Uso de *Broadcast* na Sincronização de *Checkpoints* em Protocolos Minimais

Tiemi C. Sakata[*] **Islene C. Garcia** **Luiz E. Buzato**

Universidade Estadual de Campinas
Caixa Postal 6176
13083-970 Campinas, SP, Brasil
Tel: +55 19 3788 5876 Fax: +55 19 3788 5847

{tiemi, islene, buzato}@ic.unicamp.br

Abstract. *In a synchronous checkpoiting, the application can be easily recovered in case of failures because the processes may rollback to their last checkpoint on stable storage. This article examine the minimal protocols, in which just a minimal number of processes take checkpoints to construct a consistent global checkpoint. Cao and Singhal propose a new approach to develop a minimal protocol. This approach uses a broadcast to block all processes and centralizes to a unique process the task of determine which processes should take checkpoints during the consistent global checkpoint construction. This article contains a prove the protocol proposed by Cao and Singhal is not minimal and we propose a correction to change the protocol and to guarantee the minimality.*

Resumo. *Nos protocolos de* checkpointing *síncronos, a aplicação pode ser facilmente restabelecida após a ocorrência de uma falha, pois basta que todos os processos retornem ao seu último* checkpoint *salvo. Neste artigo, exploramos a classe de protocolos síncronos minimais na qual um número minimal de processos salva* checkpoints *a cada invocação do protocolo para a construção de um* checkpoint *global consistente. Cao e Singhal propuseram uma nova abordagem para desenvolver um protocolo minimal que utiliza* broadcast *para bloquear todos os processos e centralizar a um único processo a tarefa de determinar quais processos devem salvar* checkpoints *durante a a construção do* checkpoint *global consistente. Neste texto, mostramos a não minimalidade do protocolo de Cao e Sighal e propomos uma correção para tornar o protocolo minimal.*

1. Introdução

Um sistema distribuído é composto por uma coleção de computadores autônomos interligados por uma rede de comunicação e é visto pelos usuários como um único sistema coerente [17]. As aplicações distribuídas possuem melhor desempenho, são escaláveis e permitem compartilhamento transparente de recursos existentes no sistema, porém estão mais sucetíveis a falhas. Para evitar perda de computação na recuperação de um sistema após a ocorrência de uma falha, os protocolos de *checkpointing* armazenam os estados dos

[*]Tiemi C. Sakata recebe apoio financeiro do CNPq, processo número 141808/01-2.

processos, também chamados de *checkpoints*, periodicamente em memória estável. Para garantir a recuperação de um sistema distribuído com qualquer padrão de comunicação é necessário que o sistema retroceda para o último estado global consistente, ou seja, um estado global que poderia ter sido obtido por um observador onisciente externo [4].

Nos protocolos de *checkpointing* síncronos não ocorre o chamado efeito dominó (retrocesso da aplicação ao estado inicial em caso de falha) [14]. Utiliza-se mensagens de controle para sincronizar os processos sinalizando o momento de início e término da execução do protocolo. O início é determinado pela invocação por algum dos processos de sua implementação local do protocolo. O término é determinado via a verificação de uma condição global que indica que todos os *checkpoints* necessários foram armazenados. A execução do protocolo, desde o início por um dos processos, o iniciador, até seu término é denominada *construção global*. Ao final de uma construção global, o *checkpoint* global consistente representado pelos últimos *checkpoints* locais é o *checkpoint* global consistente mais recente.

Os protocolos sícronos minimais tentam diminuir o custo total de armazenamento induzindo um número minimal de processos a gravarem seus *checkpoints* [8, 10, 15]. Cao e Singhal provam que todo protocolo minimal é bloqueante [1], ou seja, os processos que salvam *checkpoints* ficam bloqueados para a aplicação durante a construção global.

Houve na literatura um esforço em tentar reduzir o número de mensagens de controle necessários durante uma construção global. Prakash e Singhal propõem um mecanismo de rastreamento de dependências e utilizam um vetor de bits propagado durante a construção global para reduzir o número de mensagens de controle necessário [12]. Notamos porém que este mecanismo não é suficiente para garantir a minimalidade e propusemos um novo mecanismo baseado em precedência entre os *checkpoints* dos processos para desenvolver um protocolo minimal [15].

