Checkpointing Síncrono Bloqueante Minimal com Iniciadores Concorrentes*

Tiemi C. Sakata Islene C. Garcia Luiz E. Buzato

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

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

Abstract. *Synchronous checkpointing protocols coordinate the collection of checkpoints to easy the recovery of the system in case of failure. A blocking protocol suspends the underlying computation during checkpoint collection and a minimal protocol forces only a minimal number of process to take checkpoints. In this paper, we present a new minimal blocking checkpointing protocol that requires fewer control messages than the protocols proposed in the literature. We first analyze a simpler version of this protocol that assumes the existence of only one initiator at a time. After that we present an extension that handles concurrent initiators.*

Keywords: fault tolerance, checkpointing, distributed algorithms

Resumo. *Protocolos síncronos para checkpointing coordenam o armazenamento de checkpoints de modo a facilitar a recuperação após a ocorrência de uma falha. Um protocolo é bloqueante se suspende as atividades da computação durante o armazenamento dos checkpoints e é minimal se induz o número mínimo de processos a armazenarem seus checkpoints.*
Propomos neste artigo um novo protocolo bloqueante minimal para checkpointing. Este protocolo requer um número menor de mensagens de controle em relação aos protocolos síncronos bloqueantes minimais existentes na literatura. Analisamos inicialmente uma versão mais simples do protocolo que considera a existência de apenas um iniciador a cada instante. Posteriormente, apresentamos uma extensão que permite a existência de iniciadores concorrentes.

Palavras-chave: tolerância a falhas, *checkpointing*, algoritmos distribuídos

1. Introdução

Os protocolos síncronos bloqueantes para *checkpointing* utilizam mensagens de controle para coordenar o armazenamento de *checkpoints* suspendendo temporariamente as atividades da computação. Um *checkpoint* é um estado de interesse de um processo de uma

*Tiemi C. Sakata recebe apoio financeiro do CNPq, processo número 141808/01-2. Esta pesquisa também recebeu apoio do Laboratório de Sistemas Distribuídos (FAPESP, processo número 96/1532-9), do Laboratório de Alto Desempenho (FAPESP, processo número 97/10982-0) e do projeto Sistemas Avançados de Informação (PRONEX/FINEP, processo número 76.97.1022.00).

computação distribuída escolhido para ser gravado em memória estável. Os protocolos minimais tentam diminuir o custo total de armazenamento induzindo um número mínimo de processos a gravarem seus *checkpoints*.

Durante a execução da aplicação, um protocolo síncrono pode ser invocado diversas vezes e a cada invocação, também chamada de construção global, um *checkpoint* global consistente é construído [2, 5]. Desta forma torna-se interessante diminuir o número das mensagens de controle a cada construção global. Os protocolos presentes na literatura requerem $\mathcal{O}(n^2)$ mensagens de controle onde n é o números de processos na aplicação, tendo havido um esforço no sentido de diminuir a constante multiplicativa do n^2.

Na maioria dos protocolos síncronos bloqueantes minimais, uma construção global é executada em três fases: fase de requisições, fase de respostas e fase de liberações. O primeiro protocolo proposto na literatura utiliza no máximo, $n^2 - n$ mensagens em cada fase totalizando $3n^2 - 3n$ mensagens de controle [5]. Um protocolo posterior proposto por Leu e Bhargava passou a utilizar $n^2 - n$ mensagens na fase de requisições e $n - 1$ mensagens tanto na fase de respostas quanto na fase de liberações, porém necessita de uma fase adicional com $n^2 - n$ mensagens de confirmação, num total de $2n^2 - 2$ [7]. Prakash e Singhal propuseram uma abordagem que requer $0.25n^2$ mensagens na fase de requisições, $0.25n^2$ mensagens na fase de respostas e $n - 1$ mensagens na fase de liberações, totalizando $0.5n^2 + n - 1$ mensagens [8] mas este protocolo não é minimal [11].

Neste artigo propomos um novo protocolo bloqueante minimal que requer no máximo, $0.25n^2$ mensagens de requisição, $n - 1$ mensagens de resposta e $n - 1$ mensagens de liberação, ou seja, $0.25n^2 + 2n - 2$ mensagens de controle. Este protocolo utiliza vetores de relógios para o armazenamento dos *checkpoints* e propõe uma nova abordagem para detectar a terminação de uma construção global.

A presença de iniciadores concorrentes pode tornar o controle das construções globais mais complexo, pois cada iniciador deve construir um *checkpoint* global consistente. Koo e Toueg [5] propõem uma maneira extrema de lidar com o problema na qual se um processo detecta duas construções concorrentes, uma delas é abortada. Esta abordagem não é adequada, pois os processos podem, de maneira distribuída, abortar todas as construções concorrentes. Existem trabalhos na literatura unicamente voltados ao problema do tratamento de múltiplos iniciadores, mas estes trabalhos não lidam com protocolos minimais [9, 13]. Encerramos este artigo com uma extensão do protocolo proposto que trata o problema de múltiplos iniciadores mantendo a propriedade minimal e o número baixo de mensagens de controle.

Este documento está estruturado da seguinte forma. A Seção 2 descreve o modelo computacional. A Seção 3 cita trabalhos relacionados e a motivação do nosso trabalho. A Seção 4 descreve o protocolo proposto considerando um único iniciador e a Seção 5 descreve uma extensão que permite iniciadores concorrentes. Finalmente, a Seção 6 encerra o documento descrevendo as conclusões do trabalho.

2. Checkpointing

Esta Seção descreve o modelo computacional considerado e as definições necessárias para uma melhor compreensão do trabalho proposto.

2.1. Modelo Computacional

Um sistema distribuído é composto por um conjunto de n processos alocados em máquinas distintas, que executam eventos de maneira estritamente seqüencial e independente. Há garantia de entrega de mensagens, mas estas podem sofrer atrasos arbitrários e inclusive chegar aos seus destinos fora de ordem. 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 capacidade de armazenamento é grande o suficiente para gravar todos os *checkpoints* necessários à computação e para a sua recuperação em caso de falha.

