} = \emptyset$ . (Proposisi 4.3). Jika  $m_j \notin \ell$ ,  $m_j$  dikirim jika dan hanya jika  $\Theta_{\leq St_m(m_j)} = \emptyset$ . (Proposisi 4.4).  $m_i \in \Theta_{=St_m(m_i)}$ . Karena  $m_i \rightarrow m_j$ ,  $St_m(m_j) \geq St_m(m_i)$  menurut Lemma 4.1.

- 1) Jika  $St_m(m_i) = St_m(m_j)$  maka  $\Theta_{\leq St_m(m_j)} = \emptyset$ . Menurut Lemma 4.5, karena  $St_m(m_i) = St_m(m_j)$ ,  $m_j \notin \ell$ .
- 2) Jika  $St_m(m_i) < St_m(m_j)$  maka  $m \in \Theta_{=St_m(m_j)} \neq \emptyset$ .

**Teorema 4.1** Setelah kegagalan, eksekusi valid dan asumsi send-determinism dipenuhi, maka rollback recovery pada proses berhasil.

**Bukti.** Berdasarkan Lemma 4.1 dan 4.3 membuktikan bahwa setelah kegagalan, semua proses dapat mengirim urutan pesan valid. Lemma 4.6 menunjukkan bahwa protokol memberlakukan perintah kausal dalam pengiriman pesan.

## 5. Kesimpulan

Penelitian ini menyajikan sifat-sifat rollback recovery menggunakan uncoordinated checkpointing berbasis causality strength dengan prinsip protokol send-determinism. Protokol ini memberikan alternatif untuk dikoordinasikan antara checkpointing dan logging pesan. Sehingga diperoleh bahwa tidak terjadi efek domino, dan hanya log sebagian kecil dari pesan yang dieksekusi.

## DAFTAR PUSTAKA

- [1]. Chakarat S., Neeraj M., and Vijay K. Garg., 1999. A lightweight Algorithm for Causal Pesan Ordering in Mobile Computing Systems. *Departement Electrical and Computer Engineering*.
- [2]. Guermouche, Amina, et al., 2011. "Uncoordinated checkpointing without domino effect for send-deterministic mpi applications." *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*. IEEE.
- [3]. L. Lamport., 1978. Time, clock and the ordering of event in a distributed system. *Communications of the ACM*, 21 (7):558 - 565.
- [4] Leonardo Fialho\_, Dolores Rexachs and Emilio Luque. 2007. Defining the Checkpoint Interval for Uncoordinated Checkpointing Protocols. *Department of Computer Architecture and Operating System, University Autonoma of Barcelona, Spain*.
- [5]. S. Veerapandi, S. Gavaskar, StD, dan A. Sumithra, StD., 2017. A Hybrid Fault Tolerance System for Distributed Environment using Check Point Mechanism and Replication. *Research Scholar School of information Technology Madurai Kamaraj University*.
- [6]. Shwethashree A, dan Swathi D V., 2017. A Brief Review Of Approaches For Fault Tolerance In Distributed Systems. *Institute of Technology and Management, Ballari, Karnataka, India*.
- [7]. Yoshida, Takaichi., 2001. Pesan ordering based on the strength of a causal relation. In: *Information Networking, Proceedings. 15th International Conference on*. IEEE. p. 915-920.