Uma abordagem alternativa para a construção de protocolos minimais foi proposta por Cao e Singhal [3]. Esta abordagem utiliza o mesmo mecanismo proposto por Prakash e Singhal [12] para rastrear dependências. Ao iniciar uma construção global, o processo iniciador salva um *checkpoint* e faz um *broadcast* pedindo a todos os processos suas informações de dependências. Todos os processos que recebem essa mensagem ficam bloqueados para a aplicação e enviam uma resposta ao iniciador. O iniciador coleta as informações de dependências de todos os processos e seleciona quais processos devem salvar *checkpoints*. O iniciador envia então uma mensagem de requisição aos processos selecionados e uma mensagem de liberação para os não selecionados, desbloqueando-os.

Neste artigo mostramos que o mecanismo de rastreamento de dependências empregado por Prakash e Singhal [12] e Cao e Singhal [3] não garante que a minimalidade do protocolo. Um contra-exemplo demonstra que o protocolo de Cao e Singhal [3] não é minimal e nos permite propor um protocolo minimal que utiliza um mecanismo para capturar as precedências entre *checkpoints* ao invés de dependências entre processos.

O restante do texto está organizado da seguinte maneira. A Seção 2 descreve o modelo computacional que consideramos neste artigo. A Seção 3 compara as duas abordagens existentes para protocolos de *checkpointing* minimais. A Seção 4 descreve uma de nossas contribuições em protocolos minimais. A Seção 5 descreve o protocolo proposto. A Seção 6 encerra o documento descrevendo as conclusões do trabalho.

2. Modelo Computacional

Um sistema distribuído é composto por n processos seqüênciais e autônomos que executam eventos internos, de envio e de recepção—troca—de mensagens. A troca de mensagens é o único mecanismo de comunicação utilizado pelos processos. Há garantia de entrega de mensagens, mas estas podem sofrer atrasos arbitrários e chegar aos seus destinos fora de ordem. A comunicação é feita através de canais unidirecionais que interligam pares de processos. Consideramos que a rede de comunicação é fortemente conexa (nenhum processo é isolado), mas não necessariamente completa (a comunicação entre um par de processos pode se dar via processos intermediários). Não existem mecanismos para compartilhamento de memória, acesso a um relógio global, sincronização de relógios locais ou conhecimento a respeito das diferenças de velocidade entre os processadores. A memória estável é suficiente para gravar todos os *checkpoints* necessários à computação, garantindo a correta recuperação de seu estado em caso de falha.

2.1. Precedência e Consistência

A noção de estado consistente de um sistema distribuído é derivada do conceito de precedência causal entre eventos que foi proposto por Lamport [9]. A definição de precedência causal utiliza e_i^ι para denotar o ι-ésimo evento executado pelo processo p_i.

Definição 1 Precedência causal — *Um evento e_a^α precede um evento e_b^β ($e_a^\alpha \rightarrow e_b^\beta$) se*

(i) $a = b$ *e* $\beta = \alpha + 1$, *ou*

(ii) existe uma mensagem m que foi enviada em e_a^α e recebida em e_b^β, ou

(iii) existe um evento e_c^γ tal que $e_a^\alpha \rightarrow e_c^\gamma \wedge e_c^\gamma \rightarrow e_b^\beta$.

Um *checkpoint* é um evento interno que armazena o estado do processo em memória estável e um intervalo entre *checkpoint*s é o conjunto de eventos ocorridos entre dois *checkpoints* consecutivos do mesmo processo. Utilizamos c_i^ι para denotar o ι-ésimo *checkpoint* gravado pelo processo p_i, sendo que $\iota = 1$ representa o *checkpoint* inicial.

Dizemos que c_a^α precede c_b^β ($c_a^\alpha \rightarrow c_b^\beta$) se o evento que originou c_a^α precede o evento que originou c_b^β. Um *checkpoint* c_a^α precede diretamente um *checkpoint* c_b^β se o processo p_a enviou uma mensagem m para p_b após o armazenamento de c_a^α e p_b recebeu m antes do armazenamento de c_b^β. Uma precedência transitiva de c_a^α para c_b^β é formada por uma seqüência de mensagens iniciada por p_a após c_a^α e terminada em p_b antes de c_b^β.

A definição de checkpoint global consistente é baseada no conceito de precedência entre checkpoints, que é capturada formalmente através da relação de z-precedência entre *checkpoints* [6]. Esta relação equivale ao conceito de *zigzagpath* introduzido originalmente por Netzer e Xu [11].

Definição 2 Z-Precedência—*Um checkpoint c_a^α z-precede um checkpoint c_b^β ($c_a^\alpha \rightsquigarrow c_b^\beta$) se*

(i) $c_a^\alpha \rightarrow c_b^\beta$, *ou*

(ii) $\exists c_c^\gamma : (c_a^\alpha \rightsquigarrow c_c^\gamma) \wedge (c_c^{\gamma-1} \rightsquigarrow c_b^\beta)$

Um *checkpoint* global é formado por um conjunto de *checkpoints*, um por processo e é consistente se não existe relação de z-precedência entre os *checkpoints* do conjunto.