2.2. Precedência e Consistência

A noção de consistência está fortemente acoplada ao conceito de precedência causal entre eventos proposto por Lamport [6]. A definição a seguir 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 de *checkpoint* é o conjunto de eventos ocorridos entre dois *checkpoints* consecutivos de 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.

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

Definição 2 *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, $\forall i, j : 0 \leq i, j < n : c_i^{\iota_i} \not\rightarrow c_j^{\iota_j}$.*

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 precede o segundo *checkpoint* de p_1. A linha $\mathcal{C}'$ mostra um *checkpoint* global consistente.

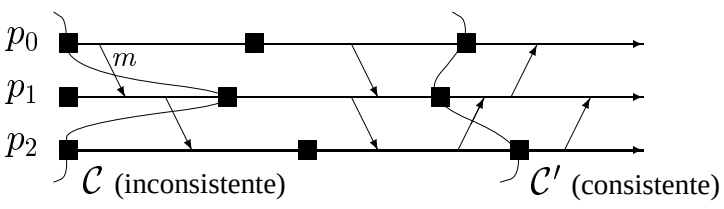


Figura 1: Consistência em *checkpoints* globais

2.3. *Checkpointing* Síncrono Bloqueante

Nos protocolos síncronos, quando um processo necessita armazenar um *checkpoint*, inicia-se um procedimento coordenado tal que a união dos últimos *checkpoints* armazenados por cada processo forma um *checkpoint* global consistente. Esse procedimento é normalmente executado em três fases, como ilustrado na Figura 2.

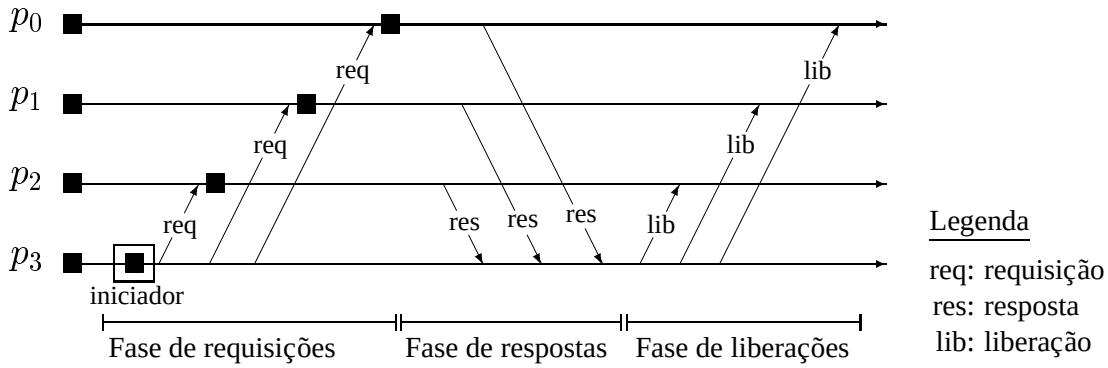


Figura 2: Protocolo de *checkpointing* síncrono bloqueante

Na primeira fase, também chamada de fase de requisições, um processo iniciador armazena um *checkpoint* local e fica bloqueado para a aplicação. Dizemos que um processo está bloqueado quando as atividades da aplicação estão suspensas, mas o processo continua apto a enviar e processar mensagens de controle. O iniciador envia uma mensagem de requisição para todos os outros processos participantes da construção global. Cada processo que recebe a mensagem de requisição grava um *checkpoint*, fica bloqueado e envia uma mensagem de resposta para o iniciador.

A segunda fase (fase de respostas) é composta pelo envio e recepção de mensagens de respostas. A terceira fase (fase de liberações) começa após o iniciador receber uma mensagem de resposta de cada um dos processos. O iniciador é desbloqueado e envia uma mensagem de liberação para todos os processos participantes. Quando um processo recebe a mensagem de liberação, é desbloqueado e continua o seu processamento.

Um *checkpoint* global consistente pode ser obtido mesmo que alguns processos não armazenem *checkpoints* durante uma construção global. Por exemplo, na Figura 3, o *checkpoint* global formado por c_0^2, c_1^2 e c_2^1 é consistente (mensagens de controle não geram inconsistência). Um protocolo para *checkpointing* síncrono é minimal se induz um número mínimo de processos a armazenarem seus *checkpoints*. A Figura 3 representa um protocolo minimal pois se p_1 não gravasse um *checkpoint* o conjunto c_0^2, c_1^1 e c_2^1 seria inconsistente.

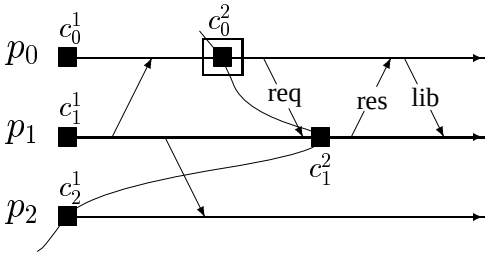


Figura 3: Protocolo síncrono minimal

3. Trabalhos Relacionados e Motivação

Apesar do limite $\mathcal{O}(n^2)$, verificamos na literatura uma redução progressiva no número de mensagens de controle utilizado por protocolos síncronos bloqueantes. Nesta Seção descrevemos alguns protocolos e idéias que motivaram o nosso trabalho utilizando a nomenclatura a seguir. Dizemos que um processo p_j é participante de uma construção global se p_j recebe uma requisição e grava um *checkpoint* durante esta construção. Um processo p_j é colaborador de p_i se p_i envia uma mensagem de requisição diretamente para p_j. O conjunto de participantes em uma construção global minimal deve ser igual para todos os protocolos enquanto o conjunto de colaboradores pode variar de protocolo para protocolo.

No protocolo proposto por Koo e Toueg, um processo p_j é colaborador de p_i se p_i recebeu uma mensagem de p_j durante o seu último intervalo de *checkpoint* [5]. Um processo iniciador envia uma mensagem para todos os seus colaboradores e cada processo que recebe uma requisição, se está desbloqueado, armazena um *checkpoint*, fica bloqueado e propaga a requisição para os seus colaboradores. Uma mensagem de resposta deve ser emitida para cada mensagem de requisição recebida. Quando o iniciador recebe uma mensagem de resposta de cada um dos seus colaboradores, inicia-se a fase de liberação e as mensagens de liberação são transmitidas de maneira semelhante à propagação da requisição. Desta forma, no pior caso (quando existe precedência direta entre todos os processos) este protocolo utiliza $n^2 - n$ mensagens de requisição, $n^2 - n$ mensagens de resposta e $n^2 - n$ mensagens de liberação.

No protocolo de Leu e Bhargava [7], são enviadas mensagens de requisição de maneira análoga ao protocolo de Koo e Toueg. Em seguida, são enviadas mensagens de confirmação para a formação de uma árvore que indica as precedências entre os processos que serão participantes nesta construção global. Esta árvore é utilizada para a transmissão de mensagens de resposta e liberação. Este protocolo requer no pior caso, $n^2 - n$ mensagens de requisição, $n^2 - n$ mensagens de confirmação, $n - 1$ mensagem de resposta e $n - 1$ mensagens de liberação, ou seja, um custo total igual a $2n^2 - 2$.

O número de mensagens de controle é reduzido no protocolo proposto por Prakash e Singhal para $0.5n^2 + n - 1$ por meio da utilização da detecção de terminação proposta por Huang [4] e da propagação de um vetor (vetor de participantes) que indica quais os processos que já estão participando da construção global [8].

Neste último protocolo, dependências diretas e transitivas entre processos são capturadas através de vetores de bits. Cada processo possui um vetor de n entradas que são iniciadas com 0 exceto a posição referente ao próprio processo que é iniciada com 1. Toda mensagem da aplicação enviada pelo processo p_i propaga seu vetor de bits. Quando um processo p_j recebe uma mensagem de p_i, p_j realiza uma operação de *ou*-lógico entre o seu vetor e o vetor recebido. Por exemplo, na Figura 4, p_0 tem o seu vetor inicialmente igual a (100) e ao receber uma mensagem de p_1, seu vetor é atualizado para (110). O processo p_i, ao iniciar uma construção global, armazena o seu *checkpoint* e propaga as mensagens de requisição para cada um de seus colaboradores. Um processo p_j é colaborador de p_i se a posição j no vetor de p_i for igual a 1. Todo processo que recebe a mensagem de requisição, se não está bloqueado, grava um *checkpoint* e propaga a requisição. Os autores deste protocolo afirmaram que ele era minimal, porém provamos a sua não minimalidade através de um contra-exemplo ilustrado pela Figura 4 [11]. Nesta figura, p_1 armazena um *checkpoint* por causa da requisição enviada por p_0 e, na recepção da requisição de p_2

armazena novamente outro *checkpoint*, sem ter enviado ou recebido nenhuma mensagem da aplicação durante esse intervalo. A união dos *checkpoints* requisitados pelo iniciador p_2 é um *checkpoint* global consistente mas não minimal.

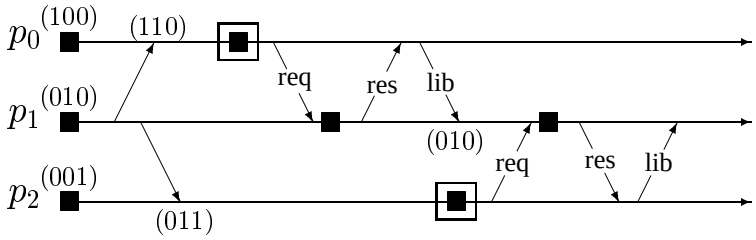


Figura 4: Protocolo proposto por Prakash e Singhal não é minimal

Podemos concluir que o protocolo proposto por Prakash e Singhal utiliza: (i) um vetor de bits para propagar as precedências transitivas entre os processos; (ii) um vetor de participantes que indica os processos participantes da construção global e (iii) detecção de terminação baseada no algoritmo proposto por Huang [4]. Observamos que o vetor de bits não é suficiente para garantir a minimalidade do protocolo e propomos um novo protocolo minimal que utiliza vetores de relógios. Este último protocolo implementa um vetor de participantes da mesma forma que (ii) porém o algoritmo de terminação utilizado é mais simples e gera um número menor de mensagens de controle.

4. VR-minimal-simples – Checkpointing Minimal com Vetores de Relógios

Nesta Seção, propomos um protocolo síncrono bloqueante chamado VR-minimal-simples, que elimina o problema existente no protocolo proposto por Prakash e Singhal. Além disso, o protocolo VR-minimal-simples requer o menor número de mensagens de controle comparado aos protocolos semelhantes existentes na literatura. Para garantir um estado global consistente, utilizamos vetores de relógios.

4.1. Vetores de Relógios

Possuem informações de precedência associadas aos *checkpoints* e são mantidos e propagados pelos processos [3, 12]. 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 $i \neq j$.

Temos as seguintes propriedades para vetores de relógios:

Propriedade 1 $c_a^\alpha \rightarrow c_b^\beta \Rightarrow VR(c_b^\beta)[a] \geq \alpha$

Propriedade 2 *Dado dois checkpoints consecutivos* $c_a^{\alpha-1}$ *e* c_a^α *e seus respectivos vetores de relógios* $VR(c_a^{\alpha-1})$ *e* $VR(c_a^\alpha)$*, é possível detectar o estabelecimento de uma nova precedência causal entre um checkpoint de* p_b *e* c_a^α *se* $VR(c_a^{\alpha-1})[b] < VR(c_a^\alpha)[b]$.

Na Figura 5, os vetores de relógios estão entre parênteses. Nesta figura, podemos notar que o primeiro *checkpoint* de p_0 precede o segundo *checkpoint* de p_1 e $VR(c_1^2)[0] = 1$ (Propriedade 1). Além disso, $VR(c_1^1)[0] = 0 < VR(c_1^2)[0] = 1$, o que indica o estabelecimento desta precedência no primeiro intervalo de p_1 (Propriedade 2).

4.2. Descrição do Protocolo

O protocolo VR-minimal-simples é iniciado por um processo p_i que armazena um *checkpoint*, incrementa o seu vetor de relógios e calcula o seu conjunto de colaboradores. Chamamos de $ultimo\_VR$ o vetor de relógios da última construção global e um processo p_j faz parte dos colaboradores de p_i se $VR_i[j] > ultimo\_VR_i[j]$ (Propriedade 2).