Definição 3 ***Checkpoint* global consistente** — *Um checkpoint global* $\mathcal{C} = \{c_0^{\iota 0}, \ldots, c_{n-1}^{\iota n-1}\}$ *é consistente se, e somente se,* $\forall i, j : 0 \leq i, j < n : c_i^{\iota i} \not\rightsquigarrow c_j^{\iota j}$.

A Figura 1 ilustra um diagrama espaço-tempo para três processos com quadrados em preto representando *checkpoints*. A linha $\mathcal{C}$ forma um *checkpoint* global inconsistente pois o primeiro *checkpoint* de p_0 z-precede o segundo *checkpoint* de p_1. A linha $\mathcal{C}'$ mostra um *checkpoint* global consistente.

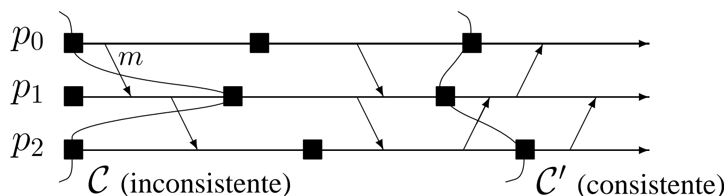

Figura 1: Consistência em *checkpoints* globais

3. Desenvolvimento de Protocolos Minimais

Um protocolo minimal é aquele que induz um conjunto minimal de processos a salvarem *checkpoints* durante uma construção global. Para que isso ocorra ele deve então induzir a retirada de *checkpoints* somente nos processos que durante o seu último intervalo de *checkpoint* forem z-precedentes do *checkpoint* armazenado pelo processo que iniciou a construção. A busca da minimalidade gerou duas abordagens para a organização dos processos durante a construção global: (i) árvore e (ii) *broadcast*.

3.1. Protocolos Minimais com Abordagem Árvore

Os protocolos propostos na literatura que utilizam a abordagem em que os processos se organizam em árvore são executados em três fases [8, 10, 15]:

1. fase de requisições – um processo iniciador invoca o protocolo síncrono armazenando um *checkpoint*, fica bloqueado e envia mensagens de requisição para os processos que dependem dele no seu último intervalo de *checkpoint*.
2. fase de respostas – cada processo que recebe uma mensagem de requisição pode salvar um *checkpoint*, ficar bloqueado e enviar uma mensagem de resposta para o processo emissor da requisição ou diretamente ao iniciador.
3. fase de liberações – após receber mensagens de resposta dos processos envolvidos, o iniciador se desbloqueia e envia uma mensagem de liberação aos processos participantes para que esses também voltem a sua computação normal.

A Figura 2 ilustra as fases de um protocolo minimal. O processo p_3 é o iniciador e envia uma mensagem de requisição para p_2. O processo p_2, após salvar um *checkpoint*, envia uma mensagem de requisição para p_1 e p_0. Note que nem todos os processos necessitam armazenar *checkpoints* em uma construção global. Todo processo que recebe uma mensagem de requisição, envia uma mensagem de resposta ao iniciador que encerra a construção global através do envio das mensagens de liberação.