Quando o iniciador p_i não possui colaboradores, a construção é encerrada (como ocorre com o processo iniciador p_0 na Figura 5); caso contrário, p_i envia uma mensagem de requisição para cada um dos colaboradores e fica bloqueado. Na figura, o iniciador p_3 grava um *checkpoint* e envia mensagens de requisição para os seus colaboradores p_2 e p_1. Cada mensagem de requisição propaga o índice do iniciador p_i ($req.ind\_inic$), o índice do último *checkpoint* do colaborador p_j conhecido pelo processo emissor da mensagem de requisição ($req.indice = VR_i[j]$, quando p_i é o emissor da requisição) e o conjunto de participantes. Inicialmente o conjunto de participantes é igual ao conjunto de colaboradores do iniciador.

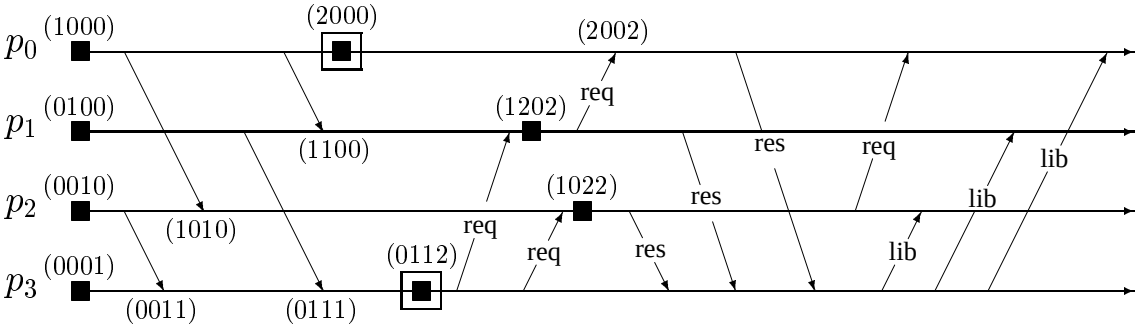


Figura 5: Protocolo VR-min-simples

Um processo desbloqueado p_j, ao receber uma mensagem de requisição, verifica se o índice do *checkpoint* da requisição ($req.indice$) é igual ao índice do seu último *checkpoint* ($VR_j[j]$). Se:

- $req.indice = VR_j[j] - p_j$ armazena um *checkpoint*, fica bloqueado, propaga a requisição para os seus colaboradores que ainda não são participantes e envia uma única mensagem de resposta ao iniciador. Na Figura 5, p_1 recebe uma mensagem de requisição de p_3 e como $req.indice = VR_1[1] = 1$, p_1 grava um *checkpoint*, propaga a mensagem requisição para o seu colaborador p_0 e envia uma mensagem de resposta para o iniciador p_3.
- $req.indice < VR_j[j] - p_j$ não precisa armazenar um *checkpoint*. Se p_j não recebeu outra mensagem de requisição referente à construção global iniciada por p_i, ou seja, se o índice de p_i propagado pela requisição for maior que o índice conhecido por p_j ($req.ind\_inic > VR_j[req.pid\_inic]$), p_j não armazena um *checkpoint* mas envia uma mensagem de resposta ao iniciador. Por exemplo, o processo p_0 na figura, ao receber a requisição, apenas atualiza o índice do iniciador e envia uma mensagem de resposta ao iniciador p_3. Quando p_j recebe uma mensagem de requisição e já conhece o índice do iniciador, isto é, uma outra requisição desta construção já foi recebida, a mensagem de requisição é simplesmente ignorada. Na mesma figura, a requisição enviada por p_2 é ignorada por p_0.

Se o processo p_k ao receber uma mensagem de requisição está bloqueado, isso implica que p_k já recebeu uma mensagem de requisição para a construção global corrente e a mensagem recebida é ignorada.

A terminação do protocolo é garantida utilizando-se dois vetores: $participantes$ (que indica quais são os processos participantes da construção) e $respostas$ (que indica quais são os processos que já enviaram respostas ao iniciador p_i). Toda mensagem de resposta propaga o vetor de participantes que o processo p_j emissor da resposta conhece. Quando p_i recebe uma resposta de p_j, p_i atualiza o seu vetor de $participantes$ unindo todos os seus participantes com os participantes de p_j e anota que já recebeu uma mensagem de resposta de p_j ($respostas_i[j] = 1$).

Quando os vetores $participantes$ e $respostas$ são iguais, p_i recebeu respostas de todos os participantes de sua construção global. O iniciador p_i envia uma mensagem de liberação para todos os processos participantes, limpa os vetores de colaboradores, participantes, respostas, atualiza o vetor $ultimo\_VR$ e fica desbloqueado. Todo processo que recebe uma mensagem de liberação, atualiza as variáveis de forma semelhante ao iniciador e fica desbloqueado.

A descrição do algoritmo **VR-min-simples** está dividido em variáveis do processo e tipos de mensagens (Algoritmo 1.1) e procedimentos atômicos (Algoritmo 1.2).

Algoritmo 1.1: VR-minimal-simples: variáveis do processo e tipos de mensagens

Variáveis do processo:
- VR $\equiv$ vetor$[0 \ldots n-1]$ de inteiros
- ultimo_VR $\equiv$ vetor$[0 \ldots n-1]$ de inteiros
- colaboradores $\equiv$ vetor$[0 \ldots n-1]$ de bits
- participantes $\equiv$ vetor$[0 \ldots n-1]$ de bits
- respostas $\equiv$ vetor$[0 \ldots n-1]$ de bits
- pid $\equiv$ inteiro
- bloqueado $\equiv$ booleano

Tipos de mensagem:
- aplicação:
  - VR $\equiv$ vetor$[0 \ldots n-1]$ de inteiros
- requisição:
  - pid_inic $\equiv$ inteiro
  - participantes $\equiv$ vetor$[0 \ldots n-1]$ de bits
  - ind_receptor $\equiv$ inteiro
  - ind_inic $\equiv$ inteiro
- resposta:
  - VR $\equiv$ vetor$[0 \ldots n-1]$ de inteiros
  - participantes $\equiv$ vetor$[0 \ldots n-1]$ de bits
- liberação:
  - VR $\equiv$ vetor$[0 \ldots n-1]$ de inteiros

4.3. Prova de Correção

Sabemos que a união dos *checkpoints* iniciais representa um *checkpoint* global consistente. Desta forma, para provar que o protocolo é correto, basta provar que após a execução de uma construção global, a união dos últimos *checkpoints* de cada processo formará um novo *checkpoint* global consistente.

Teorema 1 *Suponha que uma invocação do protocolo VR-minimal-simples foi iniciada a partir de um checkpoint global consistente. Imediatamente após o fim desta construção*

Algoritmo 1.2: VR-minimal-simples: procedimentos

Início:
∀i: VR[i] ← 0
∀i: ultimo_VR[i] ← 0
∀i: colaboradores[i] ← 0
∀i: participantes[i] ← 0
∀i: respostas[i] ← 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:
bloqueado ← *verdadeiro*
VR[pid] ← VR[pid] + 1
armazena o *checkpoint*
∀i: se (VR[i] > ultimo_VR[i])
colaboradores[i] ← 1
participantes ← colaboradores
∀i: se (colaboradores[i] = 1 *e* $i \neq$ pid)
req.pid_inic ← pid
req.ind_inic ← VR[pid]
req.ind_receptor ← VR[i]
req.participantes ← participantes
transmite (req) para p_i
se (∀$i \neq$ pid: colaboradores[i] = 0)
bloqueado ← *falso*

Recepção da requisição (req) de p_k:
se (*não* bloqueado)
se (req.ind_receptor = VR[pid])
bloqueado ← *verdadeiro*
VR[req.pid_inic] ← req.ind_inic
VR[pid] ← VR[pid] + 1
armazena o *checkpoint*
∀i: se (VR[i] > ultimo_VR[i] *e* $i \neq$ pid)
colaboradores[i] ← 1
∀i: participantes[i] ← (colaboradores[i] *ou* req.participantes[i])
∀i: se (colaboradores[i] = 1 *e* req.participantes[i] = 0)
p_req.pid_inic ← req.pid_inic
p_req.ind_inic ← req.ind_inic
p_req.ind_receptor ← VR[i]
p_req.participantes ← participantes
transmite (p_req) para p_i
res.participantes ← participantes
res.VR ← VR
transmite (res) para $p_{req.pid\_inic}$
senão // req.ind_receptor < VR[pid]
se (req.ind_inic > VR[req.pid_inic])
VR[req.pid_inic] ← req.ind_inic
res.participantes ← req.participantes
res.VR ← VR
transmite (res) para $p_{req.pid\_inic}$

Recepção da resposta (res) de p_k:
∀i: se (res.VR[i] > VR[i])
VR[i] ← res.VR[i]
respostas[k] ← 1
∀i: participantes[i] ← (participantes[i] *ou* req.participantes[i])
se (∀$i \neq$ pid: participantes[i] = respostas[i])
∀i: se (participantes[i] = 1 *e* $i \neq$ pid)
lib.VR ← VR
transmite (lib) para p_i
∀i: colaboradores[i] ← 0
∀i: participantes[i] ← 0
∀i: respostas[i] ← 0
ultimo_VR ← VR
bloqueado ← *falso*

Recepção da liberação (lib) de p_k:
∀i: se (lib.VR[i] > VR[i])
VR[i] ← lib.VR[i]
∀i: colaboradores[i] ← 0; participantes[i] ← 0
∀i: respostas[i] ← 0
ultimo_VR ← VR
bloqueado ← *falso*

global, a união dos últimos checkpoints de cada processo formará um novo checkpoint global consistente.

Prova: Para estabelecer uma contradição, suponha que após a invocação do protocolo VR-minimal-simples a união dos últimos *checkpoints* de cada processo é inconsistente. Então, existe uma precedência causal direta ou transitiva entre o último *checkpoint* c_a^α de p_a e o último *checkpoint* c_b^β de p_b ($c_a^\alpha \rightarrow c_b^\beta$).

Vamos analisar os casos em que os processos p_a e p_b são ou não participantes da construção global.

- p_a participante — não é possível pois estamos considerando o estado do sistema imediatamente após o fim da construção global e nenhuma mensagem da aplicação poderia ter sido enviada por p_a.
- p_a não participante:
  - p_b participante — de acordo com o protocolo, p_a deveria receber uma mensagem de requisição e gravaria um *checkpoint* posterior a c_a^α.
  - p_b não participante — contraria a hipótese de que antes da invocação do protocolo, a união dos últimos *checkpoints* era consistente. □

Teorema 2 *O protocolo VR-minimal-simples é minimal.*

Prova: Suponha que o iniciador p_i, em uma invocação do protocolo, armazenou um *checkpoint* c_i^ι e construiu um *checkpoint* global consistente $\mathcal{C}$ não minimal formado por c_i^ι, pelos *checkpoints* de seus participantes e pelos *checkpoints* de outros processos. Então, existe um *checkpoint* global consistente $\mathcal{C}'$ que engloba c_i^ι e pelo menos um processo p_j é participante de $\mathcal{C}$ e não de $\mathcal{C}'$.

Na Figura 6 (a) temos que p_i é iniciador e p_j é colaborador de p_i em $\mathcal{C}$ mas não em $\mathcal{C}'$. Mas se p_j é colaborador de p_i em $\mathcal{C}$, existe uma precedência direta ou transitiva entre $c_j^{\gamma-1}$ e c_i^ι e $\mathcal{C}'$ é inconsistente.