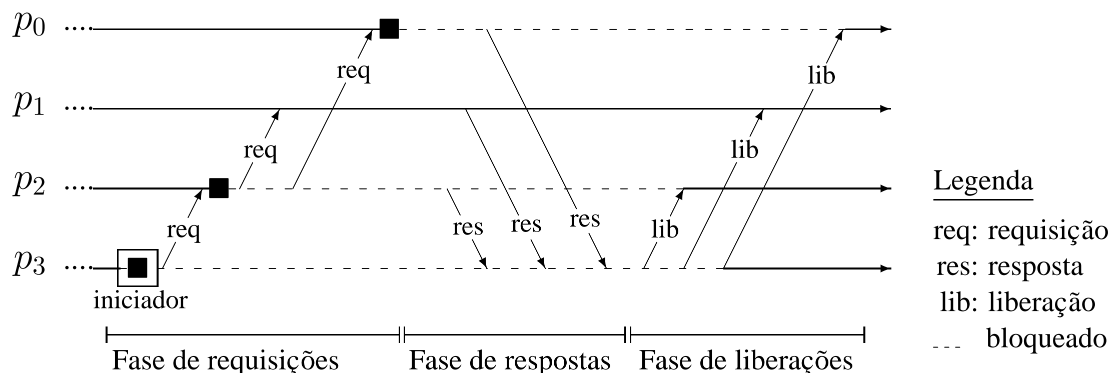

Figura 2: Protocolo Minimal com Abordagem Árvore

3.2. Protocolos Minimais com Abordagem *Broadcast*

Cao e Singhal propuseram uma nova abordagem, a qual chamamos de abordagem *broadcast*, que permite a um único processo (o iniciador) decidir quais os processos que devem armazenar *checkpoints* durante uma construção global [3]. Note que todos os processos devem ficar bloqueados para a aplicação enquanto o iniciador toma essa decisão.

O mecanismo utilizado por Cao e Singhal [3] para rastrear as dependências formadas pelas mensagens da aplicação é semelhante ao mecanismo utilizado por Prakash e Singhal [12]. Neste mecanismo, todos os processos anotam em um vetor de bits as mensagens recebidas no seu último intervalo de *checkpoints*. Este protocolo necessita de duas fases adicionais iniciais:

1. fase de *broadcast* – o iniciador faz um *broadcast* da mensagem de bloqueio. Todo processo que recebe esta mensagem envia seu vetor com as informações de dependência entre processos ao iniciador e fica bloqueado.
2. fase de respostas ao bloqueio – o iniciador constrói uma matriz $n \times n$ a partir dos vetores recebido pelos processos. Através desta matriz, o iniciador pode verificar quais os processos que dependem transitivamente do iniciador, ou seja, os processos que devem armazenar *checkpoints* (participantes) durante a construção global. O iniciador envia uma mensagem de requisição a cada participante e uma mensagem de liberação aos outros processos.

Os processos participantes da construção global passam pelas mesmas três fases da abordagem árvore, ou seja, os processos que recebem uma mensagem de requisição, armazenam um *checkpoint* e enviam uma mensagem de resposta ao iniciador. Quando o iniciador recebe uma mensagem de resposta de cada um dos participantes, fica desbloqueado e envia uma mensagem de liberação a todos os participantes da construção global para que voltem à sua computação normal. Um exemplo de execução da abordagem *broadcast* é ilustrado pela Figura 3.

Legenda

bloq: bloqueio
rb: resposta ao bloqueio
req: requisição
res: resposta
lib: liberação
--- bloqueado

Figura 3: Protocolo Minimal com Abordagem *Broadcast*

4. Uma Contribuição para o Desenvolvimento de Protocolos Minimais

A Seção anterior descreveu os protocolos síncronos minimais em função das fases e estratégias de troca de mensagens utilizadas para a obtenção de *checkpoints* globais consistentes. Nesta Seção, descrevemos esses protocolos sob o ponto de vista da busca pela minimalidade e observamos que dois deles não atingem esse objetivo. Essa é uma das contribuições deste trabalho, a outra é a descrição de um algoritmo que corrige essa deficiência.

Um protocolo minimal pode deixar de ser correto durante a construção global se:

- ele inclui no conjunto de participantes um processo cujo último *checkpoint* não z-precede o *checkpoint* armazenado pelo iniciador e/ou
- ele deixa de incluir entre os participantes um processo cujo último *checkpoint* z-precede o *checkpoint* armazenado pelo iniciador.

Um protocolo que exibe o primeiro defeito deixa de ser minimal porém constrói um *checkpoint* global consistente, porque incluiu no conjunto de participantes um processo desnecessário. Um protocolo que exibe o segundo defeito, obterá ao final da construção global um *checkpoint* global *inconsistente*.

Os protocolos propostos por Prakash e Singhal [12] e por Cao e Singhal [3] baseiam-se em um mecanismo de rastreamento de dependências que faz com que eles exibam o primeiro defeito, deixando portanto, de serem minimais. O restante desta Seção demonstra este fato através da análise de um contra-exemplo aplicado ao protocolo de Cao e Singhal [3] (Figura 4); a não minimalidade do protocolo proposto por Prakash e Singhal é descrita em [15].