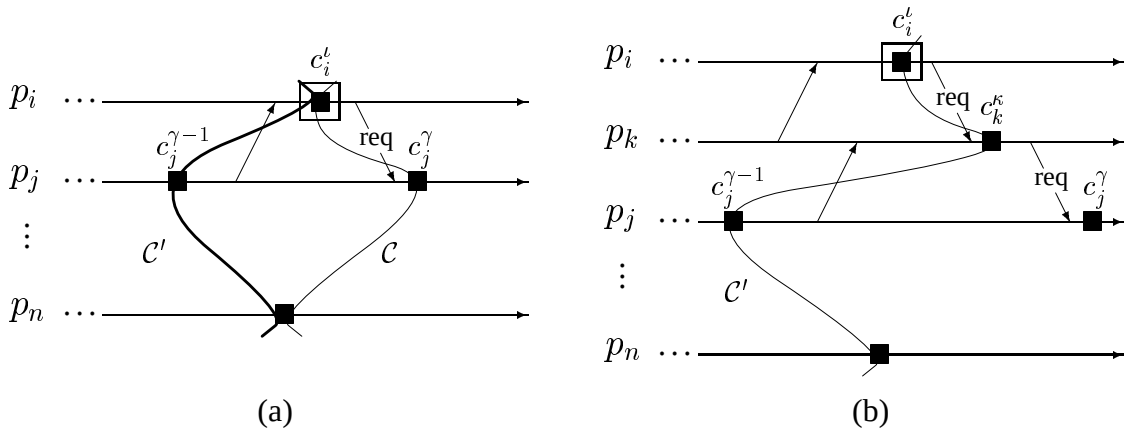


Figura 6: Suposição de não minimalidade

Suponha que p_j não é colaborador de p_i mas é colaborador de um processo p_k que é colaborador de p_i como ilustrado na Figura 6 (b). O último *checkpoint* de p_k deve necessariamente fazer parte de $\mathcal{C}'$ pois caso contrário, haveria uma inconsistência semelhante a descrita acima. Mas se p_j é colaborador de p_k em $\mathcal{C}$, existe uma precedência direta ou transitiva entre $c_j^{\gamma-1}$ e c_k^κ e $\mathcal{C}'$ é inconsistente. Esse raciocínio é válido para um processo que é colaborador do colaborador do iniciador e assim sucessivamente. □

Lema 1 *Cada processo p_j participante de uma construção global envia uma única mensagem de resposta ao processo iniciador p_i desta construção.*

Prova: O processo p_j participante de uma construção global, ao receber uma mensagem de requisição pode estar:

- bloqueado – a mensagem de requisição é ignorada e nenhuma mensagem de resposta é enviada.
- desbloqueado e :
  - grava um *checkpoint* – p_j atualiza o índice do iniciador p_i, fica bloqueado e envia uma resposta para p_i. Toda mensagem de requisição recebida após o seu bloqueio será ignorada.
  - não grava um *checkpoint* – se o índice de p_i não é conhecido por p_j, ou seja, se é a primeira requisição da construção de p_i recebida por p_j, p_j atualiza o índice do iniciador p_i e envia uma resposta para o iniciador. Toda mensagem de requisição recebida após a atualização do índice de p_i será ignorada. □

O protocolo é encerrado quando todos os participantes da construção global recebem uma mensagem de liberação do iniciador. O iniciador por sua vez, só envia mensagens de liberação, quando todos os participantes lhe enviam mensgens de resposta. O iniciador detecta que a construção pode ser encerrada quando seu vetor de participantes é igual ao seu vetor de respostas. Sabemos que todo processo participante envia uma mensagem de resposta ao iniciador (Lemma 1). Para provar que o protocolo termina corretaemente basta provar que o iniciador conhece todos os participantes de sua construção global antes de enviar as mensagens de liberação.

Teorema 3 *O protocolo VR-min-simples termina corretamente uma construção global.*

Prova: Suponha que o iniciador p_i não conhece o participante p_j no fim de sua construção global. O processo p_j portanto não é colaborador de p_i, mas é colaborador de um processo p_k que é participante conhecido por p_i. Então, p_i não encerra o protocolo enquanto p_k não envia uma resposta. Mas a recepção da mensagem de resposta de p_k faz com que p_i conheça o processo p_j. □

Podemos observar que o cenário da Figura 4 utilizado para provar a não minimalidade do protocolo proposto por Prakash e Singhal não ocorre no protocolo VR-minimal-simples como ilustra a Figura 7. Quando o processo p_1 recebe a mensagem requisição de p_2, p_1 não grava um *checkpoint* pois no momento do envio da requisição, p_2 não tinha conhecimento do último *checkpoint* gravado por p_1.

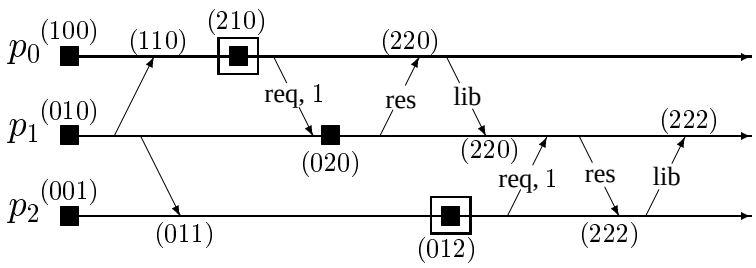


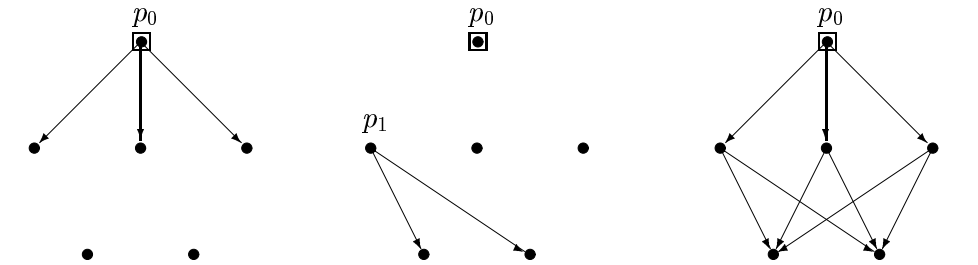
Figura 7: Protocolo VR-minimal-simples

4.4. Análise da Complexidade das Mensagens de Controle

mensagens de controle assumem um papel importante nos protocolos síncronos bloqueantes pois são responsáveis pela construção de um *checkpoint* global consistente e pelo bloqueamento/desbloqueamento dos processos participantes da construção global. Nesta Seção analisamos o número de mensagens de controle utilizado no protocolo VR-minimal-simples.