O mecanismo de execução do protocolo de Cao e Singhal está descrito na Seção 3.2 e um exemplo do funcionamento do protocolo é ilustrado pela Figura 4. Na figura, p_1 envia uma mensagem para p_0 e inicia uma construção global trivial porque p_1 não recebeu mensagens durante o seu último intervalo de *checkpoint*. Assim, essa construção encerra-se sem que p_0 tenha que salvar um *checkpoint* e o *checkpoint* global consistente minimal obtido contém os *checkpoints* $\mathcal{C} = \{a, b\}$. Em um outro momento,

Figura 4: Contra exemplo para o protocolo proposto por Cao e Singhal [3]

p_0 assume o papel de iniciador. Observe que p_0 recebeu uma mensagem de p_1 durante o seu último intervalo de *checkpoint* e é por isso que p_0 pede a p_1 que salve o *checkpoint* d. Neste ponto, o protocolo de Cao e Singhal exibe o primeiro defeito porque induziu p_1 a salvar um *checkpoint* desnecessário já que o $\mathcal{C}' = \{c, b\}$ é consistente.

5. Protocolo Broad-min

Nesta seção, propomos um protocolo síncrono minimal, chamado Broad-min, baseado no protocolo proposto por Cao e Singhal [3].

O protocolo Broad-min utiliza vetores de relógios para rastrear as precedências existentes entre os últimos *checkpoints* de cada processo [5, 16]. Cada processo possui um vetor de relógios de n entradas iniciadas com valor 0. Todo processo p_i incrementa o valor da posição i de seu vetor imediatamente antes da gravação de um *checkpoint*. Na recepção de cada mensagem msg da aplicação o processo p_j atualiza o seu vetor de relógios da seguinte maneira:

se $\texttt{VR}_j\texttt{[i]} < \texttt{msg.VR[i]}$, então $\texttt{VR}_j\texttt{[i]} \leftarrow \texttt{msg.VR[i]}$, para todo $i \neq j$.

5.1. Descrição do Protocolo

Nesta seção, mostramos o mecanismo de execução do protocolo Broad-min. Ao final da seção, descrevemos o algoritmo em pseudo-código deste protocolo e ilustramos sua execução através de um exemplo.

Início da construção global

Um processo p_i, ao iniciar uma construção global, armazena um *checkpoint*. Se p_i não recebeu nenhuma mensagem no último intervalo de *checkpoint*, a construção global é encerrada. Caso contrário, p_i fica bloqueado para a aplicação e faz um *broadcast* da mensagem de bloqueio.

Recepção da mensagem de bloqueio

Todo processo p_j, ao receber uma mensagem de bloqueio, fica bloqueado para a aplicação e envia o seu vetor de relógios `VR` e um vetor de bits, chamado vetor de participantes `VP`, ao iniciador através da mensagem de resposta ao bloqueio. O vetor de participantes indica quais os processos que dependem direta ou transitivamente de p_j no último intervalo e é calculado comparando o vetor de relógios `VR` atual com o vetor de relógios salvo juntamente com último *checkpoint* `ultimo_VR`. Ou seja, existe uma precedência causal entre um *checkpoint* de p_i com o último *checkpoint* de p_j se p_j recebeu direta ou indiretamente uma informação nova de p_i:

$$\texttt{VP}_j\texttt{[i] = 1} \;\Rightarrow\; \texttt{VR}_j\texttt{[i]} - \texttt{ultimo_VR}_j\texttt{[i]} > 0$$

Recepção da mensagem de resposta ao bloqueio

O iniciador p_i ao receber uma mensagem de resposta ao bloqueio de p_j, insere VR de p_j na linha j da matriz MVR e da mesma forma, insere VP em sua matriz MVP. Ao receber a mensagem de resposta ao bloqueio de todos os processos, p_i possui duas matrizes nxn: MVR com os vetores de relógio e MVP com as informações de dependências de cada processo. O iniciador calcula então todos os prováveis participantes de sua construção global através da multiplicação do seu vetor de participantes e MVP. Para garantir a minimalidade do protocolo, um processo p_p faz parte do conjunto de processos participantes da construção global somente se pelo menos outro participante do conjunto conhece o último *checkpoint* de p_p. O iniciador p_i envia uma mensagem de requisição para cada processo participante e uma mensagem de liberação para os outros processos.

Recepção da mensagem de requisição

Todo processo que recebe uma mensagem de requisição, salva um *checkpoint*, e envia uma mensagem de resposta ao iniciador.

Recepção da mensagem de resposta

O iniciador, ao receber uma mensagem de resposta de cada um dos participantes, conclui que a construção global pode ser encerrada e envia uma mensagem de liberação para todos os participantes e volta a processar a aplicação.

Recepção da mensagem de liberação

Todo processo que recebe uma mensagem de liberação, fica desbloqueado voltando à sua computação normal.

O protocolo Broad-min é descrito pelo Algoritmo 1. Consideramos que todo procedimento deste algoritmo é executado de forma atômica.

5.2. Exemplo

A execução deste protocolo pode ser exemplificada através do cenário ilustrado pela Figura 5. Neste cenário, a construção global iniciada por p_2 é encerrada após p_2 salvar um *checkpoint* pois p_2 não recebeu mensagens no último intervalo de *checkpoint*. Já quando p_0 inicia uma construção global, após salvar um *checkpoint*, faz um *broadcast* da mensagem de bloqueio. Ao receber a mensagem resposta ao bloqueio de todos os processos, p_0 possui as matrizes de Vetores de Relógios e de Vetores de Precedências (Figura 6).

Para calcular os possíveis participantes, p_2 multiplica seu vetor de participantes com MVP até que não haja mudança no vetor de participantes. Pela Figura 5, este cálculo é feito pelo iniciador p_0 e é descrito pela Figura 7.

Ao final da multiplicação das matrizes, temos como resultado um vetor indicando quais processos da aplicação são prováveis participantes da construção global. O nosso objetivo porém é de induzir um número minimal de processos a armazenarem *checkpoints*. Através da matriz MVR podemos notar que nenhum processo conhece o último

Algoritmo 1: Broad-min

```
Variáveis do processo:
  VR ≡ vetor[0...n − 1] de inteiros
  ultimo_VR ≡ vetor[0...n − 1] de inteiros
  participantes ≡ vetor[0...n − 1] de bits
  aux ≡ vetor[0...n − 1] de bits
  MVR ≡ matriz[0...n − 1][0...n − 1] de inteiros
  MVP ≡ matriz[0...n − 1][0...n − 1] de bits
  pid ≡ inteiro
  bloqueado ≡ booleano

Tipos de mensagem:
  m – aplicação:
      VR ≡ vetor[0...n − 1] de inteiros
  bloq – bloqueio
  rb – resposta ao bloqueio:
      VR ≡ vetor[0...n − 1] de inteiros
      VP ≡ vetor[0...n − 1] de bits
  req – requisição
  res – resposta
  lib – liberação

Início:
  ∀i: VR[i] ← 0, ultimo_VR[i] ← 0
  ∀i: participantes[i] ← 0, aux[i] ← 0
  ∀i: ∀j: MVR[i][j] ← 0, MVP[i][j] ← 0
  bloqueado ← falso
  VR[pid] ← VR[pid] + 1
  armazena o checkpoint

Envio da mensagem da aplicação (m) para p_k:
  se (não bloqueado)
      m.VR ← VR
      transmite a mensagem da aplicação (m)

Recepção da mensagem da aplicação (m) de p_k:
  se (não bloqueado)
      ∀i: se (m.VR[i] > VR[i])
              VR[i] ← m.VR[i]
      processa a mensagem da aplicação (m)

Início da construção global:
  VR[pid] ← VR[pid] + 1
  armazena o checkpoint
  MVR[pid] ← VR
  ∀i: se ((VR[i] − ultimo_VR[i]) > 0)
          MVP[pid][i] ← 1
  se (∃i ≠ pid: MVP[pid][i] = 1)
      bloqueado ← verdadeiro
      faz um broadcast do bloqueio (bloq)

Recepção do bloqueio (bloq) de p_k:
  bloqueado ← verdadeiro
  rb.VR ← VR
  ∀i: se ((VR[i] − ultimo_VR[i]) > 0)
          rb.VP[i] ← 1
  transmite a resposta (rb) do bloqueio para p_k

Recepção da resposta ao bloqueio (rb) de p_k:
  MVR[k] ← rb.VR
  MVP[k] ← rb.VP
  participantes[k] ← 1
  se (∀i: participantes[i] = 1)
      participantes ← MVP[pid]
      aux ← participantes
      participantes ← participantes ∗ MVP
      enquanto (participantes ≠ aux)
          aux ← participantes
          participantes ← participantes ∗ MVP
      ∀i ≠ pid: se (participantes[i] = 1)
                  se (∀j ≠ i: participantes[j] = 1 e
                      MVR[j][i] < MVR[i][i])
                          participantes[i] ← 0
      enquanto (participantes ≠ aux)
          aux ← participantes
          ∀i ≠ pid: se (participantes[i] = 1)
                      se (∀j ≠ i: participantes[j] = 1 e
                          MVR[j][i] < MVR[i][i])
                              participantes[i] ← 0
      se (∀i ≠ pid: participantes[i] = 0)
          bloqueado ← falso
      senão
          ∀i ≠ pid: se (participantes[i] = 1)
                  transmite a requisição (req) para p_i
              senão
                  transmite a liberação (lib) para p_i

Recepção da requisição (req) de p_k:
  VR[pid] ← VR[pid] + 1
  armazena o checkpoint
  ultimo_VR ← VR
  transmite (res) para p_k

Recepção da resposta (res) de p_k:
  respostas[k] ← 1
  se (participantes = respostas)
      ∀i ≠ pid: se (participantes[i] = 1)
              transmite a liberação (lib) para p_i
      ∀i: participantes[i] ← 0, respostas[i] ← 0
      ∀i: aux[i] ← 0
      ∀i: ∀j: MVR[i][j] ← 0, MVP[i][j] ← 0
      ultimo_VR ← VR
      bloqueado ← falso

Recepção da liberação (lib) de p_k:
  bloqueado ← falso
```