Notamos que este protocolo utiliza o número mais alto de mensagens de controle quando o iniciador envia requisições para $n/2$ processos e os $n/2$ processos enviam requisições para os processos restantes não colaboradores do iniciador. Este cenário é ilustrado pela Figura 8. Inicialmente, p_0 envia um mensagem de requisição para $n/2$ processos (Figura 8 (a)). Quando p_1 recebe a mensagem de requisição de p_0, p_1 envia mensagens de requisição para seus $n/2 - 1$ colaboradores que não são colaboradores do iniciador (Figura 8 (b)). O mesmo ocorre com outros processos que recebem requisição do iniciador. Temos então que o número de mensagens de requisição é igual a:

$$\boxed{\frac{n}{2} + \frac{n}{2}\left(\frac{n}{2} - 1\right) = \frac{n^2}{4} = 0.25n^2}$$



(a) Requisições de p_0 (b) Requisições de p_1 (c) Total de requisições

Figura 8: Pior caso para o protocolo VR-minimal-simples

Demonstramos pelo Lemma 1 que o protocolo VR-minimal-simples requer no máximo $n - 1$ mensagens de resposta e é exatamente nesta fase que conseguimos um número menor de mensagens de controle comparado ao protocolo proposto por Prakash e Singhal. Na fase de liberações, o protocolo VR-minimal-simples utiliza no máximo $n-1$ mensagens de liberação pois essas mensagens são enviadas diretamente do iniciador aos participantes da construção global. Podemos concluir que o protocolo VR-minimal-simples requer o menor número de mensagens de controle no pior caso (Tabela 1).

Protocolo	Requisições	Confirmações	Respostas	Liberações
Koo e Toueg	$n^2 - n$	–	$n^2 - n$	$n^2 - n$
Leu e Bhargava	$n^2 - n$	$n^2 - n$	$n - 1$	$n - 1$
Prakash e Singhal	$0.25n^2$	–	$0.25n^2$	$n - 1$
VR-minimal-simples	$0.25n^2$	–	$n - 1$	$n - 1$

Tabela 1: Análise do número de mensagens de controle

5. Iniciadores Concorrentes

Em um sistema distribuído, múltiplos processos podem iniciar uma construção global de forma concorrente e independente. Alguns protocolos que consideram múltiplos iniciadores foram propostos na literatura [5, 9, 7, 13].

Spezialetti e Kearns [13] propuseram um protocolo baseado no protocolo proposto por Chandy e Lamport [2], que requer que todo processo armazene seu *checkpoint*. Se um processo já armazenou um *checkpoint* para um iniciador, as requisições de outros iniciadores concorrentes são ignoradas. Os processos podem armazenar os *checkpoints* a partir de diferentes iniciadores e a combinação dos *checkpoints* de cada processo forma um *checkpoint* global consistente, mas não minimal.

Prakash e Singhal propuseram um protocolo no qual múltiplos processos podem iniciar a construção de *checkpoints* globais consistentes, gerando uma linha consistente máxima, ou seja, com os *checkpoints* mais recentes armazenados pelos iniciadores [9]. Portanto, este protocolo também não apresenta a característica minimal.

O protocolo proposto por Koo e Toueg [5] considera a possibilidade do protocolo ser executado com iniciadores concorrentes aplicando a seguinte regra: quando um processo p_j está participando da construção global para o iniciador p_i e recebe uma mensagem de requisição originada pelo iniciador p_k, a construção de p_k é abortada por p_j. Esta regra garante o correto funcionamento do protocolo, porém não há garantia de que, a cada invocação feita pelo iniciador, um *checkpoint* global consistente seja construído. No pior caso, os processos podem abortar todas as construções globais concorrentes.

O protocolo proposto por Leu e Bhargava [7] permite a execução do protocolo com múltiplos iniciadores concorrentes, sobrepondo as árvores que são construídas por cada construção global. Porém, utiliza um número alto de mensagens de controle.

O protocolo VR-minimal-simples com possibilidade de múltiplos iniciadores concorrentes é chamado de **VR-minimal** e está descrito no Algoritmo 2 (as partes análogas ao Algoritmo 1 foram omitidas). Este protocolo mantém a característica minimal e garante a construção de *checkpoints* globais consistentes na presença de iniciadores concorrentes.

Um vetor de bits chamado $inics$ foi criado para indicar os iniciadores das construções globais em execução que cada processo tem conhecimento. Cada iniciador p_i atualiza sua entrada do vetor de iniciadores ($inics_i[i] = 1$) no início da construção global.

Na recepção da mensagem de requisição req de um iniciador p_i, um processo p_j pode estar:

- desbloqueado – a execução ocorre de maneira análoga ao protocolo VR-minimal-simples (p_0 na recepção da primeira requisição na Figura 9).
- bloqueado por outra requisição de p_i – o processo p_j não armazena um novo *checkpoint* e uma mensagem de resposta é enviada de maneira análoga ao protocolo VR-minimal-simples.
- bloqueado por uma requisição de outro iniciador p_k – o processo p_j não armazena um novo *checkpoint* e neste caso, p_k pode ter conhecimento do:
  - penúltimo *checkpoint* de p_j ($req.indice = req.ultimo\_VR_j[i]$).
    O processo p_j propaga a requisição para que a construção global termine de maneira correta (na Figura 9, p_1 propaga a requisição vinda de p_2).