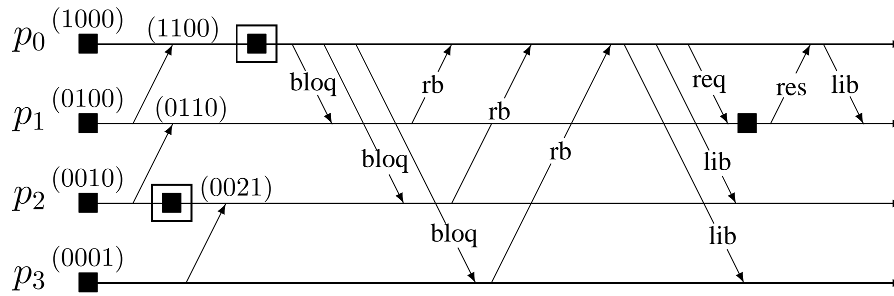

Figura 5: Cenário que ilustra a execução do Broad-min

$$MVR = \begin{pmatrix} 2 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 2 & 1 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

(a) Matriz de Vetores de Relógios

$$MVP = \begin{pmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

(b) Matriz de Vetores de Participantes

Figura 6: Matrizes construídas pelo iniciador

checkpoint de p_2 (basta verificar a coluna 2 da matriz `MVR`). Portanto p_2 deixa de ser participante da construção de do iniciador p_0. Como o único que conhecia o último *checkpoint* de p_3 era p_2 e p_2 não é participante, então p_3 também deixa de ser participante da construção global. Portanto, p_0 envia uma mensagem de requisição para o seu único participante e envia uma mensagem de liberação para p_2 e p_3. O processo p_1, ao receber a mensagem de requisição, armazena um *checkpoint* e envia uma mensagem de resposta para o iniciador. Após receber a mensagem de resposta de p_1, o iniciador envia uma mensagem de liberação para p_1 e a construção global é encerrada.

$$\begin{pmatrix} 1 & 1 & 0 & 0 \end{pmatrix} * \begin{pmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & 0 \end{pmatrix} * \begin{pmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{pmatrix} =$$

$$= \begin{pmatrix} 1 & 1 & 1 & 1 \end{pmatrix} * \begin{pmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & 1 \end{pmatrix}$$

Figura 7: Cálculo dos prováveis participantes

6. Conclusão

Os protocolos de *checkpointing* síncronos bloqueantes simplificam a recuperação de uma falha e evitam o efeito dominó mantendo *checkpoints* globais consistentes em memória estável. Para reduzir o custo de cada construção global, os protocolos minimais induzem apenas um número minimal de processos a armazenarem *checkpoints*. Sabemos que todo protocolo minimal é bloqueante, ou seja, os processos participantes construção global suspendem suas atividades para a aplicação durante a execução de *checkpointing*. Observamos que durante o desenvolvimento desses protocolos duas estratégias foram utilizadas: árvore e *broadcast*. A abordagem em que os processos se organizam em árvore bloqueia apenas os processos participantes da construção global e eles determinam se salvam ou não um checkpoint. Em contraste, a abordagem baseada em *broadcast* bloqueia todos os processos e transfere exclusivamente para o iniciador a tarefa de determinar quais processos armazenarão *checkpoints* durante a construção global.

Neste artigo, provamos que o protocolo proposto por Cao e Singhal [3], apesar de construir *checkpoints* globais consistentes, não é minimal. Notamos que o mecanismo de rastreamento de dependências entre processos utilizado por Cao e Singhal [3] não é suficiente para garantir a minimalidade do protocolo. A demonstração desse fato é simples e apóia-se em um contra-exemplo que nos motivou a procurar um novo mecanismo de rastreamento, baseado no uso de vetores de relógios lógicos. Finalmente, utilizamos o novo mecanismo para desenvolver o protocolo Broad-min, que é baseado em *broadcast* e minimal.