Algoritmo 2: VR-minimal

Variável adicional do processo:
inics ≡ vetor$[0 \ldots n-1]$ de inteiros
inic ≡ booleano

Início da variável adicional:
$\forall i$: inics[i] ← 0

Início da construção global:
bloqueado ← *verdadeiro*
VR[pid] ← VR[pid] + 1
armazena o *checkpoint*
inics[pid] ← VR[pid]
inic ← *true*
$\forall i$: se (VR[i] > ultimo_VR[i])
colaboradores[i] ← 1
participantes ← colaboradores
$\forall i$: se (colaboradores[i] = 1 *e* $i \neq$ pid)
req.pid_inic ← pid
req.ind_inic ← VR[pid]
req.ind_receptor ← VR[i]
req.participantes ← colaboradores
transmite a requisição (req) para p_i
se ($\forall i \neq$ pid: colaboradores[i] = 0)
bloqueado ← *falso*
inic ← *falso*

Recepção da requisição (req) de p_k:
se (*não* bloqueado *e* req.ind_receptor = VR[pid])
ou bloqueado *e*
req.ind_receptor = ultimo_VR[pid])
se (*não* bloqueado)
bloqueado ← *verdadeiro*
VR[pid] ← VR[pid] + 1
armazena o *checkpoint*
$\forall i$: se (VR[i] > ultimo_VR[i] *e* $i \neq$ pid)
colaboradores[i] ← 1
se (inics[req.pid_inic] < req.ind_inic)
inics[req.pid_inic] ← req.ind_inic
$\forall i$: se (colaboradores[i] = 1
ou req.participantes[i] = 1)
participantes[i] ← 1
$\forall i$: se (colaboradores[i] = 1 *e*
req.participantes[i] = 0)
p_req.pid_inic ← req.pid_inic
p_req.ind_inic ← req.ind_inic
p_req.participantes ← participantes
p_req.ind_receptor ← VR[i]
transmite a requisição (p_req) para p_i
res.VR ← VR
res.participantes ← participantes
transmite a resposta (res) para $p_{req.pid\_inic}$
senão
se (inics[req.pid_inic] < req.ind_inic)
inics[req.pid_inic] ← req.ind_inic
res.participantes ← req.participantes
transmite a resp. (res) para $p_{req.pid\_inic}$

Recepção da resposta (res) de p_k:
se (bloqueado)
$\forall i$: se (res.VR[i] > VR[i])
VR[i] ← res.VR[i]
se (res.VR[pid] = VR[pid])
$\forall i$: se (participantes[i] = 1 *ou*
res.participantes[i] = 1)
participantes[i] ← 1
respostas[k] ← 1
$\forall i$: se (participantes[i] = respostas[i] *e*
$i \neq$ pid)
$\forall i$: se (participantes[i] = 1 *e* $i \neq$ pid)
lib.VR ← VR
transmite a liberação (lib) para p_i
$\forall i$: colaboradores[i] ← 0
$\forall i$: participantes[i] ← 0
$\forall i$: respostas[i] ← 0
inic ← *falso*
ultimo_VR ← VR
bloqueado ← *falso*

Recepção da liberação (lib) de p_k:
se (bloqueado)
se (lib.VR[pid] = VR[pid] *e não* inic)
$\forall i$: se (lib.VR[i] > VR[i])
VR[i] ← lib.VR[i]
$\forall i$: colaboradores[i] ← 0
$\forall i$: participantes[i] ← 0
$\forall i$: respostas[i] ← 0
ultimo_VR ← VR
bloqueado ← *falso*

– *checkpoint* anterior ao penúltimo de p_j ($req.indice < req.ultimo\_VR_j[i]$). Implica que o processo emissor da requisição não tinha informação do penúltimo *checkpoint* e uma mensagem de resposta é enviada ao emissor. Por exemplo, na Figura 9, a requisição de p_3 para p_1 é feita por causa de uma precedência anterior a c_1^α e portanto, a mensagem de requisição não é propagada por p_1.

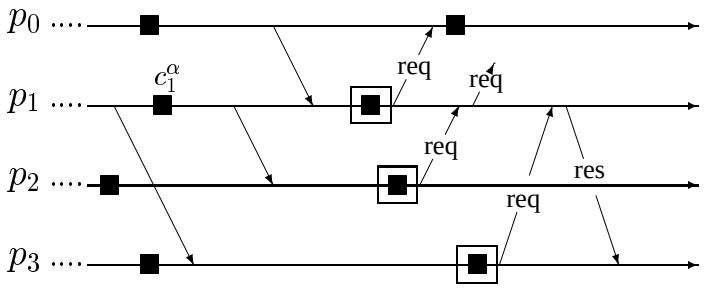


Figura 9: Protocolo com múltiplos iniciadores

Note que mesmo na presença de iniciadores concorrentes os processos podem ser desbloqueados com a primeira mensagem de liberação recebida. Foram acrescentadas condições nos procedimentos de recepção das mensagens de resposta e liberação para evitar o processamento incorreto de mensagens atrasadas.

6. Conclusão

Os protocolos síncronos para *checkpointing* coordenam o armazenamento dos *checkpoints* garantindo a construção de *checkpoints* globais consistentes. Após a ocorrência de uma falha, o sistema pode ser recuperado fazendo com que a aplicação retorne sua execução à última construção global. Os protocolos síncronos minimais reduzem ao máximo o número de *checkpoints* necessários a cada construção global.

Os protocolos presentes na literatura requerem $\mathcal{O}(n^2)$ mensagens de controle tendo havido um esforço no sentido de diminuir a constante multiplicativa do n^2. Verificamos que o protocolo proposto por Prakash e Singhal apesar de requerer um menor número de mensagens de controle, não apresentava a característica minimal [8]. O protocolo proposto neste trabalho diminui o número de mensagens de controle requerido e possui a propriedade minimal. Além disso, propomos uma extensão deste protocolo que permite sua execução com iniciadores concorrentes.

Como trabalho futuro, gostaríamos de analisar protocolos síncronos não bloqueantes pois os protocolos propostos na literatura são muito complexos [1, 10]. Temos como objetivo propor protocolos mais simples mantendo a possibilidade de múltiplos iniciadores concorrentes.

Referências

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

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

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

[4] S. T. Huang. Detecting Termination of Distributed Computations by External Agents. In *9th International Conference on Distributed Computing Systems*, pages 79–84, 1989.

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

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

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

[8] 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.

[9] R. Prakash and M. Singhal. Maximal Global Snapshot with Concurrent Initiators. In *6th IEEE Symposium on Parallel and Distributed Processing*, pages 344–351, oct 1994.

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

[11] T. C. Sakata, I. C. Garcia, and L. E. Buzato. Protocolos Síncronos Bloqueantes Minimais para Checkpointing. Relatório técnico, Instituto de Computação—Universidade Estadual de Campinas, 2003.

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

[13] M. Spezialetti and P. Kearns. Efficient Distributed Snapshots. In *6th International Conference on Distributed Computing Systems*, pages 382–388, 1986.