Cao e Singhal fizeram uma análise teórica e concluíram que a abordagem *broadcast* minimiza o tempo de bloqueio durante uma construção global. No entanto, acreditamos que a melhor abordagem depende da aplicação e também da configuração do sistema distribuído. Como trabalho futuro, seria interessante desenvolver várias aplicações e executá-las em diferentes configurações de sistemas distribuídos a fim de comparar as duas abordagens.

Outro trabalho futuro seria analisar os protocolos de *checkpointing* síncronos não bloqueantes [1, 2, 3, 7, 12, 13]. Nos protocolos não bloqueantes, os processos não suspendem a execução da aplicação durante uma construção global. Os protocolos síncronos não bloqueantes propostos na literatura são muito complexos e suas provas são baseados nos diferentes cenários gerados por eles. O problema deste tipo de prova é que a falta de um cenário pode gerar conclusões erradas. Cao e Singhal [1] detectam dois problemas existentes no protocolos proposto por Prakash e Singhal [13] e ao analisarmos as provas de correção deste último protocolo, notamos que estão incompletas.

Referências

[1] G. Cao and M. Singhal. On Coordinated Checkpointing in Distributed Systems. *IEEE Trans. on Parallel and Distributed Systems*, 9(12):1213–1225, Dec. 1998.

[2] G. Cao and M. Singhal. Mutable Checkpoints: A New Checkpointing Approach for Mobile Computing Systems. *IEEE Transaction on Parallel and Distributed Systems*, 12(2):157–172, 2001.

[3] G. Cao and M. Singhal. Checkpointing with Mutable Checkpoints. *Theoretical Computer Science*, 290(2):1127–1148, jan 2003.

[4] M. Chandy and L. Lamport. Distributed Snapshots: Determining Global States of Distributed Systems. *ACM Transaction on Computing Systems*, 3(1):63–75, Feb. 1985.

[5] Ö. Babaoğlu and K. Marzullo. Consistent Global States of Distributed Systems: Fundamental Concepts and Mechanisms. In S. Mullender, editor, *Distributed Systems*, pages 55–96. Addison-Wesley, 1993.

[6] I. C. Garcia and L. E. Buzato. Progressive Construction of Consistent Global Checkpoints. In *19th IEEE International Conference on Distributed Computing Systems*, Austin, Texas, EUA, June 1999.

[7] E. Gendelman, L. Bic, and M. B. Dillencourt. An Efficient Checkpointing Algorithm for Distributed Systems Implementing Reliable Communication Channels. In *Symposium on Reliable Distributed Systems*, pages 290–291, 1999.

[8] R. Koo and S. Toueg. Checkpointing and Rollback-Recovery for Distributed Systems. *IEEE Transaction on Software Engineering*, 13:23–31, Jan. 1987.

[9] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM*, 21(7):558–565, July 1978.

[10] P. J. Leu and B. Bhargava. Concurrent Robust Checkpointing and Recovery in Distributed Systems. In *4th IEEE Int. Conference on Data Engineering*, pages 154–163, 1988.

[11] R. H. B. Netzer and J. Xu. Necessary and Sufficient Conditions for Consistent Global Snapshots. *IEEE Transaction on Parallel and Distributed Systems*, 6(2):165–169, 1995.

[12] R. Prakash and M. Singhal. Minimal Global Snapshot and Failure Recovery using Infection. Technical Report OSU-CISRC-12/93-TR42, Department of Computer Science, The Ohio State University, 1993.

[13] R. Prakash and M. Singhal. Low-Cost Checkpointing and Failure Recovery in Mobile Computing Systems. *IEEE Transaction on Parallel and Distributed Systems*, 7(10):1035–1048, Oct. 1996.

[14] B. Randell. System Structure for Software Fault Tolerance. *IEEE Transaction on Software Engineering*, 1(2):220–232, June 1975.

[15] T. C. Sakata, I. C. Garcia, and L. E. Buzato. Checkpointing Síncrono Bloqueante Minimal com Iniciadores Concorrentes. In *Simpósio Brasileiro de Redes de Computadores*, pages 681–696, Natal, Rio Grande do Norte, May 2003.

[16] R. Schwarz and F. Mattern. Detecting Causal Relationships in Distributed Computations: In Search of the Holy Grail. *Distributed Computing*, 7(3):149–174, Mar. 1994.

[17] A. S. Tanenbaum and M. Steen. *Distributed Systems Principles and Paradigms*. Alan Apt, 2002.